**Low-Level Design Document**

1. **Introduction**
2. **Architecture Overview**
3. **Component Details**
4. **Data Flow**
5. **Component Design**
6. **Azure Service Integration**
7. **Error Handling and Logging**
8. **Security and Authentication**
9. **Performance Considerations**
10. **Testing Strategy**
11. **Deployment**
12. **Maintenance**
13. **Monitoring**

A Low-Level Design (LLD) document is a detailed description of how a system or a component will be implemented. It's a technical document that provides an in-depth look at the architecture, components, data flow, and interactions of a software system. Below is a template for creating a .NET Azure Low-Level Design document. This example focuses on designing a component that interacts with Azure services.

**Title: Low-Level Design Document - .NET Azure Component**

**1. Introduction**

**1. What Are Microservices?**

* Microservices are a way of structuring software applications as a collection of small, independent services that work together to provide the complete functionality of an application.
* Each microservice is responsible for a specific business capability or function, such as user authentication, payment processing, or product catalog.

**2. Benefits of Microservices:**

* **Scalability:** Microservices can be scaled independently, allowing you to allocate more resources to parts of your application that need it the most.
* **Flexibility:** Teams can develop and update microservices independently, making it easier to adapt to changing business requirements.
* **Fault Isolation:** If one microservice fails, it doesn't necessarily bring down the entire application, enhancing overall system resilience.
* **Technology Diversity:** Different microservices can use different technologies or programming languages, enabling you to choose the best tool for each job.
* **Rapid Deployment:** Microservices can be deployed and updated quickly, reducing time-to-market for new features.

**3. Azure Microservices Architecture:**

* Azure, Microsoft's cloud platform, provides a robust environment for building and deploying microservices.
* **Azure Kubernetes Service (AKS):** AKS is a managed Kubernetes container orchestration service that simplifies the deployment and scaling of microservices containers. It ensures high availability and resilience.
* **Azure Functions:** For serverless computing, Azure Functions allow you to run code in response to events without managing infrastructure. It's great for small, event-driven microservices.
* **Azure API Management:** This service helps you publish, secure, and manage APIs, making it easier to expose your microservices to external partners or customers.
* **Azure Service Fabric:** For more complex microservices scenarios, Azure Service Fabric provides a platform for building and managing stateful or stateless microservices at scale.
* **Azure DevOps:** Azure offers a set of DevOps tools and services to automate the development, testing, and deployment of microservices, ensuring a smooth and efficient software delivery pipeline.

**4. Challenges and Considerations:**

* **Complexity:** While microservices offer flexibility, managing many services can become complex, so it's essential to have a well-defined governance strategy.
* **Data Management:** Handling data consistency and sharing among microservices can be challenging and requires careful design.
* **Monitoring and Security:** Implementing robust monitoring and security practices are critical to maintaining the health and integrity of your microservices architecture.

**5. Business Impact:**

* By adopting Azure microservices architecture, your organization can be more agile, responsive to market changes, and efficient in delivering customer experiences.
* It enables faster development cycles, reduced downtime, and improved scalability, all of which can contribute to increased customer satisfaction and revenue growth.

**2. Architecture Overview:**

* Choose appropriate Azure services that align with the application's needs (Azure App Service, Azure Kubernetes Service, Azure SQL Database, etc.).
* Utilize Azure Availability Zones to distribute resources across physically separate data centers for redundancy.
* Implement auto-scaling to handle varying loads and traffic spikes.
* Use Azure Traffic Manager for global load balancing across regions.
* Leverage Azure Virtual Network for secure communication between components.

**12-factor principles**

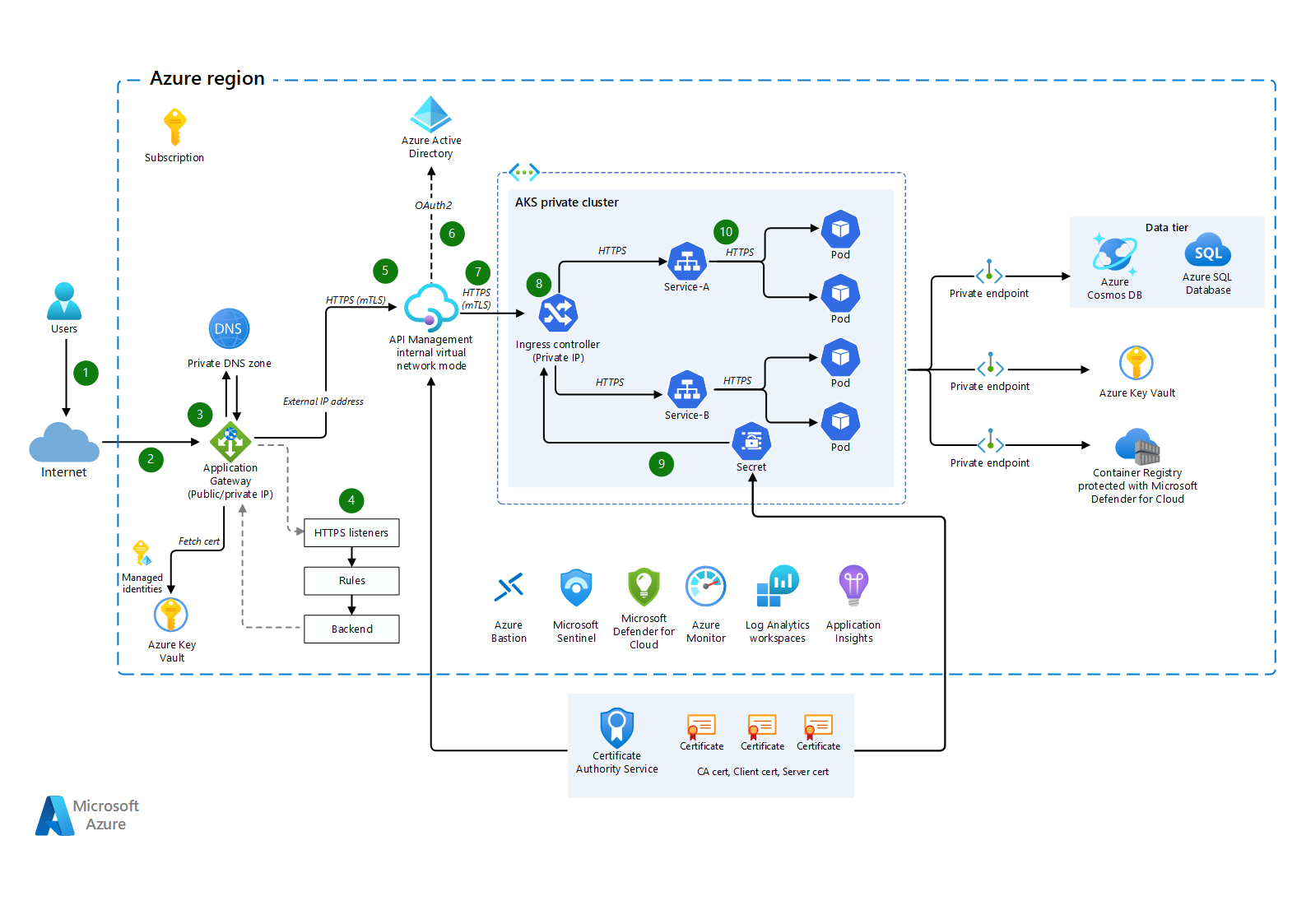
* **Codebase** (One codebase tracked in revision control, many deploys)
* **Dependencies** (Explicitly declare and isolate the dependencies)
* **Config** (Store configurations in an environment)
* **Backing Services** (treat backing resources as attached resources)

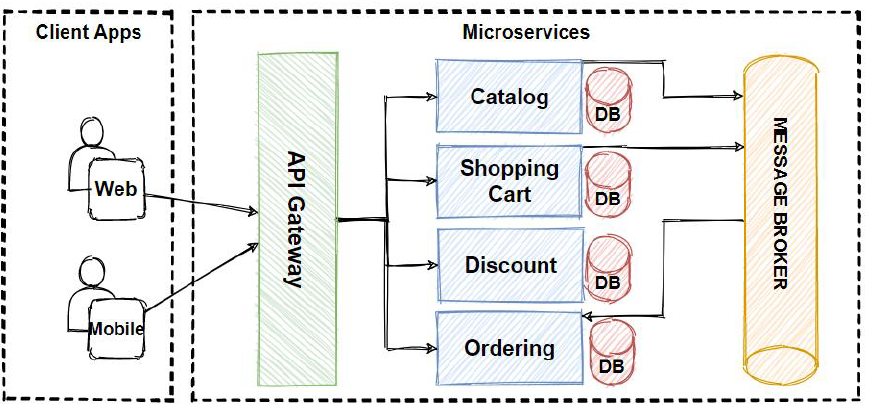
include any service that is communicated with over a network.

Treat backing services as an attached resource.

* **Build, release, and Run** (Strictly separate build and run stages)
* **Processes** (execute the app as one or more stateless processes)
* **Port Binding** (Export services via port binding)
* **Concurrency** - scaling the application (Scale out via the process model)
* **Disposability** - The microservice processes can be started or stopped immediately whenever necessary.
* **Dev/prod parity** (Keep development, staging, and production as similar as possible)
* **Logs** (Treat logs as event streams)
* **Admin processes** (Run admin/management tasks as one-off processes)







Description of how the component interacts with other components and Azure services.

1. A user makes a request to the application endpoint from the internet.
2. Azure Application Gateway receives traffic as HTTPS and presents a PFX certificate previously loaded from Azure Key Vault to the user.
3. Application Gateway uses private keys to decrypt traffic (SSL offload), performs web application firewall inspections, and re-encrypts traffic by using public keys (end-to-end encryption).
4. Application Gateway applies rules and backend settings based on the backend pool and sends traffic to the API Management backend pool over HTTPS.
5. API Management is deployed in internal virtual network mode (Developer or Premium tier only) with a private IP address. It receives traffic as HTTPS with custom domain PFX certificates.
6. Azure Active Directory (Azure AD) provides authentication and applies API Management policies via OAuth and client certificate validation. To receive and verify client certificates over HTTP/2 in API Management, you need to enable **Negotiate client certificate** on the **Custom domains** blade in API Management.
7. API Management sends traffic via HTTPS to an ingress controller for an AKS private cluster.
8. The AKS ingress controller receives the HTTPS traffic and verifies the PEM server certificate and private key. Most enterprise-level ingress controllers support mTLS. Examples include NGINX and AGIC.
9. The ingress controller processes TLS secrets (Kubernetes Secrets) by using cert.pem and key.pem. The ingress controller decrypts traffic by using a private key (offloaded). For enhanced-security secret management that's based on requirements, CSI driver integration with AKS is available.
10. The ingress controller re-encrypts traffic by using private keys and sends traffic over HTTPS to AKS pods. Depending on your requirements, you can configure AKS ingress as HTTPS backend or passthrough.

**3. Component Details:**

• Detailed description of the .NET component's responsibilities and features.

• List of Azure services that the component interacts with (e.g., Azure Blob Storage, Azure SQL Database).

Azure Services:

|  |  |  |
| --- | --- | --- |
|  | Azure App Services | App service plan |
|  | Azure SQL Database | Replication, Disaster recovery, Disk snapshot, failover protection |
|  | Azure Storage | (Blob,table,disk, queue, Fle) |
|  | Azure Service Bus | (event grid, event hub) |
|  | Azure application gateway | (Front door, load balancer, Traffic manager) |
|  | Azure API Management |  |
|  | Azure Kubernetes Service |  |
|  | Azure Container Registries |  |
|  | Azure Private endpoint |  |
|  | Azure CosmosDB |  |
|  | Azure Durable Function |  |
|  | Azure Logic app |  |
|  | Azure KeyVault |  |
|  | Azure application Insight |  |
|  | Azure Log Analytics |  |
|  | Azure Monitor |  |
|  | Microsoft Defender for Cloud |  |
|  | Microsoft Sentinel |  |
|  | Azure Bastion |  |
|  | Azure DNS Service |  |
|  | Azure DevOps |  |
|  | Azure AD(User, Role) |  |
|  | Azure Policies |  |
|  | VNET, Subnet, NSG |  |
|  | Azure Redis Cache |  |
|  | Azure Virtual Machine |  |

**Service Code Flow Details:**

1. When PR raised then validation for Service changes Ready on feature branch. (Followed Code Review checklist, Lint, Unit Testing Code coverage via Sonar Cube tool, Vulnerability, Veracode code scanning etc.
2. After PR approval- Azure DevOps CICD Pipeline triggers for build, artifact (using Docker, ACR, AKS) and deploy on AKS Services via release pipeline with lead or SME approval.

**4. Data Flow:**

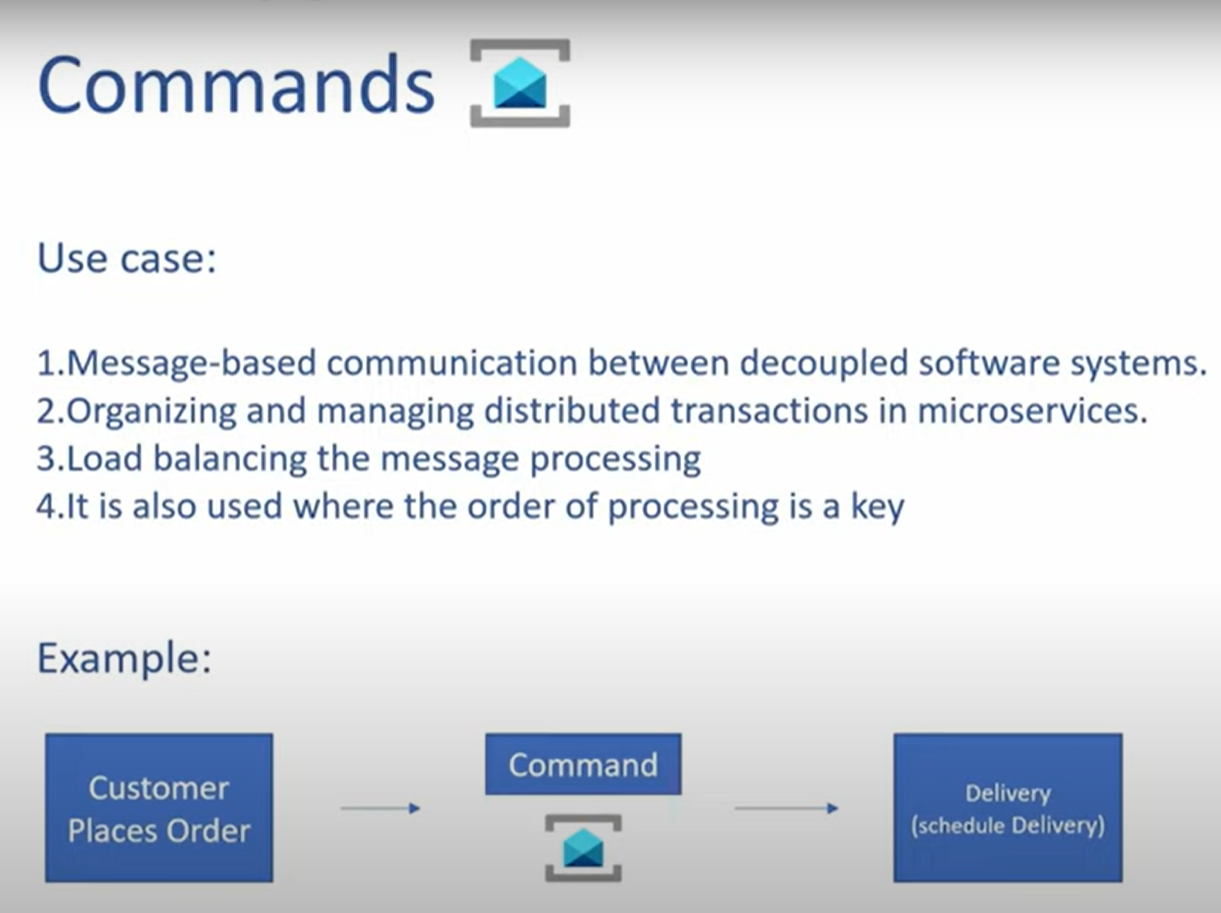
* Sequence diagrams illustrating the flow of data and interactions between the component and Azure services.
* Describe how data is fetched, processed, and stored.

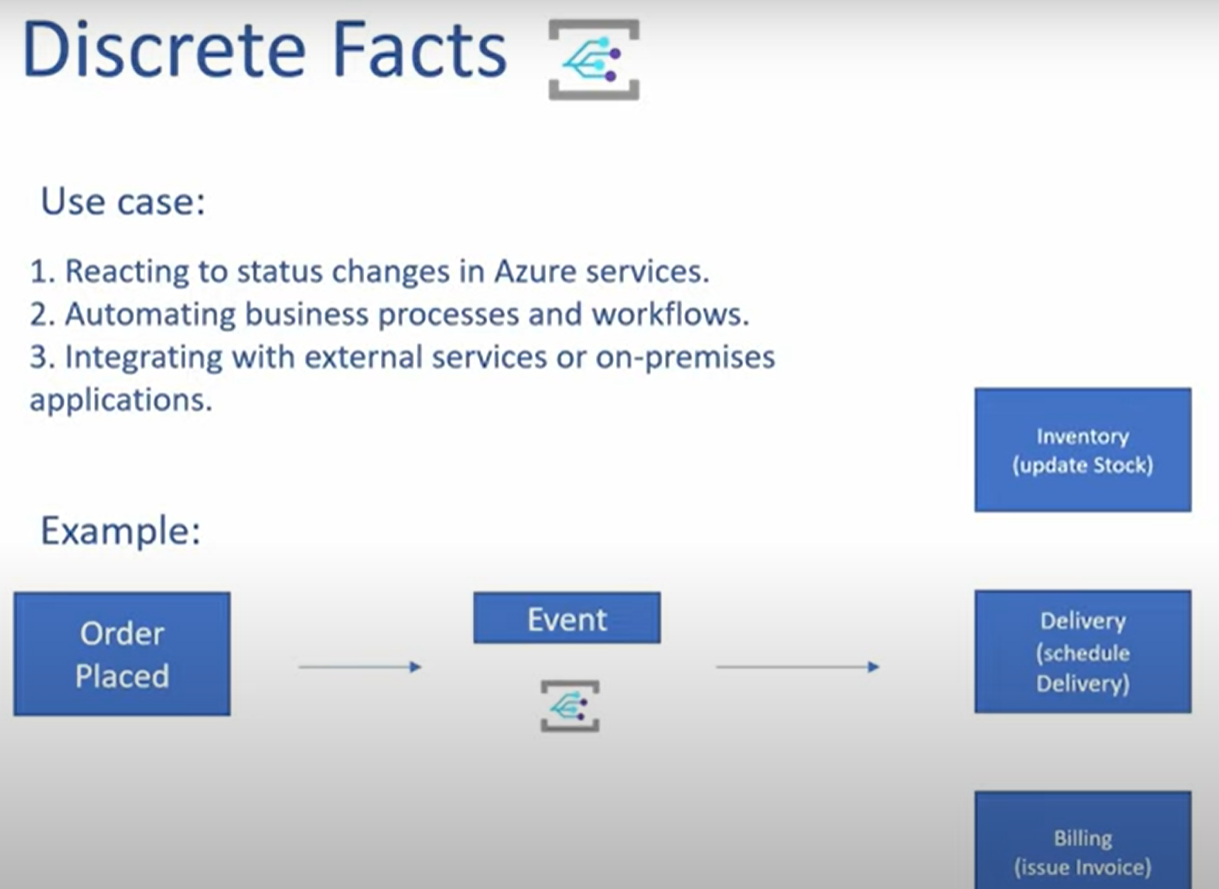
Design a fault-tolerant data storage approach using Azure SQL Database or Azure Cosmos DB with replication across regions.

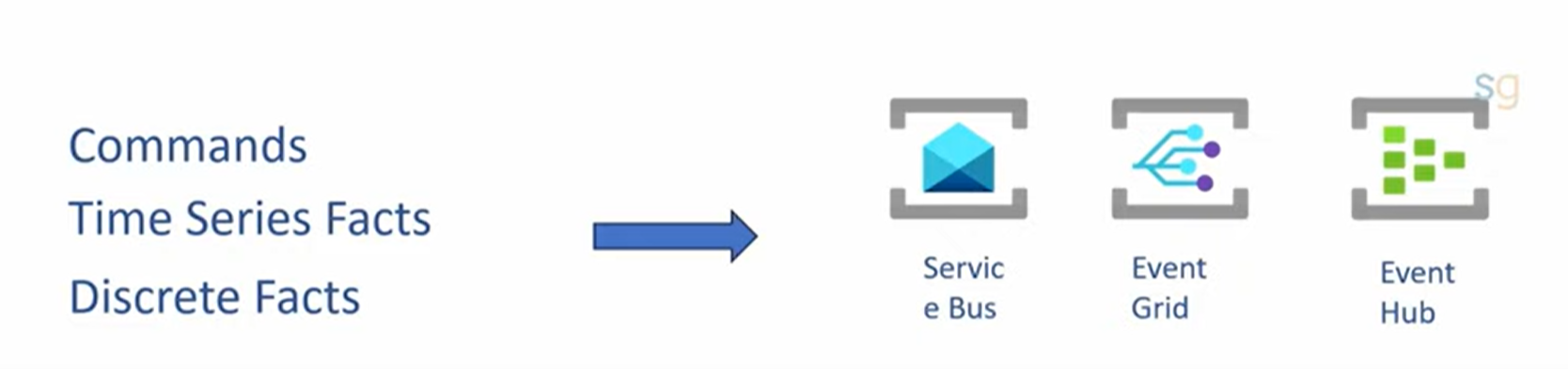
Implement geo-replication and automated backups to ensure data integrity and availability.

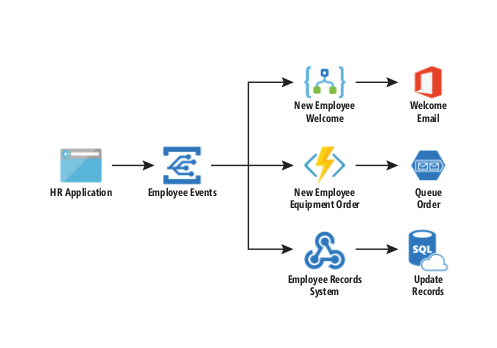
**5. Component Design:**

* Detailed class diagrams showing the internal structure of the .NET component.
* Explanation of each class's purpose, methods, and relationships.



Azure event grid :





Telemetry refers to data emitted from a system, about its behavior. The data can come in the form of **traces, metrics, and logs.**

Azure Event Hub is used more for the telemetry scenarios. Let's say if every component that's been used in the Enterprise for this e-**commerce application emits telemetry data like Log4Net or Log4J and you want to capture it**, then Azure Event Hub can be used.

thumbnail image 1 of blog post titled 
 
 
  
 
 
 
    
  
   
    
      
       Event-Driven on Azure: Part 2 – Architecting event driven applications
       
      
     
   
  
 
   
 
 
 
 
 


**Send and receive messages from an Azure Service Bus queue (.NET)**

<https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-get-started-with-queues?tabs=passwordless>

using Azure.Messaging.ServiceBus;

using Azure.Identity;

// name of your Service Bus queue

// the client that owns the connection and can be used to create senders and receivers

ServiceBusClient client;

// the sender used to publish messages to the queue

ServiceBusSender sender;

// number of messages to be sent to the queue

const int numOfMessages = 3;

// The Service Bus client types are safe to cache and use as a singleton for the lifetime

// of the application, which is best practice when messages are being published or read

// regularly.

//

// Set the transport type to AmqpWebSockets so that the ServiceBusClient uses the port 443.

// If you use the default AmqpTcp, ensure that ports 5671 and 5672 are open.

var clientOptions = new ServiceBusClientOptions

{

TransportType = ServiceBusTransportType.AmqpWebSockets

};

//TODO: Replace the "<NAMESPACE-NAME>" and "<QUEUE-NAME>" placeholders.

client = new ServiceBusClient(

"<NAMESPACE-NAME>.servicebus.windows.net",

new DefaultAzureCredential(),

clientOptions);

sender = client.CreateSender("<QUEUE-NAME>");

// create a batch

using ServiceBusMessageBatch messageBatch = await sender.CreateMessageBatchAsync();

for (int i = 1; i <= numOfMessages; i++)

{

// try adding a message to the batch

if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message {i}")))

{

// if it is too large for the batch

throw new Exception($"The message {i} is too large to fit in the batch.");

}

}

try

{

// Use the producer client to send the batch of messages to the Service Bus queue

await sender.SendMessagesAsync(messageBatch);

Console.WriteLine($"A batch of {numOfMessages} messages has been published to the queue.");

}

finally

{

// Calling DisposeAsync on client types is required to ensure that network

// resources and other unmanaged objects are properly cleaned up.

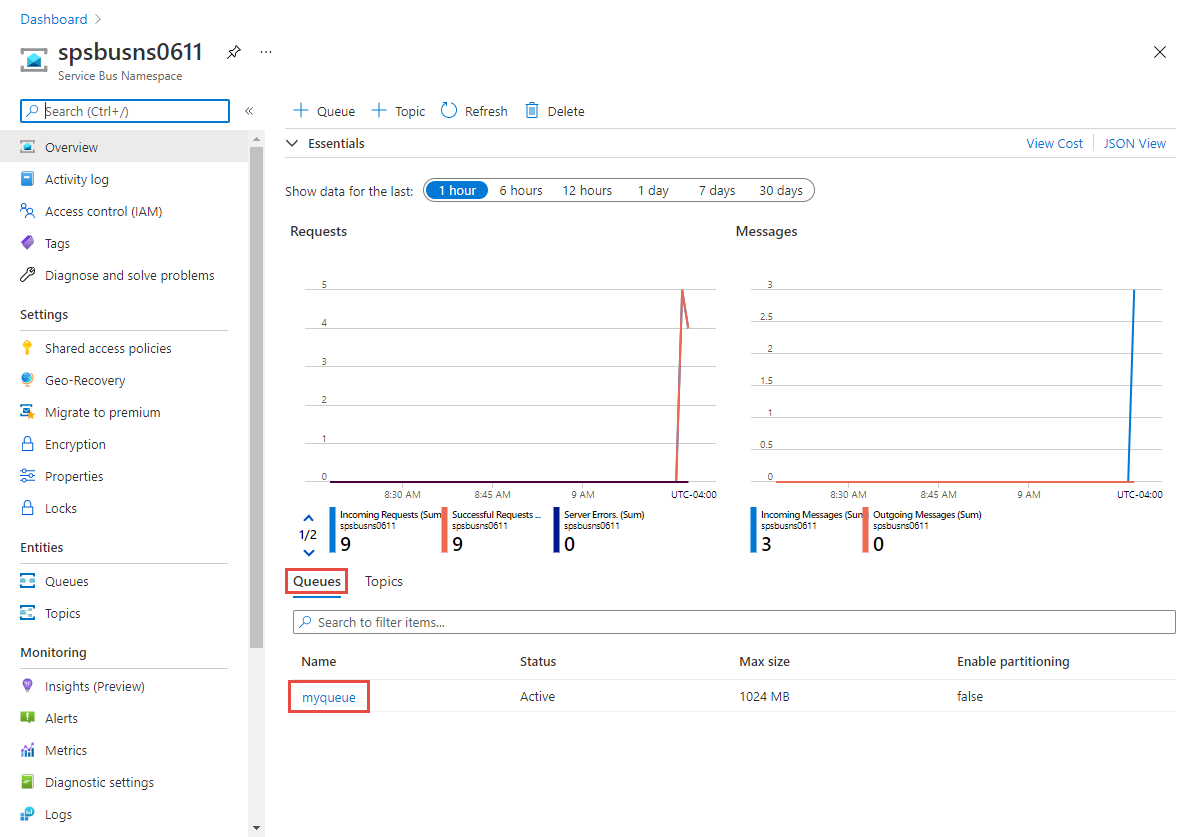
await sender.DisposeAsync();

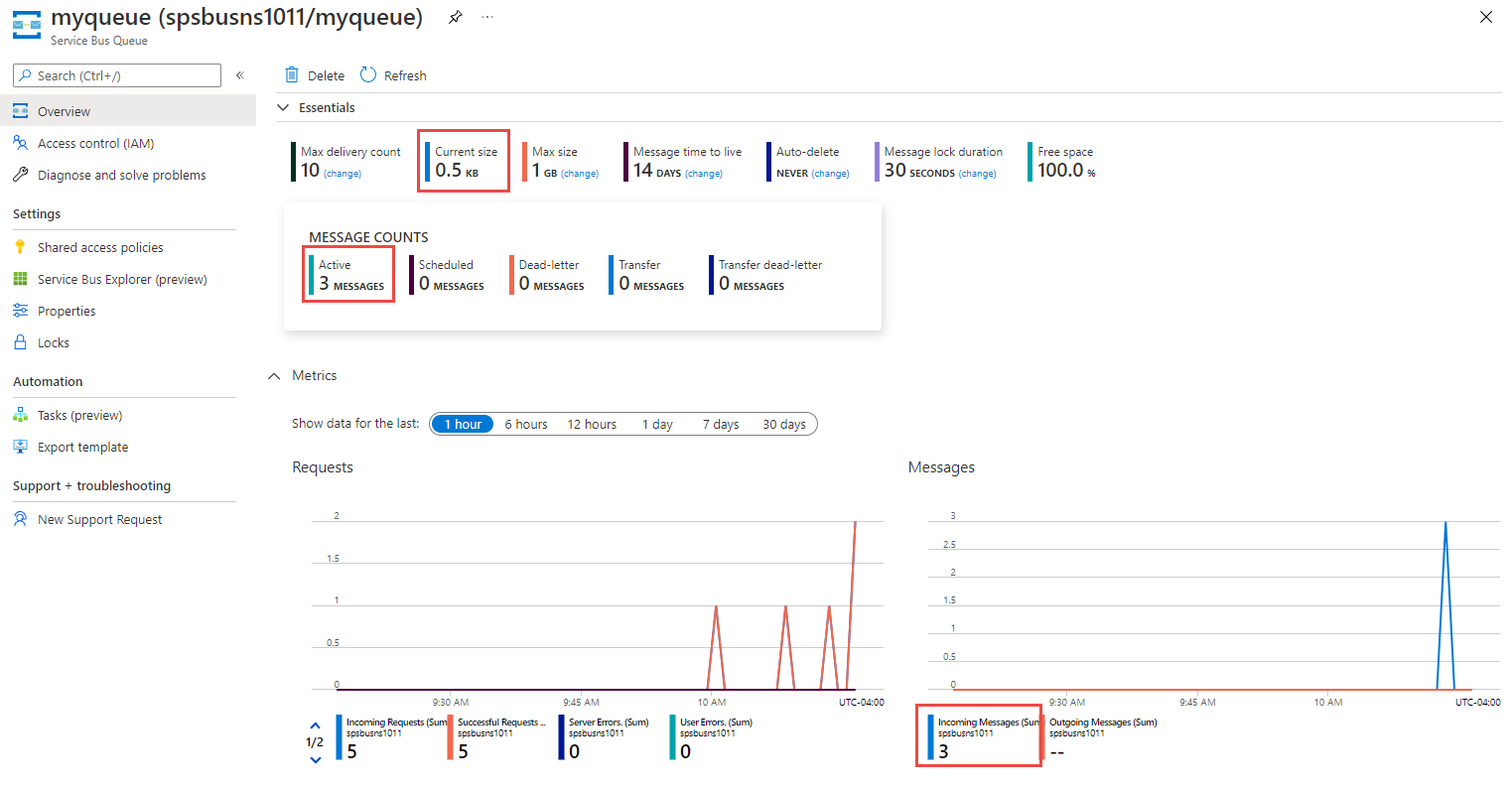
await client.DisposeAsync();

}

Console.WriteLine("Press any key to end the application");

Console.ReadKey();





Receiving Message:

// The Service Bus client types are safe to cache and use as a singleton for the lifetime

// of the application, which is best practice when messages are being published or read

// regularly.

//

// Set the transport type to AmqpWebSockets so that the ServiceBusClient uses port 443.

// If you use the default AmqpTcp, make sure that ports 5671 and 5672 are open.

// TODO: Replace the <NAMESPACE-NAME> placeholder

var clientOptions = new ServiceBusClientOptions()

{

TransportType = ServiceBusTransportType.AmqpWebSockets

};

client = new ServiceBusClient(

"<NAMESPACE-NAME>.servicebus.windows.net",

new DefaultAzureCredential(),

clientOptions);

// create a processor that we can use to process the messages

// TODO: Replace the <QUEUE-NAME> placeholder

processor = client.CreateProcessor("<QUEUE-NAME>", new ServiceBusProcessorOptions());

try

{

// add handler to process messages

processor.ProcessMessageAsync += MessageHandler;

// add handler to process any errors

processor.ProcessErrorAsync += ErrorHandler;

// start processing

await processor.StartProcessingAsync();

Console.WriteLine("Wait for a minute and then press any key to end the processing");

Console.ReadKey();

// stop processing

Console.WriteLine("\nStopping the receiver...");

await processor.StopProcessingAsync();

Console.WriteLine("Stopped receiving messages");

}

finally

{

// Calling DisposeAsync on client types is required to ensure that network

// resources and other unmanaged objects are properly cleaned up.

await processor.DisposeAsync();

await client.DisposeAsync();

}

**6. Azure Service Integration:**

* Details about how the component interacts with Azure services:
  + **Azure Blob Storage:** How the component uploads/downloads files, sets access permissions, etc.
  + **Azure SQL Database:** How the component establishes connections, performs CRUD operations, etc.
  + ... (Other Azure services as applicable)

**7. Error Handling and Logging:**

* Explanation of how errors are handled within the component.
* Description of the logging mechanism used to record component activities and errors.

version: '3.7'

services:

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:7.14.0

container\_name: elasticsearch

environment:

- discovery.type=single-node

- "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"

ports:

- "9200:9200"

kibana:

image: docker.elastic.co/kibana/kibana:7.14.0

container\_name: kibana

ports:

- "5601:5601"

environment:

- "ELASTICSEARCH\_URL=http://elasticsearch:9200"

**Start the Containers:** **docker-compose up -d**

Configuring Elasticsearch, Logstash, and Kibana (ELK) in a microservice architecture involves setting up each component to collect, process, and visualize log data. Below, I'll provide a step-by-step guide on how to configure each component for log management.

**Note**: This guide assumes you have Docker and Docker Compose installed on your system. If you don't want to use Docker, you can install these components manually on your server.

### Step 1: Create a Docker Compose File

Create a **docker-compose.yml** file to define the ELK services:

version: '3.7'

services:

elasticsearch:

image: docker.elastic.co/elasticsearch/elasticsearch:7.14.0

container\_name: elasticsearch

environment:

- discovery.type=single-node

ports:

- "9200:9200"

logstash:

image: docker.elastic.co/logstash/logstash:7.14.0

container\_name: logstash

volumes:

- ./logstash-config:/usr/share/logstash/pipeline

ports:

- "5000:5000"

kibana:

image: docker.elastic.co/kibana/kibana:7.14.0

container\_name: kibana

ports:

- "5601:5601"

environment:

- "ELASTICSEARCH\_URL=http://elasticsearch:9200"

In this configuration, we're using version 7.14.0 of Elasticsearch, Logstash, and Kibana. Adjust the versions as needed.

### Step 2: Create Logstash Configuration

Create a Logstash configuration file (**logstash.conf**) in a directory called **logstash-config** next to your **docker-compose.yml**:

input {

tcp {

port => 5000

codec => json

}

}

output {

elasticsearch {

hosts => "elasticsearch:9200"

index => "microservice-logs-%{+YYYY.MM.dd}"

}

}

This configuration sets Logstash to listen on port 5000 for JSON-formatted logs and sends them to Elasticsearch with an index named based on the date.

### Step 3: Start ELK Stack

Run the following command to start the ELK stack:

bashCopy code

**docker-compose up -d**

This will start Elasticsearch, Logstash, and Kibana in detached mode (-d).

### Step 4: Configure Microservices to Send Logs

Configure your microservices to send their logs to the Logstash instance, which is listening on port 5000.

### Step 5: Access Kibana

Access Kibana by opening a web browser and navigating to **http://localhost:5601** (or the IP address/hostname of your server if not running locally).

### Step 6: Set Up Index Patterns in Kibana

* In Kibana, go to the "Management" section.
* Under "Kibana", select "Index Patterns."
* Create an index pattern (e.g., **microservice-logs-\***) to match your Logstash index.
* Define the timestamp field.

### Step 7: Explore and Visualize Logs

You can now explore and visualize your microservices' log data using Kibana's Discover, Visualize, and Dashboard features.

### Step 8: Secure Your ELK Stack (Optional)

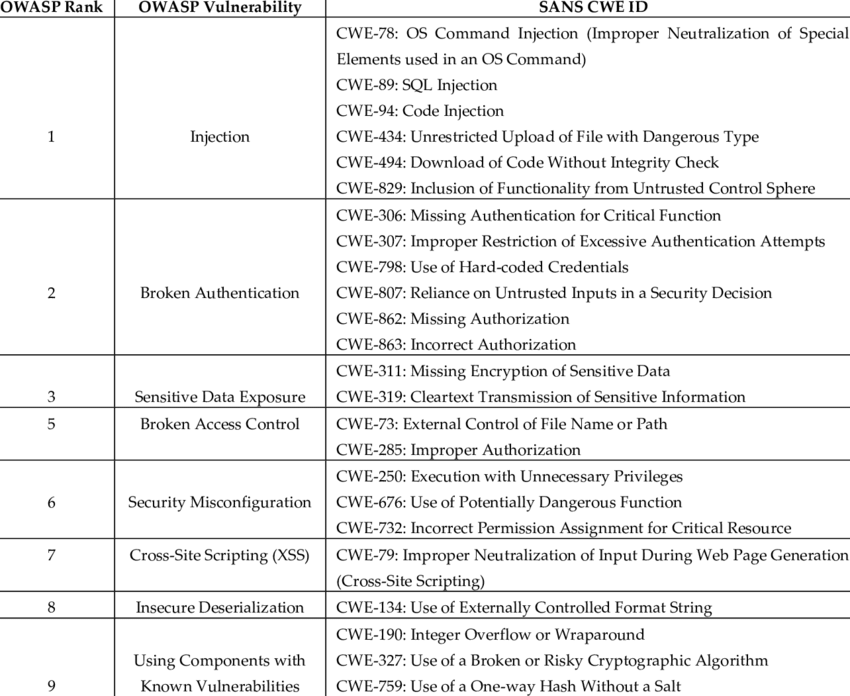
In a production environment, consider securing your ELK stack. This includes enabling authentication, authorization, and encrypting data in transit.

Remember that this is a basic setup for log management in a development or testing environment. In production, you would need to consider scaling Elasticsearch, setting up backups, and implementing additional security measures.

**8. Security and Authentication:**

* Explanation of how authentication and authorization are implemented for Azure services.
* Discuss any encryption, secure connections, or token-based authentication.

 Top 10 Security Vulnerabilities by OWASP



**Authentication Using IdentityFramework:**

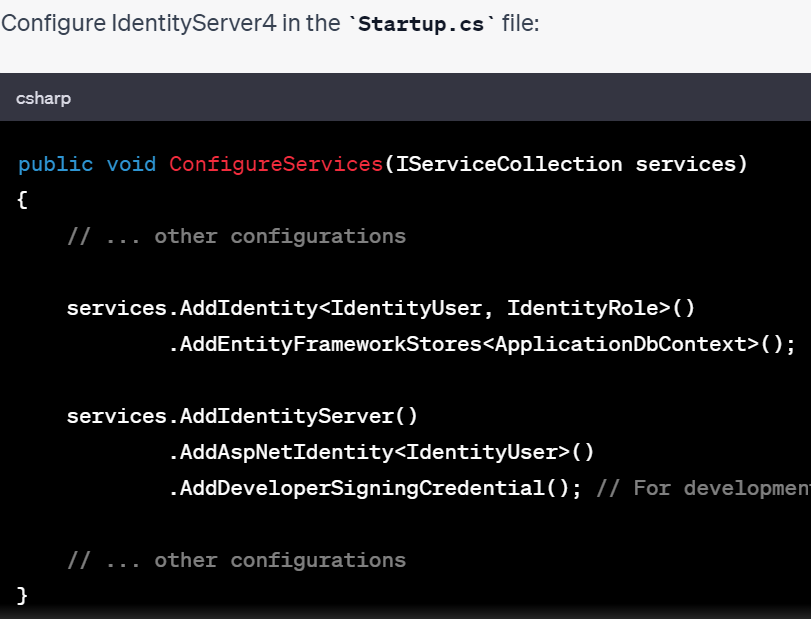
**Step 1: Create IdentityServer4 Application**

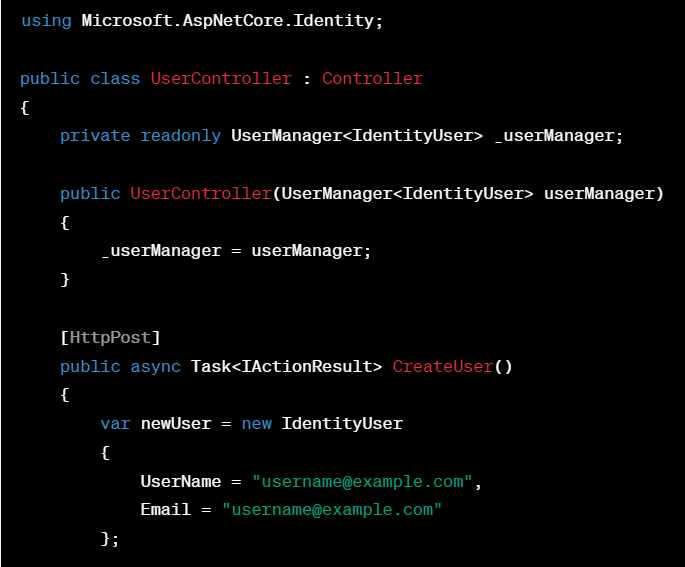
* Install-Package IdentityServer4
* Install-Package IdentityServer4.AspNetIdentity

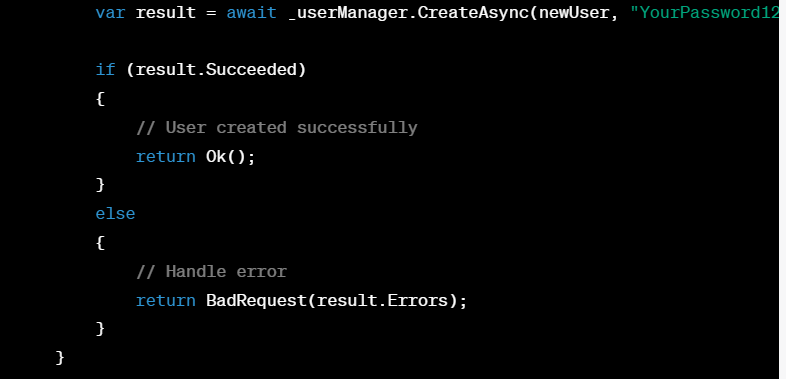
Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore

**1. Set Up Azure AD and App Service:**

* Set up an Azure Active Directory (AD) and register an application.
* Configure your Azure App Service to use Azure AD for authentication.







**Identity Server:**

var tokenResponse = await client.RequestClientCredentialsTokenAsync(new ClientCredentialsTokenRequest

{

Address = discoveryDocument.TokenEndpoint,

ClientId = "client",

ClientSecret = "secret",

Scope = "api1"

});

if (tokenResponse.IsError)

{

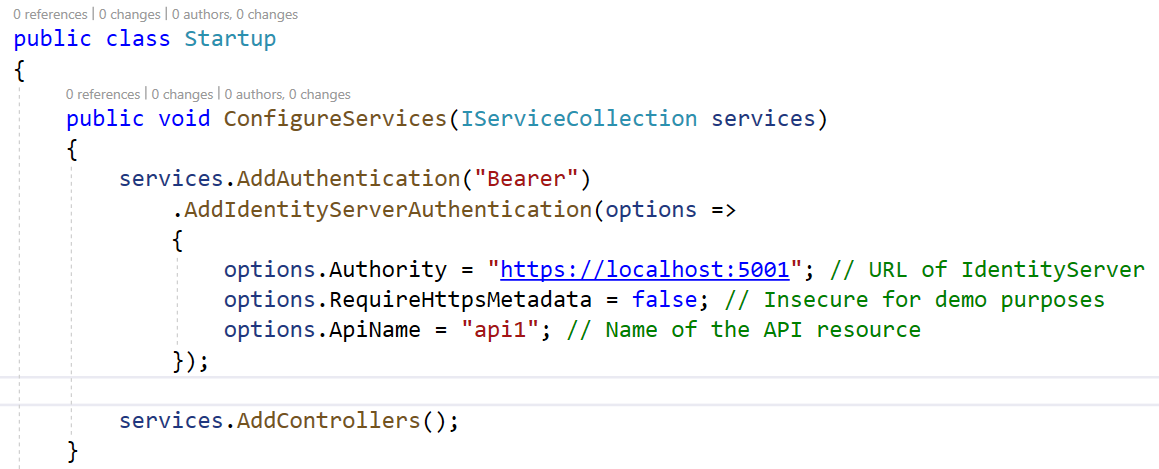
Console.WriteLine($"Error: {tokenResponse.Error}");

return;

}

Console.WriteLine($"Access Token: {tokenResponse.AccessToken}");

**API Call :**



using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

namespace Api.Controllers

{

[Authorize("ApiScope")]

[ApiController]

[Route("[controller]")]

public class WeatherForecastController : ControllerBase

{

using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace Api

{

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

// ...

**services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)**

**.AddJwtBearer(options =>**

**{**

**options.Authority = "https://your-identity-server-url";**

**options.Audience = "api-resource"; // Should match the audience claim in the token**

**});**

**services.AddAuthorization(options =>**

**{**

**options.AddPolicy("ApiScope", policy =>**

**{**

**policy.RequireAuthenticatedUser();**

**policy.RequireClaim("scope", "api-scope"); // Should match the scope claim in the token**

**});**

**});**

// ...

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

}

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Authorization;

using Microsoft.Extensions.DependencyInjection;

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.Authority = "https://your-identity-server-url";

options.Audience = "api-resource"; // Should match the audience claim in the token

});

services.AddAuthorization(options =>

{

options.AddPolicy("ApiScope", policy =>

{

policy.RequireAuthenticatedUser();

policy.RequireClaim("scope", "api-scope"); // Should match the scope claim in the token

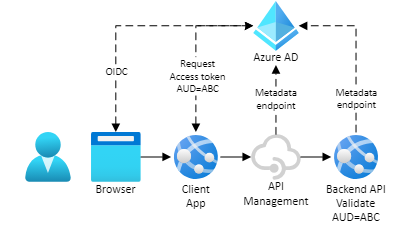
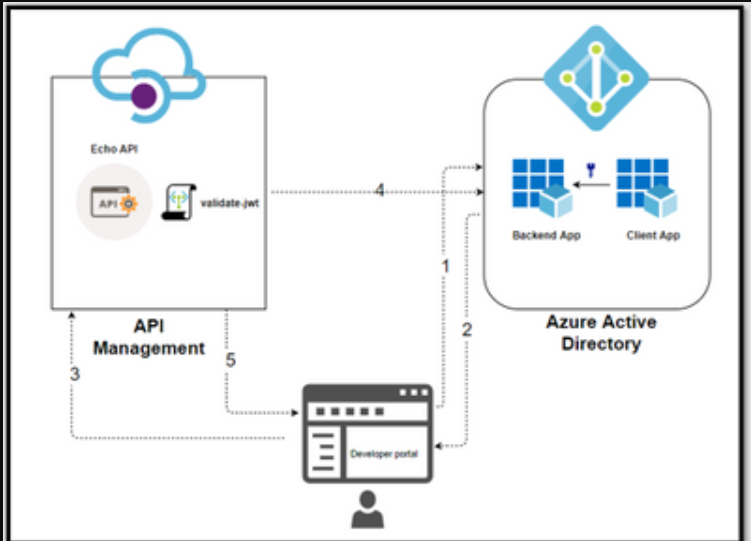
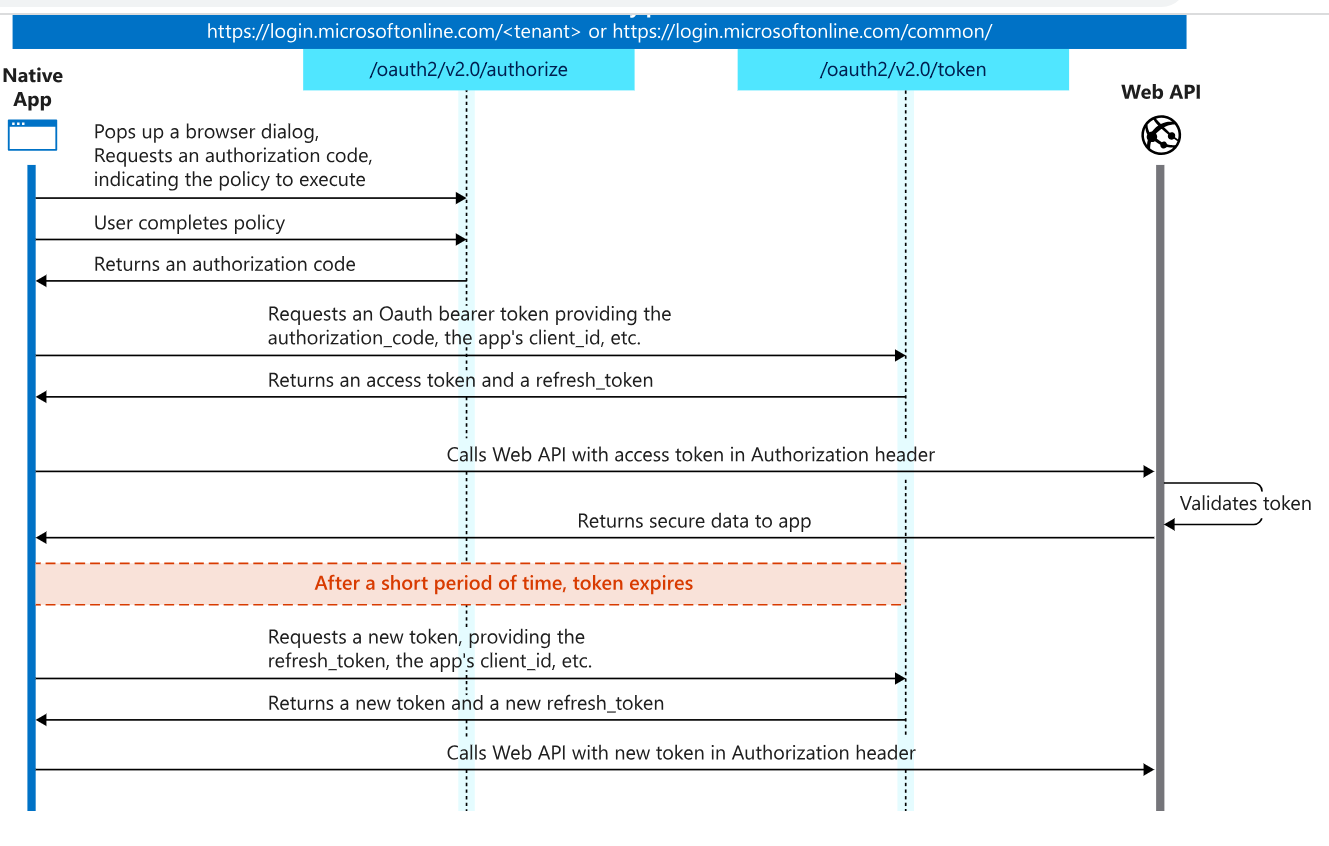
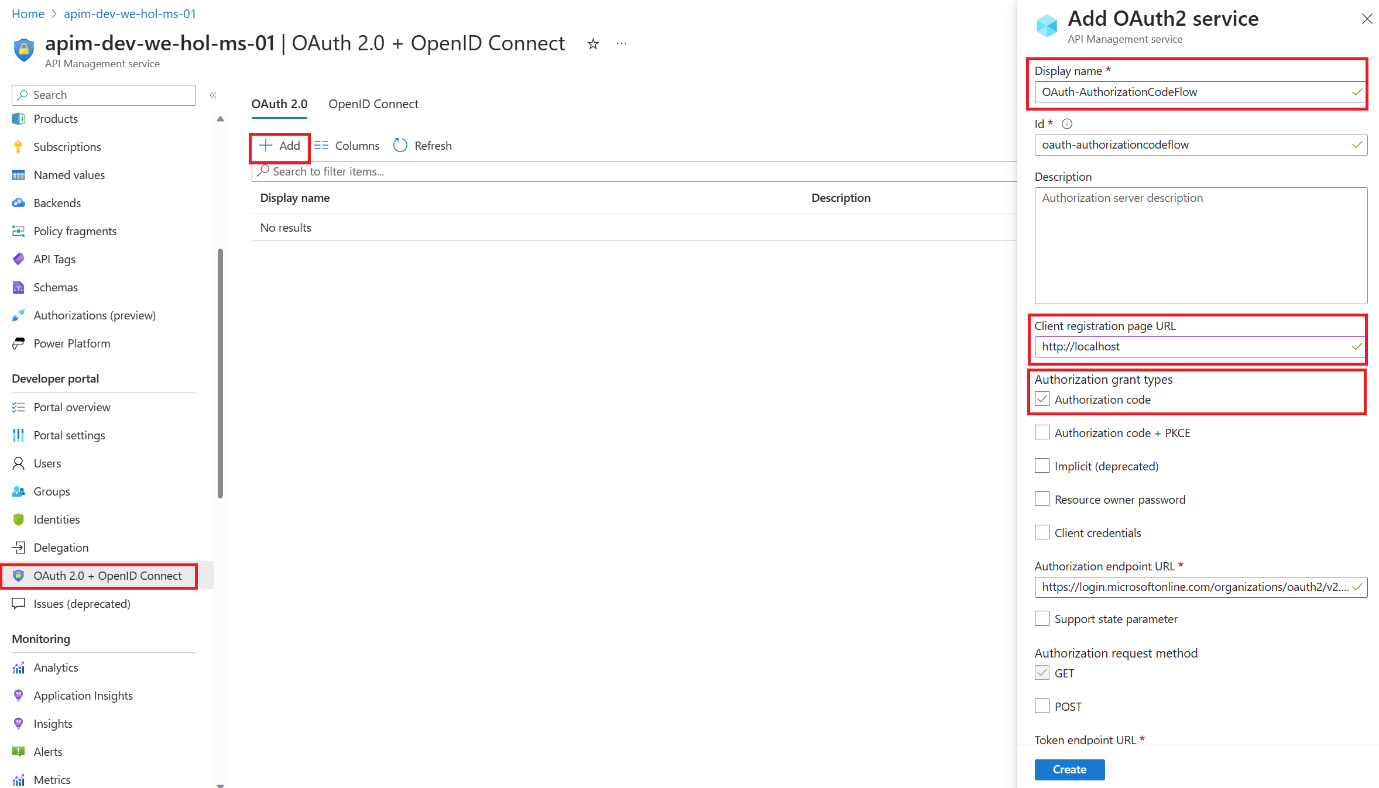
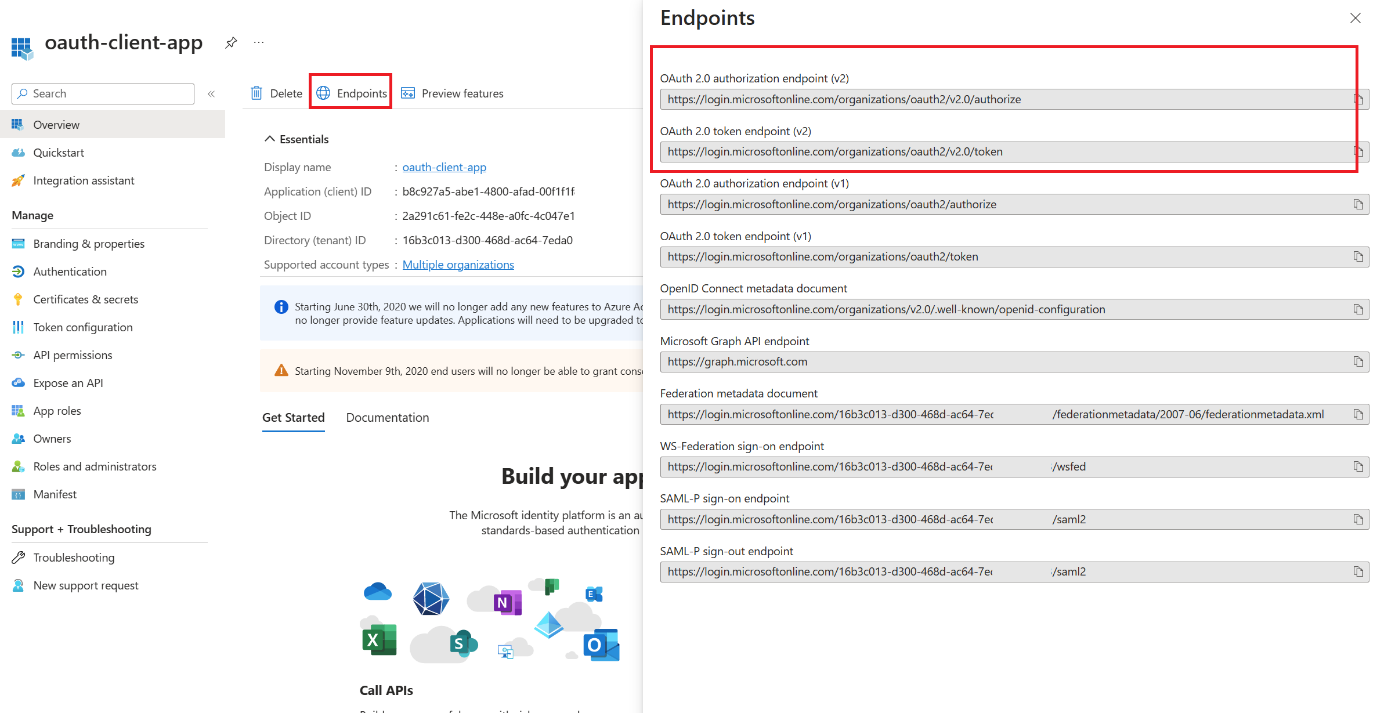
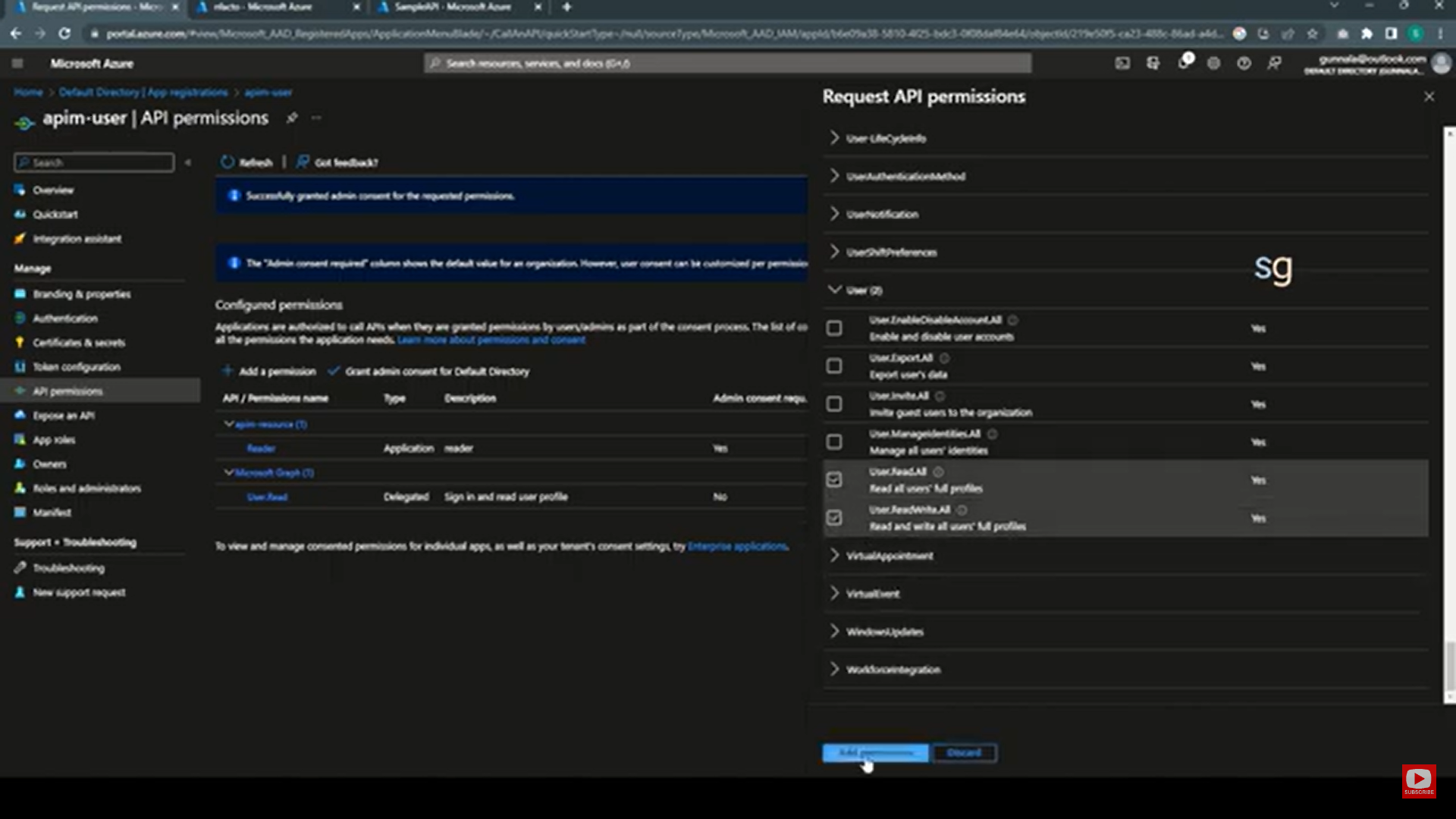
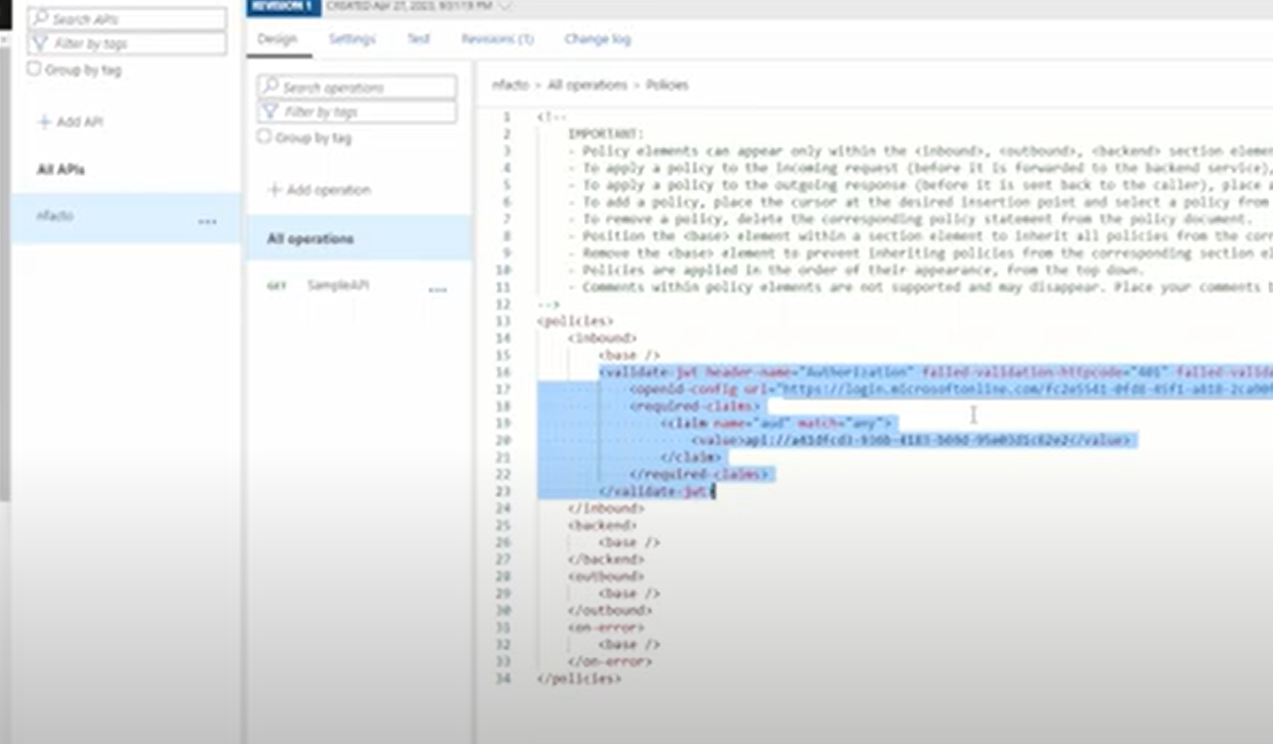
});

});

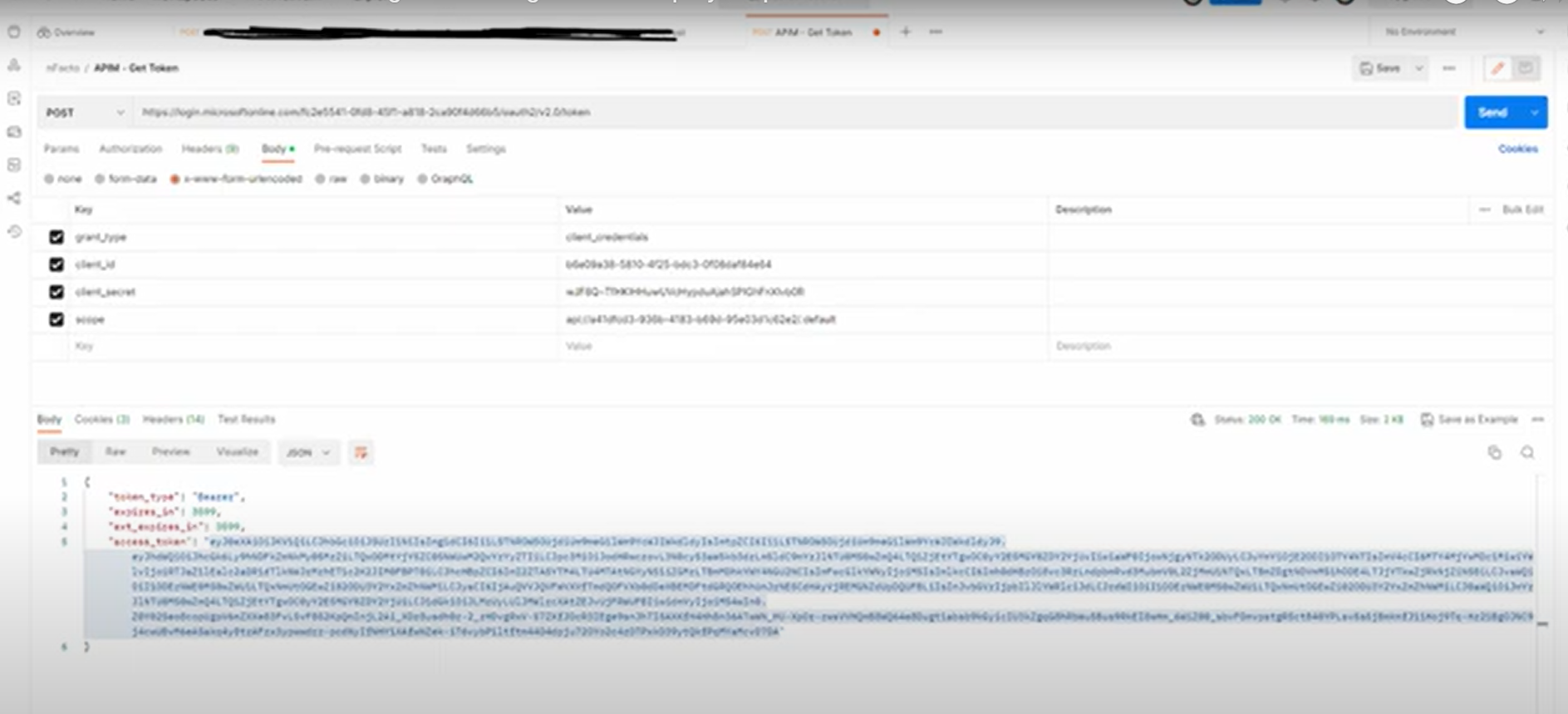
// ...

}

**2. Configure Azure API Management:**

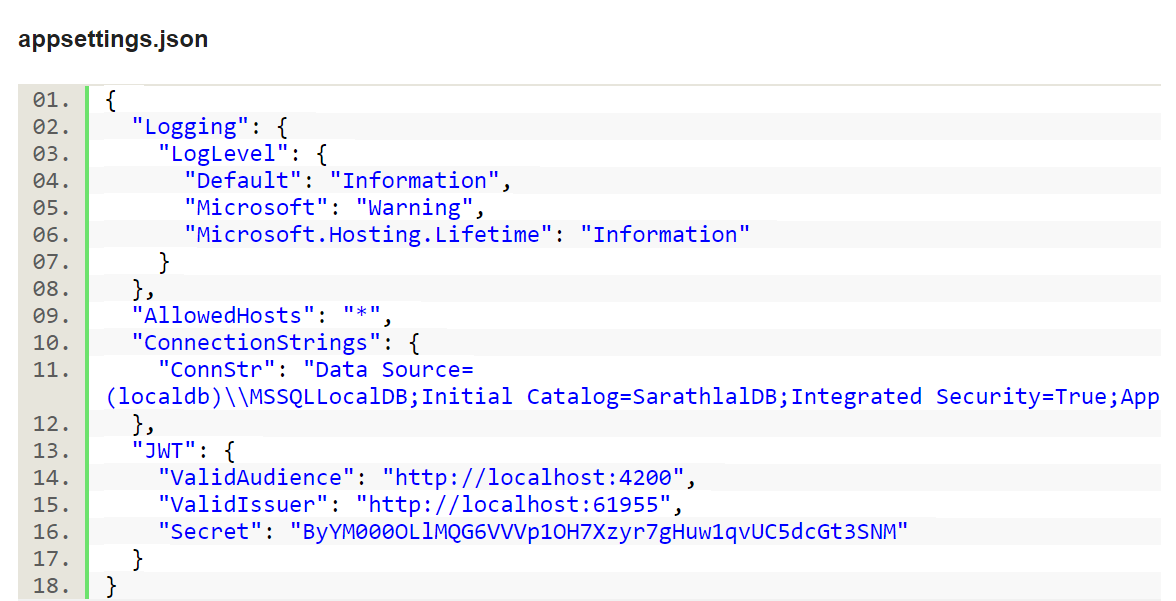
* Set up an Azure API Management service.
* Import the API from your Azure App Service into APIM.
* 
* 
* 
* 
* 
* 
* 
* <https://azure.github.io/apim-lab/apim-lab/7-security/security-7-2-3-oauth2-authorization-grant-flow.html>

1. **Obtain JWT Token:** In Postman, you can follow these steps to obtain a JWT token:

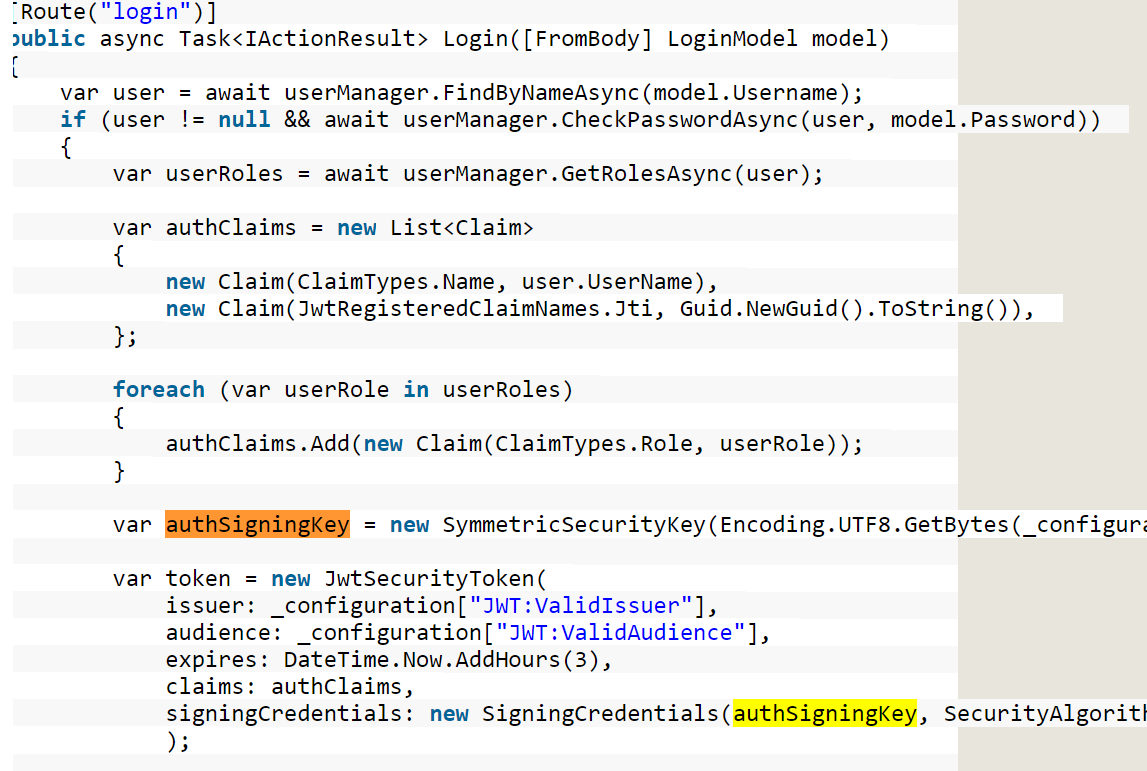


2nd way (MVC web app and web API)

https://www.c-sharpcorner.com/article/authentication-and-authorization-in-asp-net-core-web-api-with-json-web-tokens/



**var authSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration["JWT:Secret"]));**



**return** Ok(**new**

{

token = **new** JwtSecurityTokenHandler().WriteToken(**token**),

expiration = token.ValidTo

});



**9. Performance Considerations:**

* Discuss any performance bottlenecks anticipated and how they are addressed.
* Consider optimizations such as caching, asynchronous operations, etc.

Performance tool:

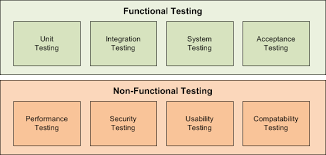
1. ReSharper(ReSharper is a code analysis and debugging tool available as an extender to Visual Studio. )
2. Lint (Formatting)
3. Sonar Lint
4. [SonarQube Code Profiling and Static Code Analysis](https://stackoverflow.com/questions/67263183/sonarqube-code-profiling-and-static-code-analysis)
5. Veracode (Code scanning)
6. Snyk
7. OWASP

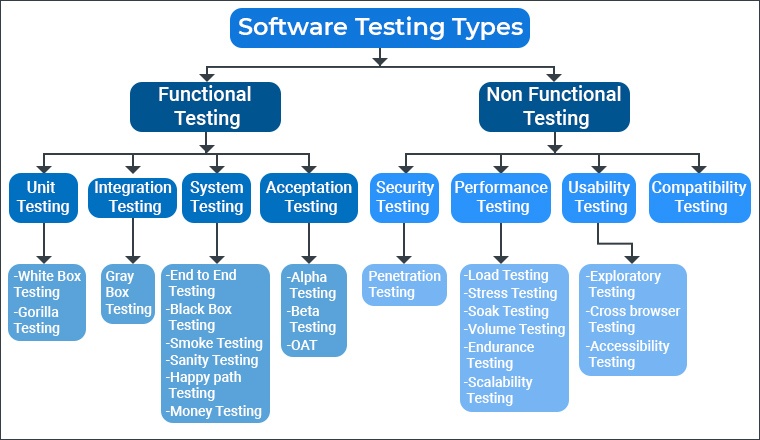
* [Comparison of the Best Extensions for Visual Studio](https://www.softwaretestinghelp.com/visual-studio-extensions/#Comparison_of_the_Best_Extensions_for_Visual_Studio)
* [#1) SonarLint](https://www.softwaretestinghelp.com/visual-studio-extensions/#1_SonarLint)
* [#2) Visual Assist](https://www.softwaretestinghelp.com/visual-studio-extensions/#2_Visual_Assist)
* [#3) Resharper](https://www.softwaretestinghelp.com/visual-studio-extensions/#3_Resharper)
* [#4) Prettier](https://www.softwaretestinghelp.com/visual-studio-extensions/#4_Prettier)
* [#5) Visual Studio Spell Checker](https://www.softwaretestinghelp.com/visual-studio-extensions/#5_Visual_Studio_Spell_Checker)
* [#6) Code Maid](https://www.softwaretestinghelp.com/visual-studio-extensions/#6_Code_Maid)
* [#7) VS Color Output](https://www.softwaretestinghelp.com/visual-studio-extensions/#7_VS_Color_Output)
* [#8) Visual Studio IntelliCode](https://www.softwaretestinghelp.com/visual-studio-extensions/#8_Visual_Studio_IntelliCode)
* [#9) SQLite and SQL Server Compact Toolbox](https://www.softwaretestinghelp.com/visual-studio-extensions/#9_SQLite_and_SQL_Server_Compact_Toolbox)
* [#10) SlowCheetah](https://www.softwaretestinghelp.com/visual-studio-extensions/#10_SlowCheetah)
* [#11) OzoCode](https://www.softwaretestinghelp.com/visual-studio-extensions/#11_OzoCode)

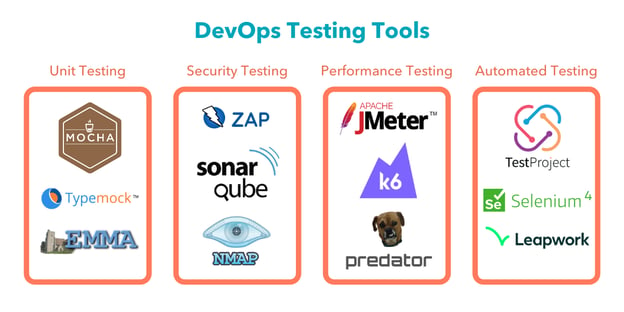
**The following are the best extensions for Visual Studio:**

1. [**SonarLint**](https://www.sonarsource.com/products/sonarlint/?utm_medium=paid&utm_source=sth&utm_campaign=sonarlint&utm_content=visualstudioextension-article)
2. [**Visual Assist**](https://www.wholetomato.com/?utm_source=SoftwareTestingHelp&utm_medium=Leads%20Acquisition&utm_content=Product-listing&utm_campaign=Product-listing)
3. Resharper
4. Prettier
5. Visual Studio Spell Checker
6. Code Maid
7. VS Color Output
8. Visual Studio IntelliCode
9. SQLite and SQL Server Compact Toolbox
10. SlowCheetah
11. OzoCode

**Testing Strategy:** - Describe how the component will be tested, including unit tests, integration tests, and Azure-specific testing (e.g., using Azure DevOps pipelines).





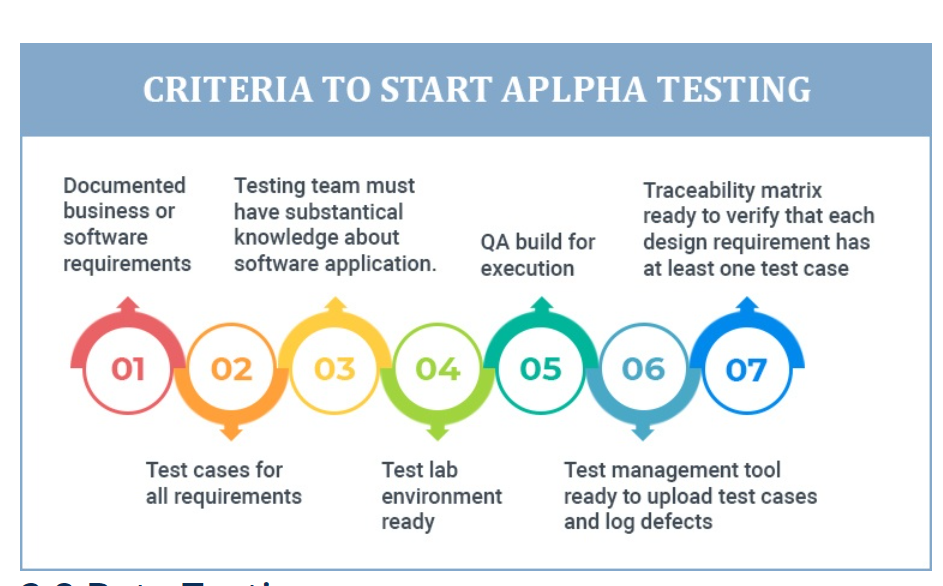


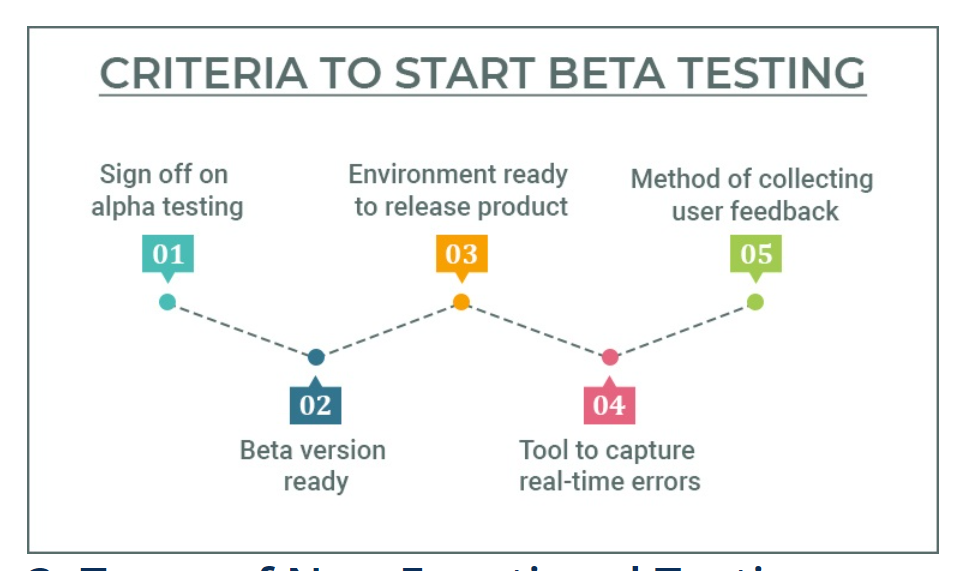
Unit testing – Nunit

Security Testing- SonarQube

Performance testing – Jmeter

Automation testing - Selenium





Software testing URL (Share with Developer and tester)

<https://www.tatvasoft.com/outsourcing/2021/09/types-of-software-testing.html>

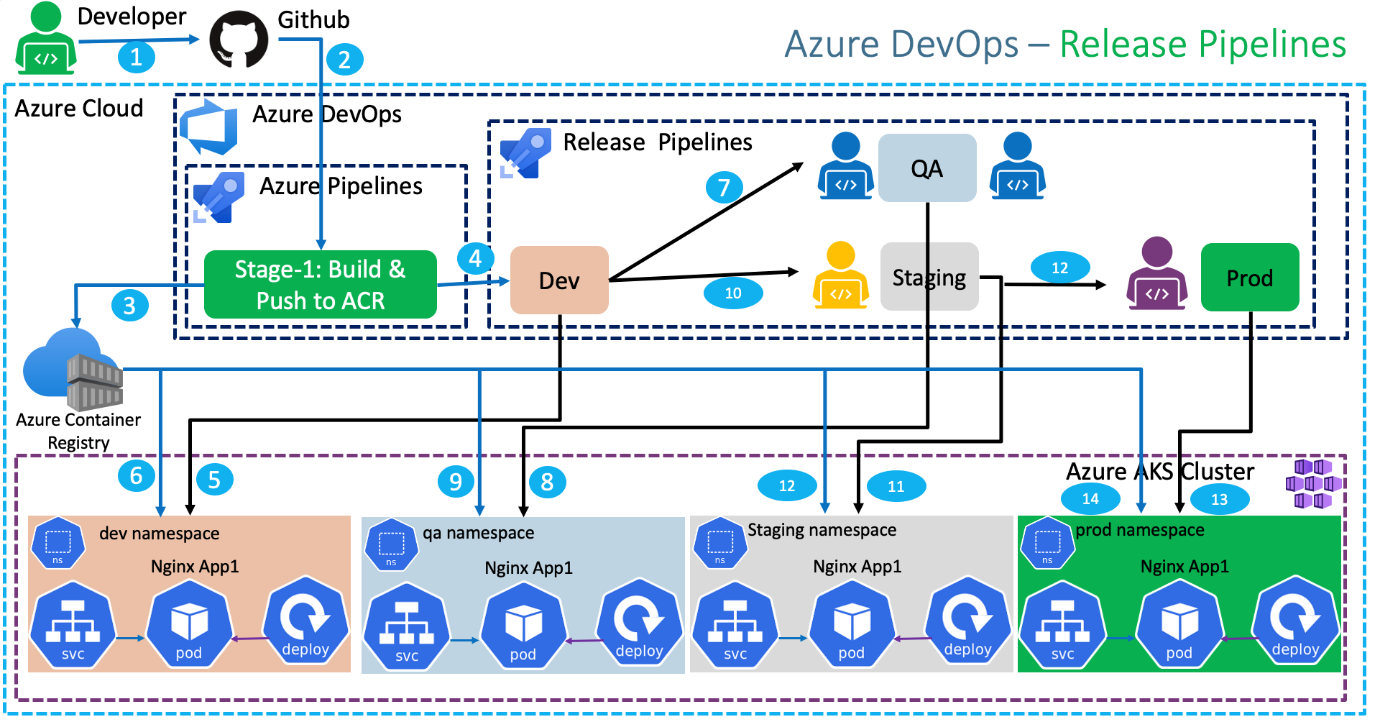
1. **Deployment:** - Describe the process of deploying the component to Azure, including any required configurations. - Discuss considerations for scalability and high availability.

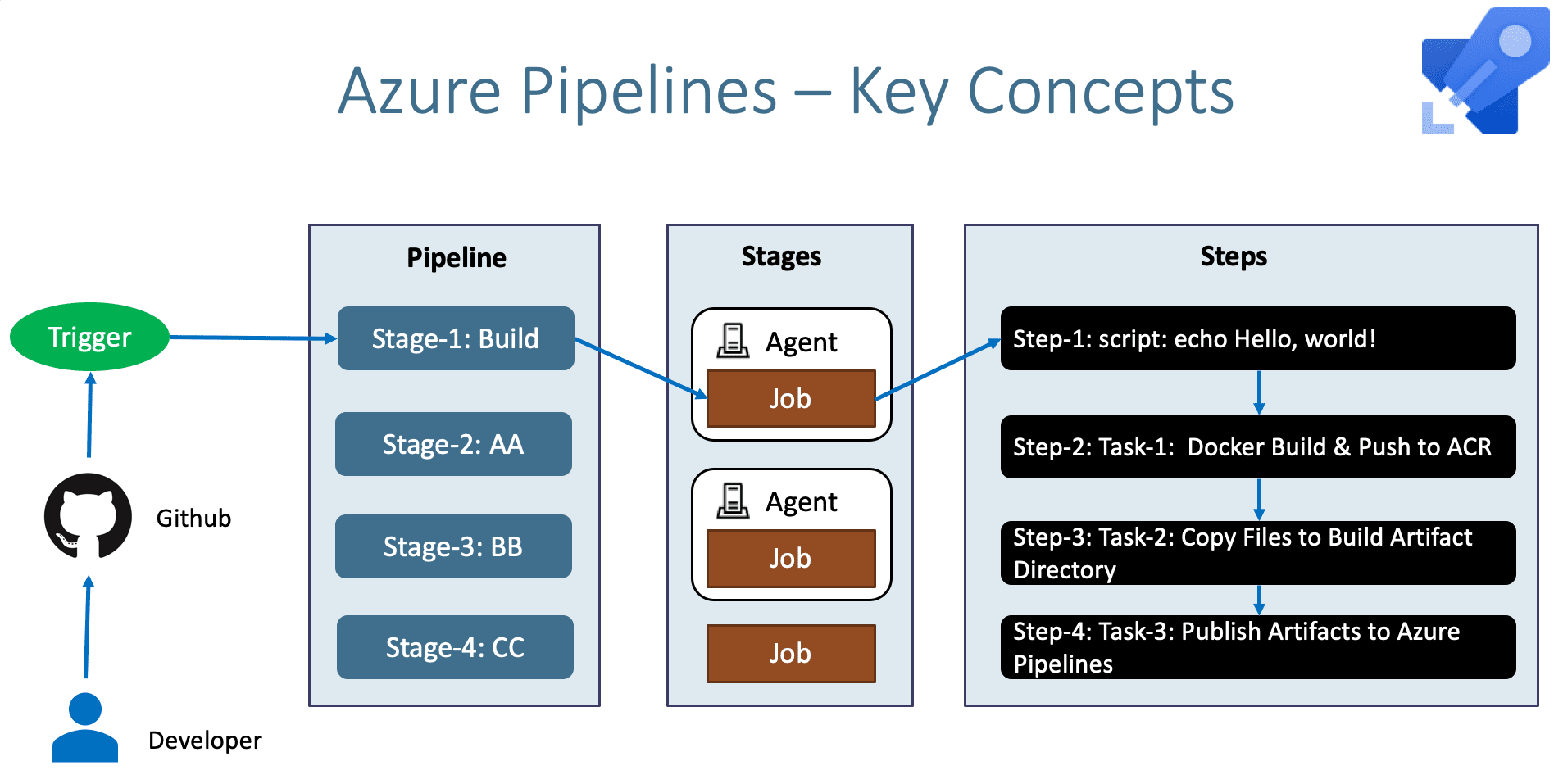


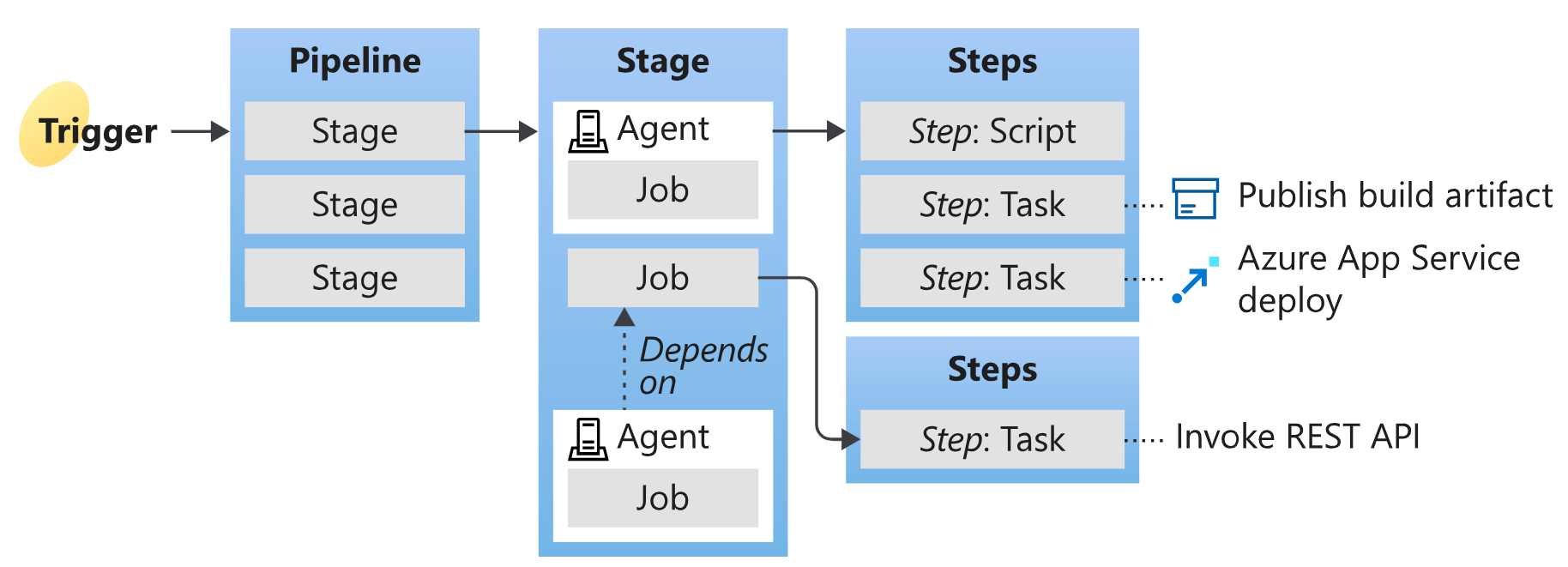
 scheduler is responsible for assigning work to the various nodes.

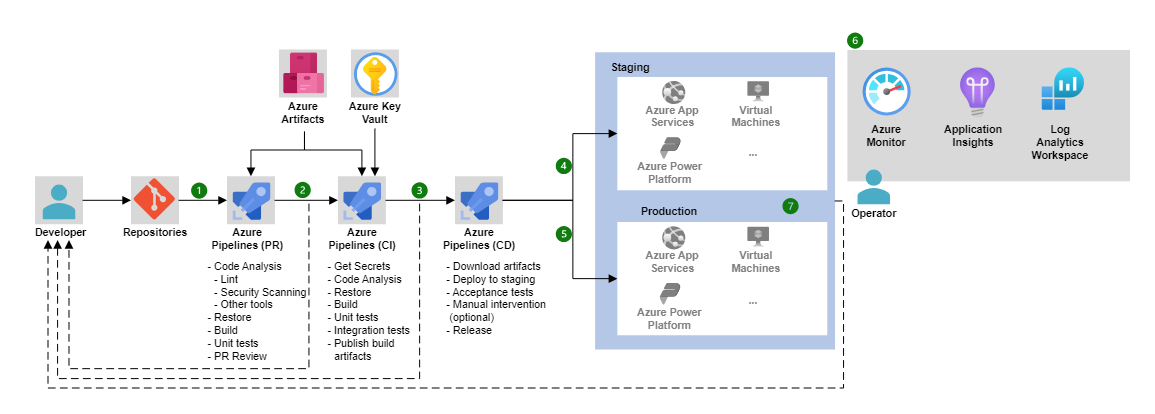
 controller-manager is responsible for making sure that the shared state of the cluster is operating as expected.

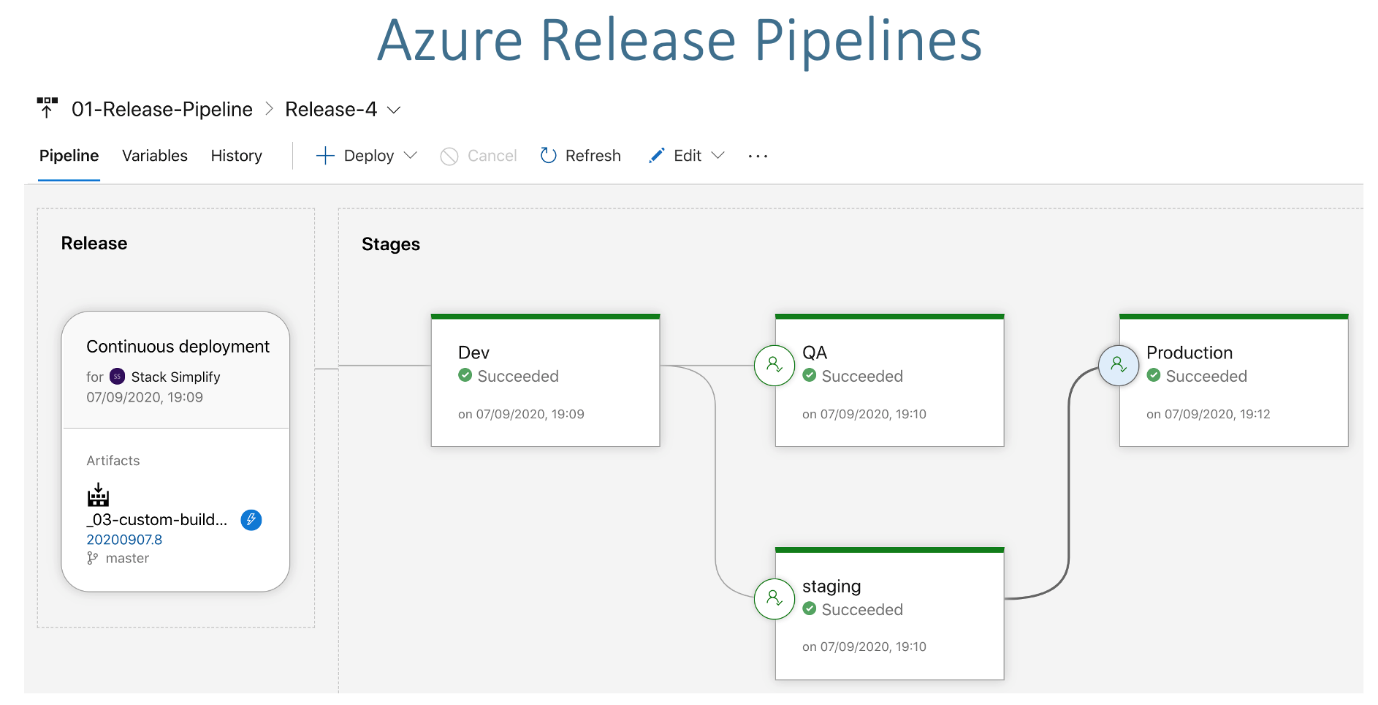
Kubelet tracks the state of a pod to ensure that all the containers are running. It provides a heartbeat message every few seconds to the control plane. If a replication controller does not receive that message, the node is marked as unhealthy.

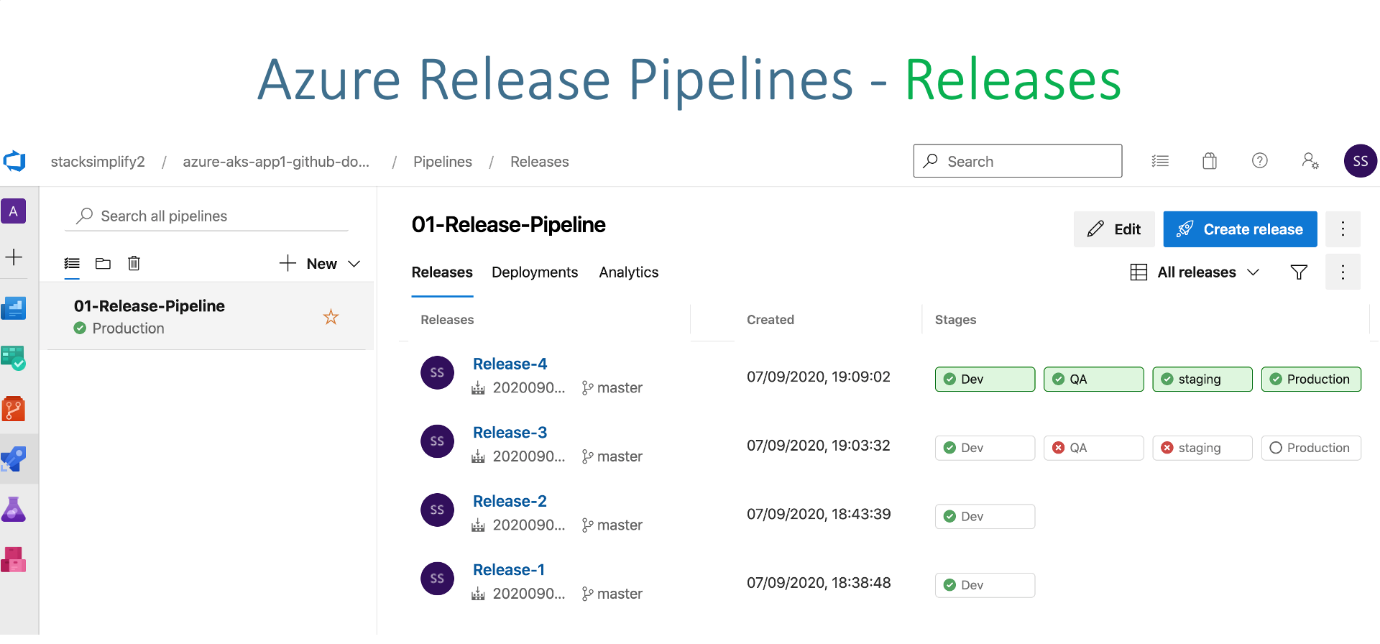




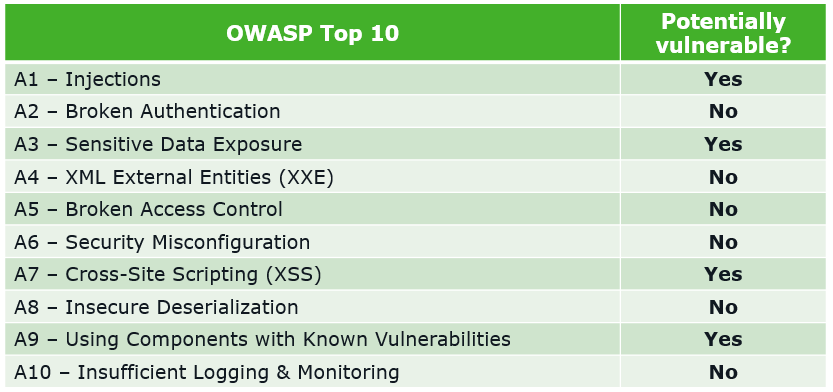








OWASP



Kube proxy routes traffic coming into a node from the service. It forwards requests for work to the correct containers.

 distributed key-value store that Kubernetes uses to share information about the overall state of a cluster.

**12. Maintenance and Monitoring:** - Explain how the component will be maintained, updated, and monitored in the Azure environment. - Describe any Azure monitoring tools or services that will be used.

**13. Conclusion:** - Summarize the key points of the Low-Level Design and reiterate the component's importance within the overall system.

Remember that the Low-Level Design document should be comprehensive, addressing technical details in depth. However, it's also essential to strike a balance between providing enough information and avoiding unnecessary complexity.

Design pattern:

 Azure DevOps/

**transformation from on-premises to Azure hosted solutions.**

Migrating from on-premises solutions to Azure-hosted solutions involves several steps and considerations to ensure a successful transition. Here's a high-level overview of the transformation process:

1. **Assessment and Planning:**
   * Evaluate your existing on-premises infrastructure, applications, and workloads to identify what can be migrated to Azure.
   * Determine the goals of your migration, whether it's cost savings, scalability, improved performance, or other factors.
   * Create a detailed migration plan that outlines the sequence of migration, dependencies, and potential challenges.
2. **Azure Resource Selection:**
   * Choose the appropriate Azure services and resources that align with your on-premises workloads. For example, virtual machines, containers, serverless functions, databases, etc.
   * Consider factors such as compute, storage, networking, and security requirements.
3. **Data Migration:**
   * Plan the migration of your data to Azure. This might involve using Azure Data Migration Services, Azure Database Migration Service, or other migration tools depending on your data sources.
   * Ensure data integrity and consistency during the migration process.
4. **Application Migration:**
   * Depending on your applications, you might need to rehost, refactor, rearchitect, or rebuild them to fit the Azure environment.
   * Lift-and-shift (rehosting) involves moving applications with minimal changes to Azure virtual machines.
   * Refactoring and rearchitecting involve adapting applications to leverage Azure-native services for better scalability and performance.
5. **Networking and Security:**
   * Configure Azure networking to replicate your on-premises network architecture. This might involve creating virtual networks, subnets, VPNs, and ExpressRoute connections.
   * Implement security measures, such as Azure Active Directory for identity and access management, and configure firewall rules to secure your resources.
6. **Testing and Validation:**
   * Conduct thorough testing of the migrated applications and workloads in the Azure environment.
   * Perform functional testing, performance testing, and user acceptance testing to ensure everything works as expected.
7. **Data Synchronization and Cutover:**
   * Plan the final data synchronization and cutover to minimize downtime.
   * Redirect users and traffic from on-premises to the Azure-hosted solutions.
8. **Monitoring and Optimization:**
   * Set up monitoring and alerting using tools like Azure Monitor and Azure Application Insights to ensure the performance and availability of your solutions.
   * Continuously monitor and optimize your Azure resources based on usage patterns and performance data.
9. **Training and Documentation:**
   * Provide training to your team on managing and operating Azure-hosted solutions.
   * Create documentation that outlines the new Azure environment, architecture, and operational procedures.
10. **Post-Migration Review:**
    * After migration, conduct a review to assess the success of the migration and identify areas for improvement.

Remember that the migration process can vary significantly based on your specific applications, workloads, and requirements. It's crucial to engage with Azure experts and utilize Azure's migration tools and resources to ensure a smooth and successful transition.

Microservices architecture is a software design approach where an application is divided into small, independently deployable services that work together to provide the overall functionality of the application. When designing microservices, several design patterns and best practices can help ensure that your architecture is scalable, maintainable, and resilient. Here are some important microservices design patterns:

1. **Service Decomposition**:
   * Break down your monolithic application into smaller services based on well-defined boundaries, typically around business capabilities.
   * Each microservice should have a single responsibility, and services should communicate through APIs (typically RESTful HTTP or gRPC).
2. **API Gateway**:
   * Implement an API gateway to provide a single entry point for client applications.
   * The API gateway can handle tasks like authentication, load balancing, caching, and routing requests to the appropriate microservices.
3. **Service Registry and Discovery**:
   * Use a service registry (e.g., Netflix Eureka, HashiCorp Consul) to register and discover microservices dynamically.
   * This allows services to locate and communicate with each other without hardcoding IP addresses and ports.
4. **Load Balancing**:
   * Employ load balancers to distribute incoming requests evenly across multiple instances of a microservice to ensure scalability and high availability.
5. **Circuit Breaker**:
   * Implement a circuit breaker pattern (e.g., Netflix Hystrix) to prevent cascading failures.
   * It monitors for failures and can temporarily stop sending requests to a failing service, allowing it to recover.
6. **Event Sourcing**:
   * Store the state of a system as a sequence of immutable events.
   * Each microservice can subscribe to relevant events and update its own state accordingly.
   * Event-driven communication between microservices enables loose coupling and scalability.
7. **Saga Pattern**:
   * Manage distributed transactions across multiple microservices using the saga pattern.
   * Break down a complex transaction into a series of smaller, isolated steps with compensating actions in case of failures.
8. **Command-Query Responsibility Segregation (CQRS)**:
   * Separate the read and write operations into different models.
   * Use CQRS to optimize the read side for querying and the write side for updating data.
9. **Database Per Service**:
   * Each microservice should have its own database (or schema) that is private to that service.
   * This isolation helps with independent development and scalability but may require data synchronization mechanisms when needed.
10. **Containerization and Orchestration**:
    * Use containers (e.g., Docker) to package microservices and an orchestration platform (e.g., Kubernetes) to manage container deployments and scaling.
11. **Centralized Logging and Monitoring**:
    * Implement centralized logging and monitoring (e.g., ELK Stack, Prometheus, Grafana) to gain visibility into the health and performance of your microservices.
12. **Resilience Testing**:
    * Regularly perform chaos engineering and resilience testing to identify weaknesses in your microservices architecture and improve fault tolerance.
13. **Versioning and API Contracts**:
    * Establish clear versioning strategies and API contracts to ensure backward compatibility when evolving microservices.
14. **Security**:
    * Implement security measures like authentication, authorization, and encryption at both the network and application levels to protect your microservices.
15. **Cross-Cutting Concerns**:
    * Handle cross-cutting concerns like authentication, authorization, and logging consistently across all microservices using libraries or infrastructure services.

Microservices architecture is a design approach that structures an application as a collection of loosely coupled and independently deployable services. When designing microservices, various design patterns can help address common challenges and ensure that your system is scalable, maintainable, and resilient. Here are some essential microservices design patterns:

1. **Service Registry and Discovery**:
   * **Pattern**: Service Registry
   * **Description**: Maintain a central service registry where microservices can register themselves when they start up. Clients use service discovery to find and communicate with these services dynamically.
   * **Use Case**: Ensures that services can locate and communicate with each other in a dynamic and distributed environment.
2. **API Gateway**:
   * **Pattern**: API Gateway
   * **Description**: Implement a single entry point for clients to interact with your microservices. The API Gateway handles routing, authentication, load balancing, and sometimes even request/response transformation.
   * **Use Case**: Simplifies client interactions and provides a unified API for consumers while distributing requests to the appropriate microservices.
3. **Circuit Breaker**:
   * **Pattern**: Circuit Breaker
   * **Description**: Detect and prevent repeated failures when calling a service by temporarily "opening" the circuit. If the service is unhealthy, requests are not forwarded until it recovers.
   * **Use Case**: Enhances system resilience by preventing cascading failures.
4. **Bulkhead**:
   * **Pattern**: Bulkhead
   * **Description**: Isolate different parts of the system to prevent failures in one area from affecting others. It involves setting resource limits and boundaries for each microservice or component.
   * **Use Case**: Prevents resource exhaustion in one service from affecting the performance of others.
5. **Saga Pattern**:
   * **Pattern**: Saga
   * **Description**: Handles long-running distributed transactions by breaking them into a series of smaller, manageable steps (sagas). Each step is a separate microservice operation, and compensation actions can be executed if a step fails.
   * **Use Case**: Ensures data consistency and integrity in a distributed system.
6. **Event Sourcing**:
   * **Pattern**: Event Sourcing
   * **Description**: Stores the state of a system as a series of immutable events. This enables auditing, versioning, and rebuilding the system's state at any point in time.
   * **Use Case**: Useful for maintaining a complete audit trail and for complex event-driven applications.
7. **Command Query Responsibility Segregation (CQRS)**:
   * **Pattern**: CQRS
   * **Description**: Separates the command (write) and query (read) responsibilities into separate models. This allows for optimized data storage and retrieval.
   * **Use Case**: Optimizes performance for read-heavy and write-heavy operations in a microservices system.
8. **Data Sharding**:
   * **Pattern**: Data Sharding
   * **Description**: Distributes data across multiple databases or instances to improve scalability and reduce contention.
   * **Use Case**: Useful when dealing with large datasets and high write loads.
9. **Retry and Timeout Strategies**:
   * **Pattern**: Retry and Timeout
   * **Description**: Implement strategies for handling network or service failures, including exponential backoff and timeouts, to ensure robust communication between microservices.
   * **Use Case**: Enhances system reliability in unreliable network environments.
10. **Immutable Deployments**:
    * **Pattern**: Immutable Deployments
    * **Description**: Deploy new versions of microservices by creating entirely new instances instead of modifying existing ones. This reduces deployment risk and simplifies rollback.
    * **Use Case**: Ensures consistent and reliable deployments.

These microservices design patterns are just a starting point, and their selection depends on your specific project requirements and constraints. Effective microservices design requires a deep understanding of your application's domain and careful consideration of how each pattern can help address challenges and improve the overall architecture.

Was this response better or worse?

# migrate jenkins pipelines to azure devops

trigger :

 - main

pool:

  vmImage: 'windows-latest'

# parameters are expected from the jenkins pipeline

variables:

  solution: '\*\*/\*.sln'

  buildPlatform: 'Any CPU'

  buildConfiguration: 'Release'

  MSBuildVerbosityLevel: 'minimal'

  Branch: 'task/nov-2021-maint'

  MSDeployMode: 'Y'

  RootDirectory: '$(Pipeline.Workspace)'

parameters:

  - name: Environment

    type: string

    displayName: Environment

    default: Test

    values:

      - Test

      - QA

      - Stage

      - Prod

  - name: websiteNames

    type: object

    displayName: Websites

    default: ChildTime

    values:

      - ChildTime

      - CreativeKids

      - Everbrook

      - LaPetite

      - Montessori

      - ChildrensCourtyard

steps:

  - ${{ each websiteName in parameters.websiteNames }}:

        - task: CmdLine@2

          displayName: Npm Install

          inputs:

            script: |

              CD LCG.${{websiteName}}.Web

              npm install

        - task: CmdLine@2

          displayName: Gulp build

          inputs:

            script: |

              CD LCG.${{websiteName}}.Web

              node\_modules\.bin\gulp build --prod

        - task: MSBuild@1

          displayName: Build a Visual Studio project or solution using MSBuild

          inputs:

            solution: '\*\*/LCG.${{websiteName}}.Web/\*.csproj'

            msbuildVersion: '14.0'

            msbuildArchitecture: 'x64'

            configuration: Configuration['parameters.Environment']

            msbuildArguments: '-p:PublishDir=$(Build.SourcesDirectory)/LCG.CreativeKids.Web/Bin'

Top of Form