Azure Certification Overview

Responsibilities:

- **Participate in all phases of development:**
  - Requirements gathering
  - Design
  - Development
  - Deployment
  - Security
  - Maintenance
  - Performance tuning
  - Monitoring

Proficiency in Azure:

- **SDK (Software Development Kit):**

```
# Example of using Azure SDK for Python
from azure.identity import DefaultAzureCredential
from azure.mgmt.resource import ResourceManagementClient

credential = DefaultAzureCredential()
subscription_id = "your_subscription_id"
resource_client = ResourceManagementClient(credential, subscription_id)
```

- **Data Storage Options:**
  - Azure Blob Storage
  - Azure Table Storage
  - Azure Queue Storage
  - Azure Files
- **Data Connections:**
  - SQL Database
  - Cosmos DB
  - Data Factory
- **APIs:**

```
// Example of calling Azure API using C#
using Microsoft.Azure.Management.ResourceManager.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent.Core;

var azure =
```

```
Azure.Authenticate(credentials).WithSubscription("your_subscription_id");
var resourceGroup = azure.ResourceGroups.GetByName("myResourceGroup");
```

- **App Authentication and Authorization:**
  - Azure Active Directory (Azure AD)
  - Managed Identities
- **Compute and Container Deployment:**
  - Virtual Machines
  - App Services
  - Azure Kubernetes Service (AKS)
  - Azure Container Instances
- **Debugging:**
  - Application Insights
  - Log Analytics

Partnership With:

- Cloud Solution Architects
- Database Administrators (DBAs)
- DevOps Engineers
- Infrastructure Administrators
- Other Stakeholders

Required Experience and Skills:

- **At least two years of programming experience.**
- **Proficient in Azure SDKs:**
  - Familiarity with languages like Python, C#, Java, etc., in context with Azure.
- **Tools Proficiency:**

```
# Azure CLI command example
az login
az group create --name MyResourceGroup --location "East US"
```

Skills Measured:

1. **Develop Azure Compute Solutions**
   - Implement virtual machines, scale sets, app services, containers.
2. **Develop for Azure Storage**

    - Use Azure Blob, Queue, Table, and File storage.

3. **Implement Azure Security**
    - Secure resources with Azure AD, manage keys, secrets, and manage security policies.

4. **Monitor, Troubleshoot, and Optimize Azure Solutions**
    - Use Azure Monitor, Application Insights for performance monitoring.

5. **Connect to and Consume Azure Services and Third-Party Services**
    - Integrate with other Azure services, external APIs, and services.

These notes should give you a solid foundation for understanding what is expected from you in the Azure certification exam. Make sure to practice with the tools and SDKs, as practical experience will be crucial.

AZ-204: Implement Azure App Service Web Apps

Overview of Azure App Service:

- **Functions of Azure App Service:**
    - Hosting web applications, REST APIs, and mobile back ends.
    - Supports multiple languages like .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python.

Key Operations:

- **Creating and Updating an App:**

```
# Create a new web app
az webapp up --sku F1 --name <app-name> --resource-group <resource-group-
name>

# Update an existing web app
az webapp config appsettings set --resource-group <resource-group-name> --
name <app-name> --settings "key=value"
```

- **Authentication and Authorization:**
    - Configure Authentication/Authorization in App Service to secure your app.
    - Use Azure AD, Google, Facebook, Twitter, or Microsoft Account for authentication.

- **App Settings:**
    - Manage configuration settings outside of your deployment package.

```
# Add app settings
az webapp config appsettings set --resource-group <resource-group-name> --
name <app-name> --settings "Setting1=Value1" "Setting2=Value2"
```

- **Scaling Apps:**
    - Scale-up (change the pricing tier for more resources) or scale-out (add more instances).

```
# Scale up to a different tier
az appservice plan update --resource-group <resource-group-name> --name
<app-service-plan-name> --sku P1V2

# Scale out (adjust instance count)
az appservice plan update --resource-group <resource-group-name> --name
<app-service-plan-name> --number-of-workers 5
```

- **Deployment Slots:**
    - Slots allow you to deploy different versions of your app for testing before making them live.

```
# Create a deployment slot
az webapp deployment slot create --name <app-name> --resource-group
<resource-group-name> --slot staging

# Swap slots
az webapp deployment slot swap --name <app-name> --resource-group <resource-
group-name> --slot staging --target-slot production
```

Prerequisites:

- **Experience:**
    - At least one year of experience developing scalable solutions through all phases of software development.
- **Azure Knowledge:**
    - Basic understanding of Azure, cloud concepts, Azure services, and operations via the Azure portal.
- **Recommended Training:**
    - If new to Azure, complete the **AZ-900: Azure Fundamentals** course to get foundational knowledge.

Additional Tips:

- **Practice with Azure App Service:**
    - Use the Azure portal, Azure CLI, and ARM templates to get hands-on experience.
    - Experiment with different authentication providers and understand how they integrate with your applications.

- **Understand Deployment:**
    - Learn how continuous deployment works with Azure DevOps, GitHub actions, or other CI/CD tools.
- **Monitor and Troubleshoot:**
    - Get familiar with Azure Monitor and Application Insights for monitoring your web apps' performance and diagnosing issues.

By mastering these concepts and practices, you'll be well-prepared for the AZ-204 exam section on implementing Azure App Service web apps.

Introduction to Azure App Service Evaluation

Azure App Service Key Components and Value:

- **Platform as a Service (PaaS):** Azure App Service provides a managed environment for hosting web applications, REST APIs, and mobile backends.
- **Supported Languages:** .NET, .NET Core, Java, Ruby, Node.js, PHP, Python.
- **Built-in Features:**
    - Automatic scaling, load balancing, and high availability.
    - **Continuous Deployment:** Integration with source control systems like GitHub, Azure DevOps.
- **Value Proposition:**
    - **Simplicity:** Easy to deploy and manage web applications without worrying about infrastructure.
    - **Cost-Effective:** Pay only for what you use with various pricing tiers.
    - **Security:** SSL/TLS encryption, authentication/authorization capabilities.

Authentication and Authorization in Azure App Service:

- **Authentication/Authorization Middleware:**
    - Azure App Service can handle user authentication for your app.
    - Supports authentication providers like Azure AD, Google, Twitter, Microsoft Account.
- **Implementation:**
    - **App Settings:**

```
# Enable Authentication/Authorization for an app
az webapp auth update --resource-group <resource-group-name> --name
<app-name> --enabled true --action AllowAnonymous
```

Controlling Traffic to Your Web App:

- **Inbound Traffic Control:**
  - **IP Restrictions:** Allow or block traffic from specific IP addresses.

```
# Add an IP restriction
az webapp config access-restriction add --resource-group <resource-
group-name> --name <app-name> --rule-name "AllowSpecificIP" --action
Allow --ip-address <ip-address>
```

  - **Service Endpoints:** Secure access to Azure services from your web app.
- **Outbound Traffic Control:**
  - **Hybrid Connections:** Connect to on-premises systems securely.
  - **Virtual Network Integration:** Route outbound traffic through a VNet.

Deploying an App to App Service Using Azure CLI:

- **Deployment Steps:**
  1. **Prepare Your App:**
     - Ensure your app is compatible with App Service (check runtime support).
  2. **Create App Service Plan and Web App:**

```
# Create an App Service Plan
az appservice plan create --name myAppServicePlan --resource-group
<resource-group-name> --sku FREE

# Create a new web app
az webapp create --resource-group <resource-group-name> --plan
myAppServicePlan --name <app-name>
```

  3. **Deploy using Azure CLI:**

```
# Deploy from a local folder
az webapp up --name <app-name> --resource-group <resource-group-name> -
-sku F1
```

     - **Git Deployment:**

```
# Configure deployment from a Git repository
az webapp deployment source config --name <app-name> --resource-
group <resource-group-name> --repo-url <git-repo-url> --branch
master --manual-integration
```

Learning Objectives Recap:

- **Describe:** Understand and articulate the features and benefits of Azure App Service.
- **Explain:** Gain insights into how authentication works within App Service.
- **Identify:** Know the tools and settings for managing network traffic.
- **Deploy:** Practice deploying an app using Azure CLI for real-world applicability.

These notes will help you present a thorough evaluation of Azure App Service to your company, showcasing its capabilities, security features, and deployment processes.

Azure App Service Overview

- **Purpose:**
    - HTTP-based service for hosting web applications, REST APIs, and mobile backends.
    - Supports development in various programming languages and frameworks.
- **OS Support:**
    - Runs on Windows and Linux environments.

Key Features:

- **Auto-Scale Support:**
    - **Scaling Up/Down:** Adjust resources like cores and RAM on the host machine.
    - **Scaling Out/In:** Increase/decrease the number of instances running your app.
- **Container Support:**
    - **Deployment:** Deploy containerized apps on both Windows and Linux.
    - **Sources:** Use containers from Azure Container Registry or Docker Hub.
    - **Multi-container Apps:** Supports complex applications with multiple containers.
    - **Orchestration:** Use Docker Compose for managing container instances.
- **Continuous Integration/Deployment (CI/CD):**
    - **Integration Options:** Azure DevOps, GitHub, Bitbucket, FTP, or local Git.
    - **Process:** Auto-syncs code changes to the web app.
    - **Container CI/CD:** Support for containerized deployments.
- **Deployment Slots:**
    - **Functionality:** Use separate slots for staging deployments (Standard tier or above).
    - **Swapping:** Easily swap content and configurations between slots, including production.

App Service on Linux:

- **Native Support:** Host apps directly on Linux for languages like Node.js, Java, PHP, Python, .NET, Ruby.
- **Custom Containers:** Option to deploy your custom Docker containers.

- **Runtime Updates:** Languages and frameworks are frequently updated.
- **Check Supported Runtimes:**

```
az webapp list-runtimes --os-type linux
```

Limitations of App Service on Linux:

- **Pricing:** Not available in the Shared pricing tier.
- **Portal Features:** Only Linux-compatible features are displayed in the Azure portal.
- **Storage:**
  - Built-in images use Azure Storage for content, which might have higher and variable latency.
  - Consider custom containers for apps needing high read-only access to content files for better performance.

Conclusion:

Azure App Service offers robust features for web app hosting, from scalability to containerization and seamless CI/CD. While there are some limitations, particularly with Linux-based services, the platform provides versatile options for developers to deploy applications efficiently. Remember to consider these limitations when choosing your deployment strategy for Linux-based applications to ensure you select the appropriate tier and deployment method for your needs.

Azure App Service Plans

Overview:

- **Purpose:** An App Service plan defines compute resources for web apps to run on.
- **Shared Resources:** Multiple apps can run on the same set of compute resources.

Components of an App Service Plan:

- **Operating System:** Windows or Linux.
- **Region:** Geographical location (e.g., West US, East US).
- **VM Instances:** Number of virtual machine instances.
- **VM Size:** Small, Medium, Large, etc.
- **Pricing Tier:** Determines features and cost:
  - **Shared Compute:** Free, Shared
    - Apps run on shared Azure VMs with other customers' apps.
    - CPU quotas are allocated; cannot scale out.
  - **Dedicated Compute:** Basic, Standard, Premium, PremiumV2, PremiumV3
    - Apps run on dedicated VMs.
    - Shared among apps in the same plan.
  - **Isolated:** Isolated, IsolatedV2

        - Provides dedicated VMs on isolated Azure Virtual Networks.
        - Maximum scale-out capabilities.

Note: Free and Shared tiers are for development and testing, not production.

App Execution and Scaling:

- **Free and Shared Tiers:**
    - Apps receive CPU minutes on shared instances.
    - No scale-out capability.
- **Other Tiers (Dedicated and Isolated):**
    - **Execution:** Apps run on all configured VM instances in the plan.
    - **Scaling:**
        - All apps within the plan scale together.
        - Multiple deployment slots share VM instances.
        - Diagnostics, backups, WebJobs also consume resources from these instances.

Scaling Up or Down:

- **Flexibility:** You can change the pricing tier at any time, which scales the plan up or down.

App Performance Considerations:

- **Isolating Apps:**
    - Move resource-intensive apps to their own App Service plan.
    - Allows independent scaling and resource allocation.
- **Cost Saving:**
    - Multiple apps in one plan can reduce costs if resource use is complementary.
- **When to Isolate:**
    - App is resource-heavy.
    - Independent scaling is required.
    - App needs resources in a different geographical region.

Implementation Steps for App Service Plans:

- **Creating an App Service Plan:**

```
az appservice plan create --name myAppServicePlan --resource-group
<resource-group-name> --sku FREE
```

- **Scaling an App Service Plan:**

```
# Scale up to Standard tier
az appservice plan update --resource-group <resource-group-name> --name
<app-service-plan-name> --sku S1

# Scale out (increase instance count)
az appservice plan update --resource-group <resource-group-name> --name
<app-service-plan-name> --number-of-workers 5
```

- **Moving an App to a New Plan:**
    - Create a new plan with desired specs.
    - Use the Azure portal or CLI to move the app to this new plan.

Conclusion:

App Service plans are crucial for defining the compute environment for your Azure web apps. Understanding when and how to scale or isolate your applications within these plans is key to optimizing performance and cost-efficiency. Remember to consider the resource demands of your apps and their growth when planning your Azure App Service strategy.

Deployment Options for Azure App Service

Automated Deployment (Continuous Deployment):

- **Benefits:** Streamlines the release of new features and bug fixes with minimal impact on users.
- **Supported Sources:**
    - **Azure DevOps Services:**
        - Steps: Push code, build in cloud, run tests, create release, deploy to Azure Web App.
    - **GitHub:**
        - Connect repository for automatic deployment on production branch updates.
    - **Bitbucket:**
        - Similar setup to GitHub for automated deployments.

Manual Deployment:

- **Git Deployment:**
    - **Process:** Add Azure App Service as a remote Git repository, push code to deploy.

```
# Add Azure as a remote repository (example URL)
git remote add azure https://<deployment-username>@<app-
name>.scm.azurewebsites.net/<app-name>.git
```

```
# Push to Azure
git push azure master
```

- **Azure CLI:**
  - **Command:** `az webapp up` can deploy or create and deploy an app.

```
az webapp up --name <app-name> --resource-group <resource-group-name> -
-sku F1
```

- **Zip Deploy:**
  - **Method:** Push a zip file of your app to App Service via HTTP.

```
# Example using curl
curl -X POST -u <username>:<password> --data-binary @<filename>.zip
<app-url>/api/zipdeploy
```

- **FTP/S Deployment:**
  - Traditional file transfer method for code deployment.

Using Deployment Slots:

- **Purpose:** Allows staging deployments to reduce risk and downtime.
  - **Create a Slot:**

```
az webapp deployment slot create --name <app-name> --resource-group
<resource-group-name> --slot staging
```

  - **Swap Slots:**

```
az webapp deployment slot swap --name <app-name> --resource-group
<resource-group-name> --slot staging --target-slot production
```

- **Best Practices:**
  - Use slots for staging before production deployment.
  - Test, QA, and staging branches should deploy to different staging slots.

Continuous Deployment for Containers:

- **Workflow for Custom Containers:**

1. **Build and Tag the Image:**
    - Tag with git commit ID or timestamp for traceability.

```
docker build -t myappregistry.azurecr.io/myapp:v1.0-${GIT_COMMIT}
.
```

2. **Push the Image:**
    - Push to Azure Container Registry or similar.

```
az acr login --name myappregistry
docker push myappregistry.azurecr.io/myapp:v1.0-${GIT_COMMIT}
```

3. **Update Deployment Slot:**
    - Update the image tag for the staging slot to trigger a restart and image pull.

```
az webapp config container set --resource-group <resource-group-
name> --name <app-name> --slot staging --docker-registry-server-
url <container-registry-url> --docker-custom-image-name
myappregistry.azurecr.io/myapp:v1.0-${GIT_COMMIT}
```

Conclusion:

- **Automation:** Key for reducing manual errors and improving efficiency.
- **Slots:** Essential for zero-downtime deployments and testing.
- **Containers:** Require additional considerations for tagging and updating but offer flexibility and consistency across environments.

Exploring Authentication and Authorization in Azure App Service

Overview:

- **Purpose:** Azure App Service offers built-in authentication and authorization to secure your applications effortlessly.

Why Use Built-in Authentication:

- **Time and Effort Savings:**
    - Minimizes or eliminates the need for custom authentication code.
    - Focuses your development efforts on business logic rather than security infrastructure.
- **Flexibility:**

    - While you can opt for custom security implementations, the built-in feature offers immediate usability.

- **Native Integration:**
    - Authentication is integrated into the platform, requiring no specific language or SDK.

Key Features of App Service Authentication:

- **Federated Identity Providers:**
    - **Support for Multiple Providers:**
        - Microsoft Entra ID (formerly Azure AD)
        - Facebook
        - Google
        - X (formerly Twitter)

    This allows users to sign in with their existing accounts, enhancing user experience and simplifying user management.

- **No Code Requirement:**
    - Authentication can be configured through the Azure portal or Azure CLI without altering your application code.

Implementation Steps:

1. **Enable Authentication in App Service:**
    - Go to the Azure portal, navigate to your App Service, and under **Settings**, click on **Authentication / Authorization**.
    - Toggle **App Service Authentication** to **On**.

    ```
    # Enable authentication for an app using Azure CLI
    az webapp auth update --resource-group <resource-group-name> --name <app-name> --enabled true --action AllowAnonymous
    ```

2. **Configure Authentication Providers:**
    - Select the identity providers you want to use. For example:

    ```
    # Configure Microsoft Entra ID as an auth provider
    az webapp auth microsoft update --resource-group <resource-group-name> --name <app-name> \
      --client-id <client-id> --client-secret <client-secret> --issuer-url <issuer-url>
    ```

3. **Authorization Settings:**

    - Choose how your app handles unauthenticated requests:
        - **AllowAnonymous:** Allows anonymous access but still provides authentication tokens if available.
        - **Log in with :** Redirects unauthenticated users to a login page for the specified provider.

4. **Additional Configurations:**

    - **Token Store:** Can be enabled to cache tokens for your app, simplifying subsequent authentication checks.

Considerations:

- **Custom Authentication:** If App Service's authentication doesn't meet your needs, you can still implement custom solutions.
    - This might be necessary for specialized authorization logic, integration with unsupported providers, or if you need more control over the security workflow.
- **Security Best Practices:**
    - Even with built-in authentication, ensure to follow security best practices like:
        - Use HTTPS to encrypt data in transit.
        - Implement proper authorization logic within your app to complement authentication.
- **User Identity Information:**
    - When a user is authenticated, their identity claims are available through the request headers for your application to use.

Conclusion:

Azure App Service’s built-in authentication and authorization features provide a straightforward way to secure your applications, supporting multiple identity providers out of the box. This feature allows developers to maintain application security without deep dives into security implementation details, freeing up time to enhance the application's core functionalities.

Exploring Authentication and Authorization in Azure App Service

Identity Providers

Azure App Service supports federated identity where a third-party provider manages user identities and the authentication process. Here are the default providers:

- **Microsoft Identity Platform:**
    - **Endpoint:** `/.auth/login/aad`
    - **Documentation:**[App Service Microsoft identity platform login](link to doc)
- **Facebook:**
    - **Endpoint:** `/.auth/login/facebook`

    - **Documentation:**[App Service Facebook login](link to doc)

- **Google:**
    - **Endpoint:** `/.auth/login/google`
    - **Documentation:**[App Service Google login](link to doc)

- **X (formerly Twitter):**
    - **Endpoint:** `/.auth/login/twitter`
    - **Documentation:**[App Service X login](link to doc)

- **Any OpenID Connect Provider:**
    - **Endpoint:** `/.auth/login/<providerName>`
    - **Documentation:**[App Service OpenID Connect login](link to doc)

- **GitHub:**
    - **Endpoint:** `/.auth/login/github`
    - **Documentation:**[App Service GitHub login](link to doc)

Each provider's endpoint is used for user authentication and token validation.

How Authentication and Authorization Works

- **Module Execution:**
    - The authentication module operates in the same sandbox as your application for Windows, but in a separate container for Linux and custom containers.

- **Request Handling:**
    - **Authentication:** Handles user and client authentication with the specified identity provider.
    - **Token Management:** Validates, stores, and refreshes OAuth tokens.
    - **Session Management:** Manages the authenticated session state.
    - **Header Injection:** Injects identity information into HTTP headers.

- **Configuration:**
    - Can be set up via Azure Resource Manager settings or a configuration file, without requiring SDKs or code changes.

- **Platform Independence:**
    - This system is language-agnostic, meaning you don't need to modify your application's code to utilize the authentication features.

Note for Linux and Containers:

- Authentication runs in a separate, isolated container, which means there's no direct in-process integration with language frameworks.

Example Configuration via Azure CLI:

To enable authentication for an app with Microsoft Entra ID:

```
az webapp auth update --resource-group <resource-group-name> --name <app-name> \
  --enabled true --action AllowAnonymous \
  --aad-client-id <client-id> --aad-client-secret <client-secret> \
  --aad-issuer-url <issuer-url>
```

Key Points:

- **Multiple Providers:** You can configure multiple sign-in options for users.
- **No Code Changes:** Authentication can be enabled without altering your application's codebase.
- **Security Features:**
  - Token management ensures secure and efficient handling of OAuth tokens.
  - Session management keeps users signed in across requests.

Conclusion:

Azure App Service's authentication and authorization capabilities simplify securing your applications by offloading the complexities of identity management to trusted providers. This integration allows for a seamless setup where your application remains focused on its core functionality while the platform handles user identity and security.

Authentication Flow in Azure App Service

Overview:

- **Provider SDK Usage:** The flow varies based on whether the application uses the identity provider's SDK or relies on App Service for authentication.

Authentication Flow Steps:

- **Without Provider SDK (Server-Directed Flow):**
  1. **Sign User In:**
     - Redirects to `/.auth/login/<provider>`.
  2. **Post-Authentication:**
     - Provider redirects back to `/.auth/login/<provider>/callback`.
  3. **Establish Authenticated Session:**
     - App Service adds an authenticated cookie to the response.
  4. **Serve Authenticated Content:**
     - Client includes the authentication cookie in subsequent requests (handled by the browser).

- **With Provider SDK (Client-Directed Flow):**

    1. **Sign User In:**
        - Client code uses the provider's SDK to authenticate and receive a token.
        - See provider's documentation for specifics.
    2. **Post-Authentication:**
        - Client code sends the token to `/.auth/login/<provider>` for validation.
    3. **Establish Authenticated Session:**
        - App Service returns its authentication token.
    4. **Serve Authenticated Content:**
        - Client presents the token in the `X-ZUMO-AUTH` header (handled by Mobile Apps client SDKs).

Note: For browser clients, App Service can automatically redirect unauthenticated users to `/.auth/login/<provider>`.

Authorization Behavior Configuration:

- **Allow Unauthenticated Requests:**
    - Authentication is deferred to your application code.
    - Authenticated requests include authentication info in headers.
    - Useful for presenting multiple sign-in options or handling anonymous requests.
- **Require Authentication:**
    - Rejects unauthenticated traffic:
        - **Redirect:** Sends the user to `/.auth/login/<provider>`.
        - **HTTP 401 Unauthorized:** For native mobile apps or when configured to return unauthorized.
        - **HTTP 403 Forbidden:** Can be set as the rejection response.

Caution:

- Requiring authentication for all calls might not be suitable for apps with public pages, like single-page applications.

Configuration via Azure Portal:

- **Authentication Settings:**
    - Navigate to your App Service in the Azure portal.
    - Go to **Authentication / Authorization** under **Settings**.
    - **Toggle Authentication:** On.
    - **Choose Action:** Select how to handle unauthenticated requests (Allow or Require).

Example Configuration via Azure CLI:

To configure App Service to require authentication with redirection:

```
az webapp auth update --resource-group <resource-group-name> --name <app-name> \
  --enabled true --action LoginWithMicrosoftAccount
```

Conclusion:

Azure App Service provides a flexible authentication framework that can operate with or without direct integration with provider SDKs. The choice between server-directed and client-directed flows depends on your application's nature and interaction with users. The authorization settings allow you to control access to your application, balancing between security and user experience.

Token Store in Azure App Service

Overview:

- **Function:** Azure App Service includes a built-in token store for managing tokens associated with authenticated users.
- **Activation:**
    - Automatically enabled when you configure authentication with any identity provider.

Token Store Benefits:

- **Token Management:**
    - Stores tokens issued by the identity providers.
    - Handles token refresh automatically, reducing the need for users to re-authenticate.
- **Session Management:**
    - Helps maintain authenticated sessions across requests without repeated authentication.

Logging and Tracing in App Service

Purpose:

- **Authentication Insights:** Provides detailed logs for troubleshooting authentication issues.

Implementation:

- **Enable Application Logging:**
    - Go to your App Service in the Azure portal.
    - Under **Monitoring**, select **App Service logs**.
    - Enable **Application Logging (Filesystem)** or **Application Logging (Blob)**.

- **Viewing Logs:**
  - **Log Stream:** In the Azure portal, you can use the Log stream to view logs in real-time.
  - **Log Files:** Logs are stored in the `LogFiles` directory of your App Service or in the specified Blob storage.

Log Content:

- **Authentication and Authorization Traces:**
  - Logs include information about authentication requests, token validation, and any errors encountered.
  - Useful for diagnosing issues like unexpected authentication failures.

Example Configuration via Azure CLI:

To enable application logging:

```
az webapp log config --resource-group <resource-group-name> --name <app-name> \
  --application-logging true --level verbose --web-server-logging filesystem
```

And to view logs:

```
az webapp log tail --resource-group <resource-group-name> --name <app-name>
```

Notes:

- **Security:** Ensure that log files containing sensitive authentication information are secured and not exposed publicly.
- **Best Practices for Logging:**
  - Regularly review logs for security events.
  - Use logs for performance tuning and debugging beyond just authentication issues.
  - Consider integrating with Azure Monitor for advanced log analysis and alerts.

Conclusion:

The token store simplifies the management of OAuth tokens, enhancing user experience by seamlessly handling session maintenance. Meanwhile, the logging and tracing capabilities ensure that you can troubleshoot authentication problems efficiently, with comprehensive details logged directly in your application's log files.

Azure App Service Networking Features

Overview:

- **Default Accessibility:** Apps in App Service are internet-accessible by default, with outbound traffic limited to internet endpoints.
- **Deployment Types:**
  - **Multi-tenant:** Includes Free, Shared, Basic, Standard, Premium, PremiumV2, PremiumV3 SKUs.
  - **Single-tenant:** App Service Environment (ASE) for Isolated SKU plans within a virtual network.

Multi-tenant Networking Features:

- **Network Architecture:**
  - **Front Ends:** Handle incoming HTTP/HTTPS requests.
  - **Workers:** Host customer workloads.
  - **Network Isolation:** Not directly connectable to your network due to multi-tenant environment.
- **Networking Features:**

  Inbound Traffic Control:

  - **App-assigned Address:**
    - Provides a dedicated IP for SSL needs or a unique inbound address.

    ```
    # Assign a custom domain to an app (which can have a dedicated IP)
    az webapp config hostname add --resource-group <resource-group-name> --
    name <app-name> --hostname <custom-domain>
    ```

  - **Access Restrictions:**
    - Restricts access to your app based on IP addresses.

    ```
    # Add an IP restriction
    az webapp config access-restriction add --resource-group <resource-
    group-name> --name <app-name> --rule-name "AllowSpecificIP" --action
    Allow --ip-address <ip-address>
    ```

  - **Service Endpoints:**
    - Allows secure access to Azure services like Azure Storage from your app.
  - **Private Endpoints:**
    - Connects your app privately to Azure services or on-premises networks without internet exposure.

  Outbound Traffic Control:

  - **Hybrid Connections:**

        - Enables connectivity to on-premises systems securely.

```
# Create a Hybrid Connection endpoint
az relay hybrid-connection create --resource-group <resource-group-
name> --namespace-name <namespace-name> --name <hybrid-connection-name>
--type HybridConnection
```

    - **Gateway-required Virtual Network Integration:**
        - Allows your app to connect to resources in a virtual network via an Azure Application Gateway or a VPN Gateway.
    - **Virtual Network Integration:**
        - Enables your app to access resources within an Azure VNet.

Inbound Use Cases:

- **SSL Certificate Binding:**
    - **Feature:** App-assigned address for IP-based SSL.
- **Dedicated Inbound Address:**
    - **Feature:** App-assigned address to provide a unique inbound IP.
- **IP Whitelisting:**
    - **Feature:** Access restrictions to allow access only from specific IP addresses.

Notes:

- **Feature Compatibility:** Mixing features is possible, but there are limitations due to the nature of the multi-tenant environment.
- **Security:** By using these features, you can significantly enhance the security of your applications by controlling who can access them and how they communicate outward.

Conclusion:

Azure App Service networking features offer a robust set of tools for controlling both inbound and outbound traffic. These features allow developers to tailor network behavior to specific application requirements, ensuring both security and connectivity needs are met without compromising on performance or user experience.

Default Networking Behavior in Azure App Service

Overview:

- **Scale Units:** Azure App Service runs on scale units that serve multiple customers.

- **VM Types:**
    - **Free, Shared, Basic, Standard, Premium:** Share similar worker VM types.
    - **PremiumV2:** Uses a different VM type.
    - **PremiumV3:** Uses yet another VM type.

Outbound Addresses:

- **Worker VMs and Outbound IPs:**
    - Apps share outbound IP addresses based on the VM family they run on.
    - Changing to a different VM family (like moving to PremiumV2 or PremiumV3) changes the set of outbound IPs.
- **Address Sharing:**
    - The outbound addresses are used by all apps on the same worker VM family within an App Service deployment.

Finding Outbound IP Information:

- **Azure Portal:**
    - Navigate to your app in the Azure portal.
    - Go to **Properties** in the left-hand navigation to view current outbound IP addresses.
- **Azure CLI Commands:**

  Current Outbound IP Addresses:

```
az webapp show \
    --resource-group <group_name> \
    --name <app-name> \
    --query outboundIpAddresses \
    --output tsv
```

  Possible Outbound IP Addresses (for all tiers):

```
az webapp show \
    --resource-group <group_name> \
    --name <app-name> \
    --query possibleOutboundIpAddresses \
    --output tsv
```

Notes:

- **Scaling:** When scaling out, all apps in the same App Service plan are replicated across new instances, but they will share the same set of outbound IPs unless you change VM families.

- **Predictability:** Knowing the possible outbound IPs is useful for firewall rules or when integrating with services that require IP whitelisting.
- **Multi-Tenant Implications:** In multi-tenant environments (Free and Shared), your app might share an outbound IP with other customers' apps, which might not be ideal for some scenarios.

Conclusion:

Understanding and managing outbound IP addresses in Azure App Service is crucial for applications that need to communicate externally or integrate with services that whitelist IPs. The ability to scale while maintaining or managing these addresses allows for flexible deployment strategies tailored to security and connectivity needs.

Exercise: Create a Static HTML Web App Using Azure Cloud Shell

Prerequisites:

- **Account Verification:** Use the free Azure sandbox which allows resource creation in specific regions.

Step-by-Step Guide:

1. **Switch to Classic Cloud Shell:**
    - After loading the Cloud Shell, go to **Settings** and select **Go to Classic version**.
2. **Download the Sample App:**
    - Create and navigate to a new directory:

```
mkdir htmlapp
cd htmlapp
```

    - Clone the sample repository:

```
git clone https://github.com/Azure-Samples/html-docs-hello-world.git
```

    - Set up variables for resource group and app name:

```
resourceGroup=$(az group list --query "[].{id:name}" -o tsv)
appName=az204app$RANDOM
```

3. **Create the Web App:**
    - Change to the sample code directory:

```
cd html-docs-hello-world
```

- Deploy the app using the `az webapp up` command:

```
az webapp up -g $resourceGroup -n $appName --html
```

- This command will:
  - Create a resource group if needed.
  - Create an App Service plan.
  - Create a web app.
  - Deploy files from the current directory.
- Note the `app_url` from the output to verify your app in a browser.

4. **Verify Deployment:**
   - Open the app URL in a new browser tab to ensure the site is up and running.
5. **Update and Redeploy the App:**
   - Edit the HTML file:

```
code index.html
```

     - Modify the `<h1>` tag content.
     - Use `Ctrl+S` to save, then `Ctrl+Q` to exit.
   - Redeploy the updated app:

```
az webapp up -g $resourceGroup -n $appName --html
```

     - You can recall this command using the up-arrow key.
   - Refresh the browser tab to see your changes.

Notes:

- **Region:** Ensure to select one of the available regions for your sandbox deployment.
- **Command Utility:** The `az webapp up` command simplifies the deployment process by handling several steps automatically.
- **Environment:** This exercise assumes you're working in a Cloud Shell environment, where you don't need to install the Azure CLI locally.

Conclusion:

This exercise demonstrates how to quickly set up, deploy, and update a static HTML site using Azure App Service and Azure CLI commands in the Cloud Shell. It's an excellent way to get started with Azure web services, showcasing the ease of deployment and the power of Azure's command-line tools for managing applications.

Azure Web App Configuration

Objective: Learn to configure various settings for Azure web apps.

Application Settings

- **Create and Manage Application Settings**: Application settings in Azure can be used to store configuration information:

```
az webapp config appsettings set --resource-group <resource-group-name> --
name <app-name> --settings <key>=<value>
```

SSL/TLS Certificates

- **Install SSL/TLS Certificates**: Secure web traffic by adding SSL/TLS certificates.

```
az webapp config ssl import --resource-group <resource-group-name> --name
<app-name> --cert-file <path-to-cert-file> --key-file <path-to-key-file> --
password <certificate-password>
```

Diagnostic Logging

- **Enable Diagnostic Logging**: Helps in troubleshooting and monitoring:

```
az webapp log config --resource-group <resource-group-name> --name <app-
name> --application-logging filesystem --detailed-error-messages true --
failed-request-tracing true
```

Virtual App to Directory Mappings

- **Create Virtual App to Directory Mappings**: Map a virtual path to a physical directory:

```
az webapp config set --resource-group <resource-group-name> --name <app-
name> --virtual-applications '[{"virtualPath": "/myapp", "physicalPath":
"site\\wwwroot\\myapp"}]'
```

Managing App Features

- **App Features**: Various features can be enabled or disabled via Azure CLI or portal. For example, to enable Always On:

```
az webapp config set --resource-group <resource-group-name> --name <app-
name> --always-on true
```

Key Points:

- **Configuration**: Application settings are crucial for runtime configuration without changing code.
- **Security**: SSL/TLS certificates are vital for secure communications.
- **Monitoring**: Logging is key for debugging and performance monitoring.
- **Flexibility**: Virtual directory mappings offer flexibility in how applications are structured and served.

This script provides a quick reference for managing Azure web app settings, focusing on security, performance, and configuration management. Remember, these are just the basic commands; always check Azure documentation for the latest CLI options and best practices.

Azure App Service Introduction

Key Concept: App Settings

- **Purpose**: App settings in Azure App Service are essentially environment variables that provide a way to configure your application without changing its code.

Learning Objectives:

1. **Create Application Settings**:
    - Application settings can be slot-specific or shared across deployment slots:

```
az webapp config appsettings set --resource-group <resource-group-name>
--name <app-name> --slot <slot-name> --settings <key>=<value>
```

    - For shared settings across all slots:

```
az webapp config appsettings set --resource-group <resource-group-name>
--name <app-name> --settings <key>=<value>
```

2. **SSL/TLS Certificates**:
    - Understanding how to secure your app with SSL/TLS:
        - **Import a certificate**:

```
az webapp config ssl import --resource-group <resource-group-name>
--name <app-name> --cert-file <path-to-cert-file> --key-file
<path-to-key-file> --password <certificate-password>
```

- **Bind certificate to hostname**:

```
az webapp config ssl bind --resource-group <resource-group-name> -
-name <app-name> --certificate-thumbprint <thumbprint> --ssl-type
SNI
```

3. **Enable Diagnostic Logging**:
   - Log configuration for better debugging and monitoring:

```
az webapp log config --resource-group <resource-group-name> --name
<app-name> --application-logging filesystem --detailed-error-messages
true --failed-request-tracing true
```

4. **Virtual App to Directory Mappings**:
   - Useful for structuring applications with different virtual paths mapping to physical directories:

```
az webapp config set --resource-group <resource-group-name> --name
<app-name> --virtual-applications '[{"virtualPath": "/myapp",
"physicalPath": "site\\wwwroot\\myapp"}]'
```

Note: Always check the latest Azure CLI documentation for any updates or changes to command syntax.

These notes summarize the introduction to Azure App Service settings, focusing on application configuration, security, logging, and application structure management. Remember, each of these operations can be performed through the Azure Portal as well, but using the CLI can streamline your workflow, especially in automated environments or scripts.

Key Concepts:

- **Application Settings**: These are environment variables provided to your application code in Azure App Service. For Linux apps and custom containers, these are passed using the `--env` flag.

Accessing Application Settings:

- Navigate to your app's management page:
  - Go to **Environment variables** > **Application settings**.

ASP.NET and ASP.NET Core Usage:

- App settings in App Service override those in `Web.config` or `appsettings.json`.
- Local settings can be kept in `Web.config` or `appsettings.json`, while production secrets should reside in App Service for security.

Security:

- All app settings are encrypted at rest.

Adding and Editing Settings:

- **Single Setting Addition**:
  - Select **+ Add** to add a new setting.
  - If using deployment slots, decide if the setting should be slot-specific or swappable.
- **Bulk Editing/Adding Settings**:
  - Use **Advanced edit** for bulk operations. Here's an example JSON format for app settings:

```
[
  {
    "name": "<key-1>",
    "value": "<value-1>",
    "slotSetting": false
  },
  {
    "name": "<key-2>",
    "value": "<value-2>",
    "slotSetting": false
  }
]
```

Configure Connection Strings:

- For ASP.NET/ASP.NET Core, connection strings in App Service override those in `Web.config`.
- For other languages, app settings are preferred unless you're using specific Azure database types for backups.

Bulk Editing Connection Strings:

- JSON format for connection strings:

```
[
  {
    "name": "name-1",
    "value": "conn-string-1",
    "type": "SQLServer",
    "slotSetting": false
  },
  {
```

```
    "name": "name-2",
    "value": "conn-string-2",
    "type": "PostgreSQL",
    "slotSetting": false
  }
]
```

Note for .NET Apps with PostgreSQL:

- Set the connection string type to **Custom** as a workaround for an issue in .NET.

Environment Variables for Custom Containers:

- To set environment variables for custom containers:
  - In **Bash** (Azure CLI):

```
az webapp config appsettings set --resource-group <group-name> --name
<app-name> --settings key1=value1 key2=value2
```

  - In **PowerShell** (Azure PowerShell):

```
Set-AzWebApp -ResourceGroupName <group-name> -Name <app-name> -
AppSettings @{"DB_HOST"="myownserver.mysql.database.azure.com"}
```

- **Verification**:
  - Use the URL `https://<app-name>.scm.azurewebsites.net/Env` to verify the environment variables in your running container.

Remember, when configuring your app, always apply your changes after editing to ensure they take effect.

General Settings Overview:

- Navigate to **Configuration > General settings** to manage these settings.

List of Available Settings:

1. Stack Settings:

- Define the software stack for your app:
  - Language and SDK versions.
  - For Linux and custom container apps:
    - Optional start-up command or file.

2. Platform Settings:

- **Platform Bitness**: Choose between 32-bit or 64-bit for Windows apps.
- **FTP State**:
  - Options include allowing only FTPS or disabling FTP altogether.
- **HTTP Version**:
  - Set to `2.0` for HTTP/2 protocol support.

```
# Note: Most browsers support HTTP/2 only over TLS. Ensure your custom DNS
is secured for HTTP/2 use.
```

- **Web Sockets**: Enable for real-time features like ASP.NET SignalR or socket.io.
- **Always On**:
  - Keeps the app loaded without traffic, preventing cold starts:

```
# Set to ON for maintaining continuous WebJobs or CRON triggered jobs.
```

- **ARR Affinity**:
  - Ensures session stickiness in multi-instance scenarios:

```
# Set to OFF for stateless applications.
```

- **HTTPS Only**:
  - Redirect all HTTP traffic to HTTPS.
- **Minimum TLS Version**:
  - Choose the minimum TLS version your app will accept.
- **Debugging**:
  - Enable remote debugging for specified app types. Automatically turns off after 48 hours.
- **Incoming Client Certificates**:
  - For mutual TLS authentication, forcing clients to provide certificates for access.

Important Notes:

- **Scaling**: Some settings might require scaling to higher pricing tiers.
- **HTTP/2**: Remember to secure your custom DNS name for HTTP/2 usage due to TLS requirements in modern browsers.

These settings are crucial for optimizing your application's performance, security, and compatibility with various technologies and protocols. Always consider your application's specific needs when configuring these options.

Path Mappings Overview:

- Navigate to **Configuration > Path mappings** to configure handler mappings and virtual applications/directories.

Windows Apps (Uncontainerized)

Handler Mappings:

- Custom script processors can be added for specific file extensions:
  - **Extension**: File extension to handle (e.g., `*.php`, `handler.fcgi`).
  - **Script processor**: Path to the script processor (e.g., `D:\home\site\wwwroot` for app root).

```
# Example configuration for a PHP handler:
# Extension: *.php
# Script processor: D:\home\site\wwwroot\php-cgi.exe
# Arguments: (optional)
```

  - **Arguments**: Optional command-line arguments.

Virtual Applications and Directories:

- Default path `/` is mapped to `D:\home\site\wwwroot`.
- To configure:
  - Specify virtual directory and physical path relative to `D:\home`.
  - Uncheck **Directory** to mark a virtual directory as a web application.

Linux and Containerized Apps

Custom Storage Mounts:

- For Linux apps and custom containers (both Windows and Linux):
  - Select **New Azure Storage Mount** to add custom storage:

```
# Basic Configuration
Name: [Display name of the mount]
Storage accounts: [Select your storage account]
Storage type: [Azure Blobs or Azure Files; Windows containers support only Azure
Files]
Storage container: [Select the container in basic setup]

# Advanced Configuration
Share name: [File share name for advanced setup]
Access key: [Access key for advanced setup]
```

```
Mount path: [Absolute path in container for mount point]
Deployment slot setting: [Check if this should apply to deployment slots]
```

Notes:

- **Azure Blobs** can only be mounted as read-only.
- **Deployment Slot Setting** ensures that storage mounts are consistent across deployment slots when checked.

These configurations allow you to customize how your application handles different file types and how it interacts with external storage, enhancing both functionality and performance based on your operational needs.

Enable Diagnostic Logging

Diagnostic Logging Overview:

- Azure provides various types of logs to help debug and monitor your App Service applications. Here's a breakdown:

Type	Platform	Location	Description
Application logging	Windows, Linux	App Service file system and/or Azure Storage blobs	Logs messages from your application code. Categories include: **Critical, Error, Warning, Info, Debug, Trace**.
Web server logging	Windows	App Service file system or Azure Storage blobs	Records raw HTTP request data in W3C extended log file format. Includes data like HTTP method, resource URI, client IP, client port, user agent, response code, etc.
Detailed error messages	Windows	App Service file system	Saves copies of error pages for HTTP errors >= 400. These are not sent to clients for security but are saved for internal review.
Failed request tracing	Windows	App Service file system	Provides detailed traces of failed requests, including which IIS components were involved and their processing time. Each failed request generates a folder with an XML log and an XSL stylesheet for viewing.
Deployment logging	Windows, Linux	App Service file system	Automatically logs deployment activities to help troubleshoot deployment failures. There are no specific settings to configure for this type of logging.

Notes:

- **Application Logging**:

```
# You can configure this to log to the file system or Azure Blob storage.
# Example configurations might involve setting up log levels and
destinations:
# az webapp log config --application-logging true --level information --web-
server-logging filesystem
```

- **Web Server Logging**:

```
# Can be stored in the file system or Azure Blob storage.
# az webapp log config --web-server-logging filesystem
```

- **Detailed Error Messages** and **Failed Request Tracing** are specific to Windows and offer deep insights into application errors and performance issues.
- **Deployment Logging**:

```
# This is automatic upon deployment, no configuration needed.
```

Enable Application Logging

For Windows:

- To enable application logging for Windows apps:
  - Navigate to your app in the Azure portal.
  - Select **App Service logs**.

Configuration Options:

- **Application Logging (Filesystem)**:
  - Set to **On** for temporary debugging. Automatically turns off after 12 hours.
  - This is useful for quick, short-term troubleshooting.
- **Application Logging (Blob)**:
  - Set to **On** for long-term logging.
  - Requires a blob storage container. Here's how you might set it up:

```
# Example Blob storage configuration
# az webapp log config --application-logging blob --level information -
-name <app-name> --resource-group <resource-group-name>
# --blob-storage-account <storage-account-name> --blob-container
<container-name>
```

Log Levels:

- Set the logging level for detail control:

```
# Log Levels:
# - Disabled: No logging
# - Error: Errors and Critical
# - Warning: Warnings, Errors, and Critical
# - Information: Info, Warning, Error, Critical
# - Verbose: All categories (Trace, Debug, Info, Warning, Error, Critical)
```

- **After configuration, remember to select Save to apply changes.**

Note:

- If you regenerate storage account keys, you must:
  - Turn logging off and then on again to update the configuration with the new keys.

For Linux/Container:

- In **App Service logs**:
  - Set **Application logging** to **File System**.

Quota and Retention:

- **Quota (MB)**: Define the disk space allowed for logs.

```
# Example configuration for quota
# az webapp log config --application-logging filesystem --level information
--name <app-name> --resource-group <resource-group-name> --quota 100
```

- **Retention Period (Days)**: Specify how long logs should be kept.

```
# Example for retention period
# az webapp log config --retention-in-days 7
```

- **When finished, select Save to apply the settings.**

These steps ensure you have the appropriate logging enabled for your Windows or Linux/Container apps in Azure, providing you with the necessary logs for debugging or monitoring purposes. Remember, for Linux apps, logs are only saved to the file system, not to Azure Blob Storage.

Enable Web Server Logging

- To configure web server logging:

- Select **Storage** for logs to be stored in blob storage, or **File System** for on-site storage on the App Service.

Retention Period:

- Set the **Retention Period (Days)** for how long logs should be kept.

```
# Example for setting retention period
# az webapp log config --web-server-logging filesystem --retention-in-days
30
```

- **When finished, select Save to apply the settings.**

Add Log Messages in Code

For ASP.NET:

```
System.Diagnostics.Trace.TraceError("If you're seeing this, something bad
happened");
```

For ASP.NET Core:

- Uses `Microsoft.Extensions.Logging.AzureAppServices` by default.

For Python:

- Use the `OpenCensus` package to send logs.

Stream Logs

- **Azure Portal**:
  - Navigate to your app and select **Log stream**.
- **Azure CLI**: Use for live streaming in Cloud Shell:

```
az webapp log tail --name appname --resource-group myResourceGroup
```

- **Local Console**:
  - Install Azure CLI, sign in, and follow the Azure CLI instructions.

Access Log Files

If using Azure Storage for Logs:

- You'll need a client tool that works with Azure Storage to access the logs.

For App Service File System Logs:

- **Download Logs for Linux/Container Apps:**

```
# URL for downloading logs
# https://<app-name>.scm.azurewebsites.net/api/logs/docker/zip
```

- **Download Logs for Windows Apps:**

```
# URL for downloading logs
# https://<app-name>.scm.azurewebsites.net/api/dump
```

Notes:

- For Linux/containers, the ZIP contains logs from both the docker host and container. For scaled-out apps, there's one set per instance.
- Logs are located in the `/home/LogFiles` directory on the App Service.

This setup allows you to configure where and how long logs are kept, log from within your application, stream logs in real-time, and retrieve them for analysis. Remember, some logs might not be in chronological order due to buffering.

Configure Security Certificates

Overview:

- Azure App Service provides various options to manage certificates for securing data transmission:

Option	Description
Create a free App Service managed certificate	A no-cost private certificate for securing custom domains within App Service.
Purchase an App Service certificate	A managed private certificate by Azure with features like automated management and flexible renewals.
Import a certificate from Key Vault	Suitable for integrating with Azure Key Vault for centralized certificate management.
Upload a private certificate	For pre-existing certificates from third-party providers, you can upload directly to App Service.
Upload a public certificate	For public certificates needed in application code, not for domain security.

Private Certificate Requirements:

- For certificates to be used in App Service, they must:

    - Be exported as a **password-protected PFX file** using triple DES encryption.
    - Have a **private key** of at least 2048 bits in length.
    - Include **all intermediate certificates** and the **root certificate** in the chain.

- For securing a custom domain with TLS binding, additional requirements are:

    - Must have an **Extended Key Usage** for server authentication (OID = 1.3.6.1.5.5.7.3.1).
    - Be signed by a **trusted certificate authority**.

Notes:

- Certificates uploaded to an app are stored within the deployment unit linked to the app service plan's resource group and region, making them available for other apps sharing the same environment.

This configuration ensures your application's data in transit is secure, using certificates either provided by Azure or brought in from external sources. Remember, when dealing with custom domains, the certificate must meet specific criteria for TLS/SSL bindings.

Creating a Free Managed Certificate

App Service Plan Requirements:

- Your App Service plan must be at least in the **Basic** tier (Standard, Premium, or Isolated also work).

Steps to Create a Free Managed Certificate:

1. **Prerequisites**:

    - **Custom DNS Configuration**: Ensure your custom domain is correctly set up with Azure App Service.
    - **CAA Record**: For some domains, you'll need to add a CAA record to allow DigiCert to issue certificates:

      ```
      0 issue digicert.com
      ```

2. **Certificate Management**:

    - Azure App Service will handle:
        - Certificate issuance by DigiCert.
        - Automatic renewal every six months, 45 days before expiration.

3. **Limitations**:

    - **No Wildcard Certificates**: You cannot use this for wildcard domains like `*.example.com`.
    - **Not for Client Certificates**: Cannot be used with client certificate authentication by thumbprint.
    - **No Private DNS**: Does not work with private DNS zones.
    - **Non-Exportable**: These certificates are managed by Azure and cannot be exported.
    - **ASE Incompatible**: Not supported in App Service Environment.
    - **Character Restrictions**: Only supports alphanumeric characters, dashes (-), and periods (.), with a maximum length of 64 characters for the domain name.

Key Considerations:

- **Avoid Certificate Pinning**: Since Azure manages these certificates, the root issuer or other aspects might change, so don't pin to specific certificate details.

This solution provides an effortless way to secure your custom domain in Azure App Service, but be mindful of its limitations, especially if you require more advanced certificate management features.

Importing an App Service Certificate

Azure Management for Purchased App Service Certificates:

- **Purchase**: Azure handles the procurement from the certificate provider.
- **Verification**: Automatically verifies the domain for the certificate.
- **Storage**: Stores the certificate in **Azure Key Vault**.
- **Renewal**: Manages the renewal process.
- **Synchronization**: Automatically updates or synchronizes the certificate across your App Service apps.

Actions with an Existing App Service Certificate:

- **Import**: You can import your App Service Certificate into Azure App Service:

```
# Example command to import a certificate:
# az webapp config ssl import --resource-group <resource-group-name> --name
<app-name> --certificate-name <certificate-name> --key-vault <key-vault-
name> --key-vault-certificate-name <vault-certificate-name>
```

- **Management**:
    - **Renew**: Extend the validity of your certificate.
    - **Rekey**: Generate a new key pair for the certificate, useful if you suspect key compromise.
    - **Export**: Export the certificate for use elsewhere if needed, although direct export from Azure might not be straightforward, you can download it from Key Vault.

Important Note:

- **Azure National Clouds**: App Service Certificates are **not supported** in these environments.

This functionality provides a straightforward method for managing SSL/TLS certificates within Azure, simplifying the process of securing your web applications with minimal manual intervention after the initial setup.

Azure App Service - Scaling Applications

Overview

- **Module Duration**: 22 minutes remaining
- **Progress**: 0 of 7 units completed

Learning Objectives

- **Understand Autoscale**: Learn how autoscale operates within the Azure App Service environment.
- **Identifying Autoscale Factors**:
  - **CPU Usage**: When your app exceeds a certain CPU threshold.
  - **Memory Usage**: Similar to CPU, but for memory.
  - **Queue Length**: Useful for applications dealing with message queues.
  - **Request Rate**: Number of HTTP requests over a period.
- **Enabling Autoscale**:
  - **Manual Scale**: Fixed number of instances.

```
az webapp update --name MyWebApp --resource-group MyResourceGroup --slots 3
```

  - **Auto Scale**: Dynamically adjusts based on defined rules.

```
az monitor autoscale create --resource MyWebApp --resource-group
MyResourceGroup --name MyAutoscale --min-count 1 --max-count 10 --count 1 --
recurrence '{ "timeZone": "Pacific Standard Time", "schedule": { "timeZone":
"Pacific Standard Time", "days": [ "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday" ], "hours": 9, "minutes": 0 } }'
```

- **Creating Sound Autoscale Conditions**:
  - **Thresholds**: Set appropriate thresholds for scaling actions.
  - **Cool Down Periods**: Prevent rapid scaling up and down by setting a cool down time.
  - **Prevent Over-Scaling**: Ensure you don't scale beyond what your system can handle or what's necessary.

Key Takeaways

- Autoscale helps in managing costs by scaling down during off-peak times and scaling up when demand increases.
- Proper configuration is crucial to avoid unnecessary scaling which can lead to higher costs or poor performance during unexpected traffic spikes.

Actions

- **Review Current Settings**: Check existing autoscale settings for your apps.
- **Configure Autoscale**: Use the Azure portal or CLI to set up or adjust autoscale rules.
- **Monitor and Adjust**: Regularly check performance metrics and adjust rules as needed for optimal operation.

This script provides an overview, key learning points, and examples of how to scale applications in Azure App Service using both manual and automatic scaling methods. Remember, the key to effective autoscaling is understanding your application's performance needs and setting conditions that align with those needs while keeping costs in check.

Introduction to Autoscaling in Azure

Key Points:

- **Autoscaling** dynamically adjusts the resources of your application based on usage, thereby optimizing performance and cost.

Learning Objectives:

1. **Identify Scenarios for Autoscaling:**
    - E-commerce sites during sales events
    - Social media platforms during viral content
    - Any application expecting variable load patterns
2. **Create Autoscaling Rules for a Web App:**
    - Rules are based on metrics like CPU usage, memory consumption, or request rates.

    Example of Setting Autoscaling for a Web App:

    ```
    az monitor autoscale create --resource MyWebApp --resource-group
    MyResourceGroup --name MyAutoscaleRules --min-count 1 --max-count 10 --count
    1 --recurrence '{ "timeZone": "Pacific Standard Time", "schedule": {
    "timeZone": "Pacific Standard Time", "days": [ "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday" ], "hours": 9, "minutes": 0 } }'
    ```

3. **Monitor the Effects of Autoscaling:**
    - Use Azure Monitor to track how autoscaling impacts your application.
    - Look for metrics like instance count, response times, and resource utilization.

    Example of Monitoring Autoscale Effects:

    ```
    az monitor metrics list --resource MyWebApp --metric-names CpuPercentage --
    interval PT1M --aggregation average
    ```

Glossary:

- **Autoscale Condition**: Criteria used to determine when to scale up or down.
- **Cool Down Period**: Time to wait before the next scaling action to prevent rapid scaling.

Why Autoscaling?

- **Scalability**: Handles traffic spikes without manual intervention.
- **Cost Efficiency**: Scales down during low usage to save on costs.
- **Performance**: Maintains performance by adding resources when needed.

Best Practices:

- **Set Realistic Thresholds**: Too low might lead to unnecessary scaling, too high might degrade performance.
- **Use Sensible Cool Down Periods**: Prevents rapid, costly fluctuations.
- **Test Autoscale Settings**: Simulate load to ensure autoscaling works as expected.

These notes cover the essentials of autoscaling in Azure, including why it's useful, how to implement it, and what to monitor after implementation. Remember, the actual metrics and thresholds you'll use will depend on your specific application's needs and behavior.

Azure Autoscaling & Automatic Scaling

Autoscaling Overview

- **Definition**: Autoscaling is a mechanism to adjust resources based on demand.
- **Functionality**: Scales **in** or **out** (not up or down).
    - **Triggers**:
        - **Schedule-based**
        - **Metric-based** (e.g., CPU, memory, requests)

Azure App Service Scaling Options

1. **Autoscaling with Azure Autoscale**

- **Mechanism**: Uses rules defined by the user to scale.
- **Purpose**: Adds or removes resources to meet demands.

2. **Azure App Service Automatic Scaling**

- **Mechanism**: Automatically scales based on selected parameters without user-defined rules.

Azure App Service Autoscaling Features

- **Monitors**: Resource metrics like CPU, memory, etc.
- **Action**:
    - Adds or removes **web servers**
    - Balances load across servers
- **Does Not**: Change individual server's CPU, memory, or storage capacity.

Key Points

- **Scaling Direction**:

    - **Out**: Adds more instances
    - **In**: Reduces instances
- **Response**: Proactive to ensure resources are available before overload occurs.

Remember, with Azure App Service, you're not just throwing more hardware at the problem; you're smartly managing your cloud resources to match your app's workload!

Azure Autoscaling Rules

Rule Definition

- **Purpose**: Define thresholds for metrics to trigger scaling events.
- **Components**:
    - **Trigger**: Metric crossing a specified threshold.
    - **Action**: Scale **in** or **out** accordingly.

Considerations for Rules:

- **DoS Protection**: Avoid scaling in response to malicious traffic spikes. Instead:
    - Implement request filtering and detection mechanisms.

When to Use Autoscaling

Benefits:

- **Elasticity**: Scales with workload changes, ideal for:
    - Holiday traffic spikes in e-commerce.
- **Availability & Fault Tolerance**:
    - Prevents request denial due to instance overload or crash.

Limitations:

- **Resource-Intensive Requests**:
    - If each request requires significant processing:

```
- Autoscaling might not suffice; consider manual scaling up instead.
```

- **Long-Term Growth**:
    - Autoscaling has monitoring overhead. Manual scaling might be more cost-effective for predictable growth.

Instance Count:

- **Few Instances**:
    - Limited initial capacity can lead to service downtime despite autoscaling.

Best Practices:

- **Careful Rule Settings**: Ensure rules do not misinterpret malicious traffic as legitimate load.
- **Evaluate Workload**: Determine if autoscaling meets your application's processing demands or if manual intervention is needed.
- **Monitor and Adjust**: Regularly review autoscaling performance against your app's real-world demands to optimize cost and performance.

Remember, autoscaling is not a one-size-fits-all solution. It's about having the right number of servers at the right time, not necessarily about having the most powerful servers all the time.

Azure App Service Automatic Scaling

Overview:

- **Purpose**: Automatically manages scale-out decisions for web apps and App Service Plans.
- **Difference from Azure Autoscale**:
  - **Autoscale**: User-defined rules based on schedules/resources.
  - **Automatic Scaling**: Platform makes decisions based on app performance needs.

Features:

- **Prewarming**:
  - The system prewarms instances to buffer performance during scale-out events.
- **Billing**:
  - **Per-second**: You're charged for each instance, including prewarmed ones.

When to Use Automatic Scaling:

1. **Simplified Configuration**:
   - If you wish to avoid the complexity of setting up and managing autoscale rules.
2. **Independent Scaling**:
   - When you need different web apps within the same App Service Plan to scale independently.
3. **Backend Integration**:
   - For web apps linked with slower-scaling backends like databases or legacy systems:
     - **Setting Limits**: You can define a maximum number of instances to prevent overwhelming backend resources.

Key Points:

- Automatic scaling aims to:
  - **Improve Performance**: By avoiding cold starts and ensuring smooth performance transitions.
  - **Balance Load**: Automatically adjusts to the workload without manual rule setting.

Remember, with automatic scaling, you're paying for flexibility and performance. Every second counts, quite literally!

Azure Autoscaling Factors

Purpose of Autoscaling:

- **Ensure Resources**: Available for high demand periods.
- **Cost Management**: Scale back during low demand.

Configuration Options:

- **Resource-Based Scaling**:
  - Scales based on metrics like CPU usage, memory, etc.
- **Schedule-Based Scaling**:
  - Scales according to predefined time schedules.

Understanding Autoscaling in App Service Plans:

App Service Plan and Autoscaling:

- Autoscaling is tied to the **App Service Plan** (ASP) configuration of the web app.
  - **Scaling Out**: Azure provisions new hardware instances as specified by the ASP.

Instance Limits:

- **Prevent Over-Scaling**: Each ASP has a limit on how many instances can be created:
  - **Pricing Tier**: Determines the maximum number of instances for autoscaling.

Important Note:

- **Tier Limitations**: Not all pricing tiers of the App Service Plan support autoscaling. Check your plan's capabilities before relying on autoscaling.

Key Takeaways:

- **Metrics**: Use resource metrics smartly to trigger scaling events.
- **Scheduling**: Anticipate regular demand changes with schedule-based autoscaling.
- **Plan Limits**: Be aware of your App Service Plan's instance limit to manage expectations and costs.

Autoscale Conditions in Azure

Autoscaling Options:

- **Metric-Based Scaling**:
  - **Example Metrics**:
    - Disk queue length
    - Number of HTTP requests pending

- **Schedule-Based Scaling**:
  - **Example**: Scale out at specific times/days, with an end date for scaling back in.

Combining Scaling Approaches:

- Can combine both metric and schedule-based scaling for more nuanced control.
  - **Example**: Scale out based on HTTP requests but only during business hours.

Multiple Conditions:

- **Multiple Rules**: Different conditions can be set for various scenarios.
- **Default Condition**: Always active if other conditions don't apply; no schedule needed.

Metrics for Autoscale Rules:

- **CPU Percentage**: Indicates CPU utilization across instances.
  - High value = potential for processing delays.
- **Memory Percentage**: Shows memory usage.
  - High value = risk of memory depletion.
- **Disk Queue Length**: Measures outstanding I/O requests.
  - High value = potential disk contention.
- **Http Queue Length**: Shows pending HTTP requests.
  - Large number = risk of HTTP 408 errors.
- **Data In/Out**: Monitors network traffic in and out.

Cross-Service Metrics:

- Can also scale based on metrics from other Azure services, e.g., Azure Service Bus Queue length.

Key Points:

- **Autoscale Rules**: Define when to scale based on metrics.
- **Flexibility**: Combine rules to match your app's unique traffic patterns and resource needs.
- **Monitoring**: Keep an eye on multiple dimensions to ensure your app scales appropriately.

Remember, setting up autoscaling is like setting up a smart thermostat for your app's environment - it keeps everything at just the right level without you having to do much more than define the comfort zones.

How Autoscale Rules Analyze Metrics

Metric Analysis Steps:

1. **Time Grain Aggregation**:

- **Definition**: Aggregates metric values for all instances over a short time period.
  - **Typical Time Grain**: 1 minute.
  - **Aggregation Options**:
    - Average
    - Minimum

        - Maximum
        - Sum
        - Last
        - Count

2. **Duration Aggregation**:

- **Purpose**: To assess if changes are significant enough for scaling.
    - **Duration**: User-defined period, minimum of 5 minutes.
    - **Example**: If Duration is 10 minutes, it aggregates 10 time grain values.
    - **Aggregation Method**: Can differ from time grain aggregation.
        - **Example**:
            - Time Grain: Average CPU%
            - Duration Aggregation: Maximum of those averages over 10 minutes.

Autoscale Actions:

- **Scale-Out**: Increases instance count.
- **Scale-In**: Reduces instance count.
- **Operators**:
    - `>` for scale-out (when metric exceeds threshold)
    - `<` for scale-in (when metric falls below threshold)
- **Set Instance Count**: Can also directly set the number of instances.

Cooldown Period:

- **Definition**: Time frame post-action where no new scaling occurs.
    - **Purpose**: Allows system stabilization.
    - **Minimum**: 5 minutes.

Key Points:

- Autoscaling involves **trend analysis** over time.
- **Time Grain** helps in quick response to changes.
- **Duration** ensures that the change isn't just a blip.
- **Cooldown** prevents overreaction to temporary spikes or drops in metrics.

Remember, autoscaling is like adjusting sails on a boat; you're looking at the wind (metrics) over time, not just the gusts (momentary spikes), and you give the ship (your app) time to adjust to the new direction before making another change.

Pairing and Combining Autoscale Rules

Paired Autoscale Rules:

- **Strategy**: Define rules in pairs for both scaling out and in.

- **Scale-Out Rule**: Triggers when metrics exceed an upper limit.
- **Scale-In Rule**: Triggers when metrics fall below a lower limit.

Combining Autoscale Rules:

Example Configuration:

```
- **Rules within one condition**:
  - **Scale Out**:
    - HTTP queue length > 10 → Add 1 instance
    - CPU utilization > 70% → Add 1 instance
  - **Scale In**:
    - HTTP queue length = 0 → Remove 1 instance
    - CPU utilization < 50% → Remove 1 instance
```

Rule Execution:

- **Scaling Out**: Occurs if **any** of the scale-out conditions are met.
- **Scaling In**: Only happens when **all** scale-in conditions are true.

Separate Conditions for Different Logic:

- If individual scale-in rules should trigger independently, they must be in **separate autoscale conditions**.

Key Points:

- **Pairing**: Helps manage both increase and decrease in workload efficiently.
- **Combining**: Allows for more complex scaling behavior tailored to different scenarios.
- **Condition Logic**:
  - Use OR logic for scaling out (any condition can trigger).
  - Use AND logic for scaling in (all conditions must be met) unless specified otherwise.

Remember, with autoscaling, you're not just telling your app to grow or shrink; you're teaching it when to hibernate or when to go full Hulk mode, ensuring it's always just the right size for the task at hand.

Enable Autoscaling in Azure App Service

To Start Autoscaling:

1. **Navigate**:
   - Go to your **App Service Plan** in the Azure portal.
   - Find **Scale out (App Service plan)** under *Settings*.

Pricing Tier Consideration:

- **Note**:
  - **Development Tiers**:

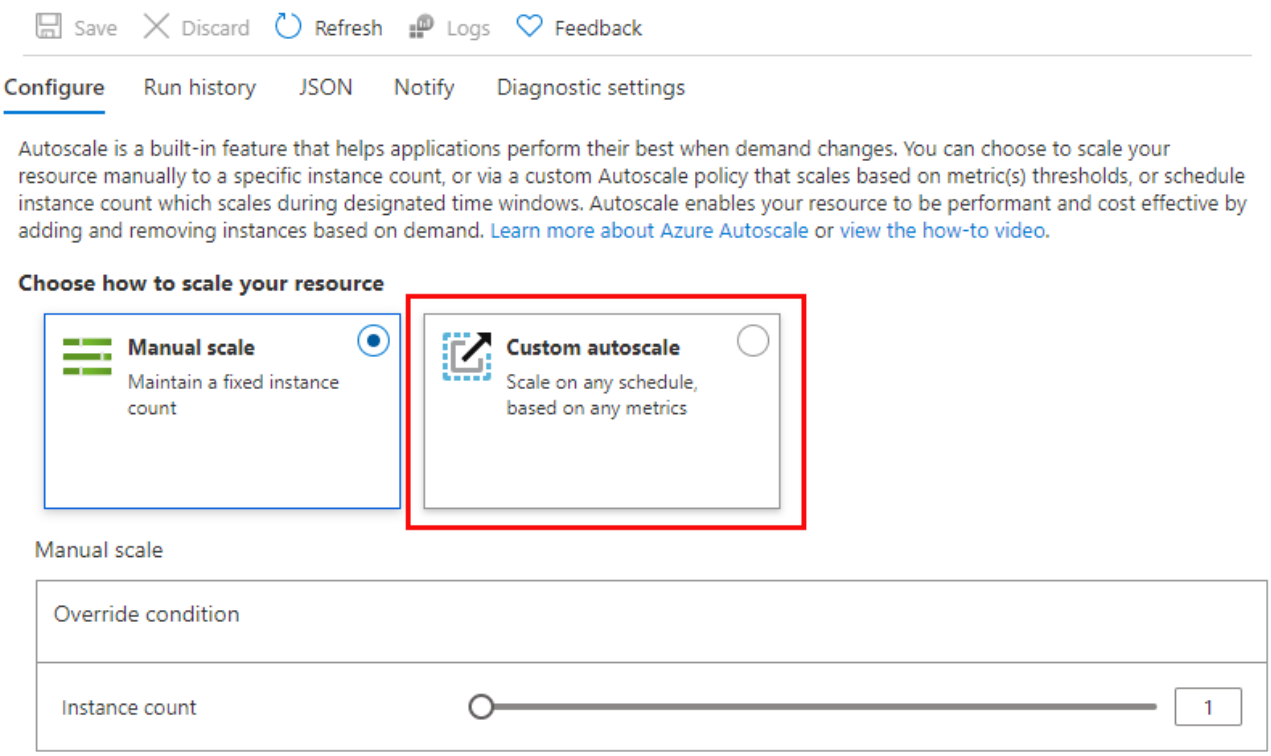
        - **F1, D1** (single instance)
        - **B1** (manual scaling) do not support autoscaling.
    - **Action**: Upgrade to **S1 or any P-tier** for autoscaling capability.

Enabling Autoscaling:

- **Default State**: Manual scaling.
- **Custom Autoscaling**:
    - Select **Custom autoscale** to access condition groups for scaling settings.

Visual Guide:

- **Azure Portal Navigation**:

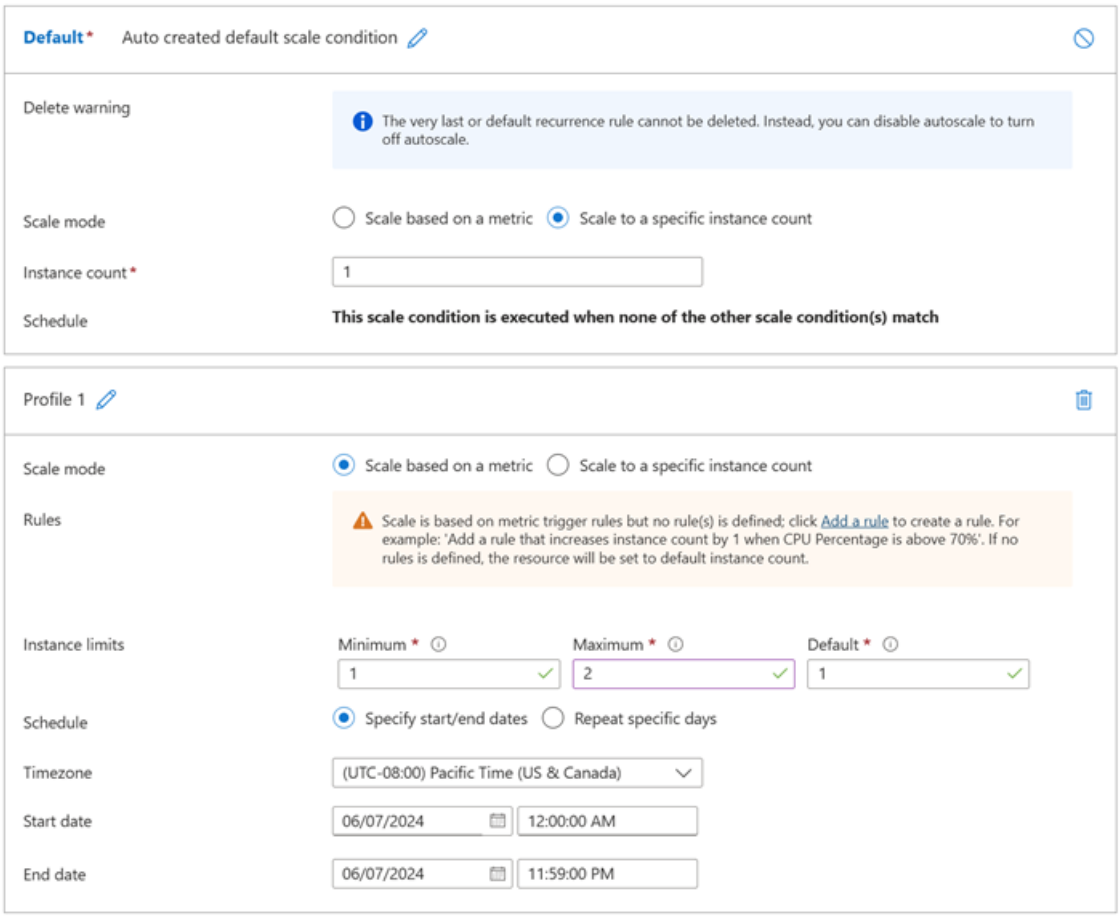


Adding Scale Conditions:

Process:

- **Default Condition**:
    - Initially set to manual scaling, can be edited.
    - Triggered when no other conditions apply.
- **Custom Conditions**:
    - **Type**:
        - **Metric-Based**: Scale according to resource metrics like CPU usage or HTTP queue length.
        - **Instance Count**: Set to scale to a specific number of instances.
    - **Limits**:
        - **Minimum/Maximum Instances**: Can be set, with max limited by your pricing tier.

- **Condition Page Example**:



  - **Schedules**:
    - Conditions can have a schedule defining active periods.

Key Actions:

- **Enable**: Switch to Custom autoscale for automatic scaling.
- **Configure**: Set or edit rules for scaling behavior.
- **Monitor**: Keep track of autoscaling activity through the Azure portal.

Remember, setting up autoscaling is like tuning an instrument; you need to get the settings just right to ensure your application performs harmoniously under varying loads.

Creating and Monitoring Scale Rules in Azure

Create Scale Rules:

- **Metric-Based Scale Conditions**:
  - Include **one or more scale rules**.
  - Use **Add a rule** to define custom rules.

Rule Definition:

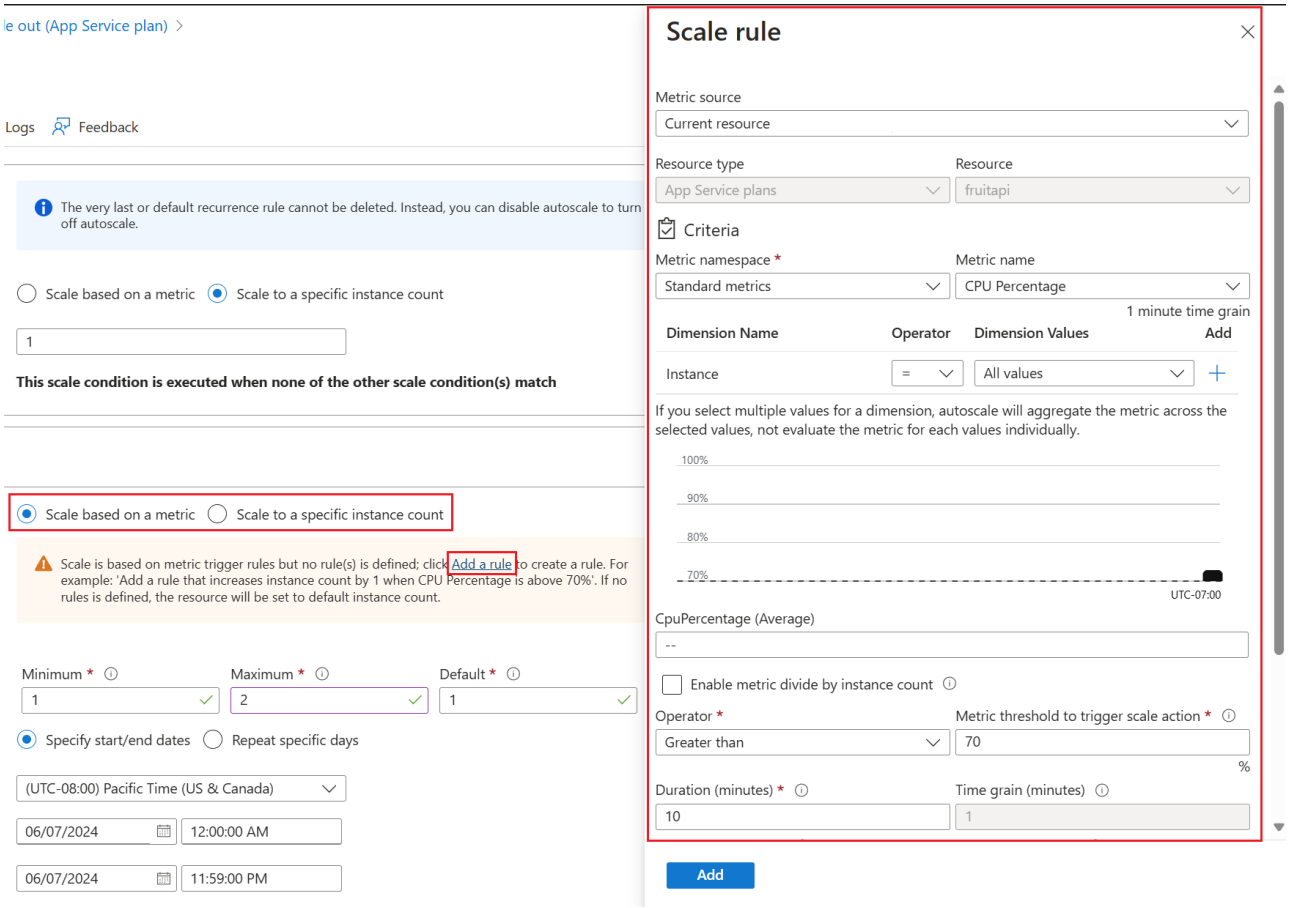
- **Criteria**:
  - Metrics to monitor

- Aggregation methods (Average, Minimum, Maximum, etc.)
- Operators (>, <, etc.)
- Thresholds for triggering actions

- **Autoscale Action**:
  - **Scale Out** or **Scale In** based on the rule's criteria.

Visual Reference:

- **Scale Rule Settings**:

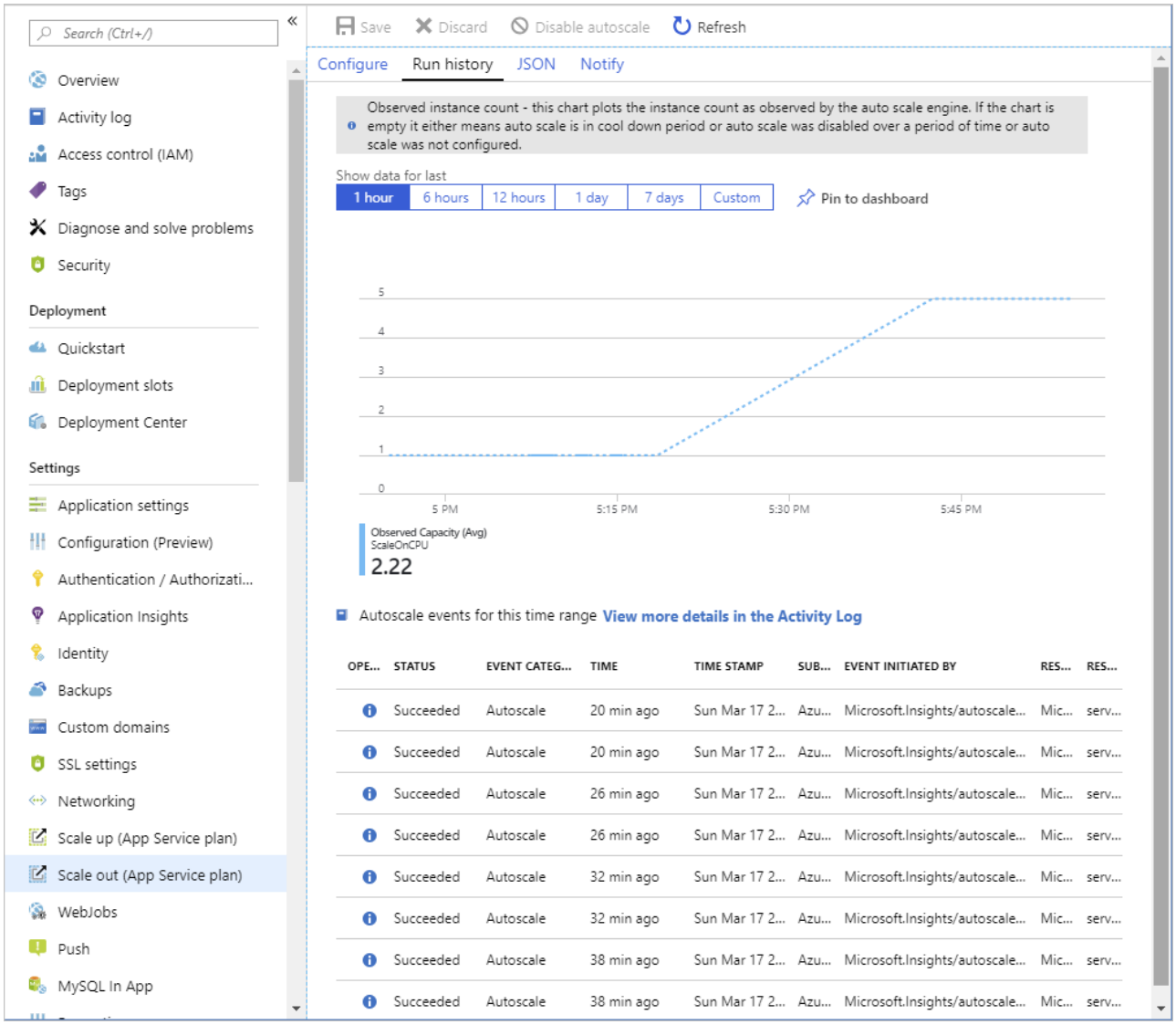


Monitor Autoscaling Activity:

- **Run History Chart**:
  - **Azure Portal** provides the **Run history chart** to track autoscaling events.
  - **Purpose**:
    - Shows **instance count changes** over time.
    - Identifies **which autoscale condition** triggered each change.

Visual Reference:

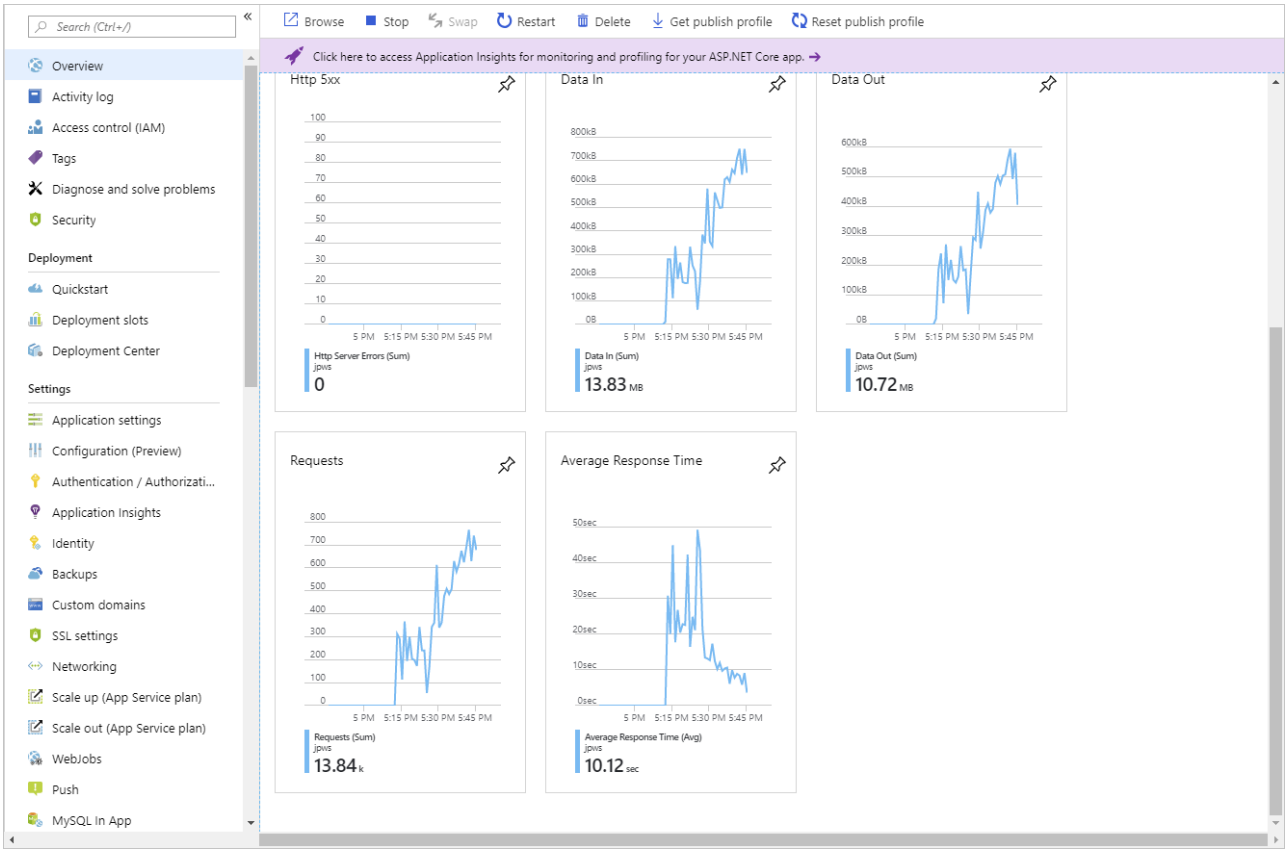
- **Run History Chart**:



- **Correlating with Metrics**:
    - Use the Run history alongside metrics on the **Overview page** to understand resource utilization at the time of autoscaling events.

Visual Reference:

- **Overview Metrics**:



Key Actions:

- **Create**: Define rules with specific metrics and conditions for scaling.
- **Monitor**: Use Azure's tools to review and analyze autoscaling events and resource metrics.

Remember, setting up scale rules is like fine-tuning an engine; you adjust the parameters until your application runs smoothly, scaling appropriately with the demands placed upon it.

Azure Autoscale Notes

Autoscale Concepts:

- **Horizontal Scaling:**
  - **Scale Out:** Increase the number of instances when demand grows.
  - **Scale In:** Decrease the number of instances when demand reduces.
  - An autoscale setting includes:
    - **Maximum Instances:** The upper limit of instances.
    - **Minimum Instances:** The lower limit of instances.
    - **Default Instances:** The starting number of instances before any scaling occurs.

Threshold Mechanics:

- Autoscale operates by monitoring metrics:
  - It checks if the metric (like CPU usage) has exceeded a threshold.
  - For example:

```
"scale out by one instance when average CPU > 80% when instance count
is 2"
```

    - This means, if the average CPU usage across all instances exceeds 80% with currently 2 instances, add one more instance.

Monitoring and Logging:

- **Activity Log:**
    - All scaling actions, successful or not, are logged here.
    - You can set up alerts for:
        - **Email**
        - **SMS**
        - **Webhooks**

Best Practices:

- **Avoid Conflicting Rules:** Ensure your autoscale rules do not counteract each other, causing unnecessary scaling actions.

These practices help in managing your resources efficiently, ensuring your systems scale appropriately with workload demands while avoiding over or under-provisioning. Remember, the last thing you want is your server farm looking like a yo-yo at a children's party – up and down, up and down. Keep those scales balanced, my friend!

Azure Autoscale Best Practices

Setting Instance Limits:

- **Margin Between Limits:** Ensure there's a gap between your `minimum` and `maximum` instance settings. If both are set to `2`, no scaling will occur. Here's an example:

```
minimum=2
maximum=5
```

Diagnostic Metric Selection:

- **Statistic for Scaling:** Choose from `Average`, `Minimum`, `Maximum`, or `Total` when setting up autoscale based on metrics. Typically, `Average` is used.

```
metric_statistic = 'Average'
```

Threshold Selection:

- **Avoid Similar Thresholds:** Do not set autoscale thresholds too close for scale-out and scale-in. This can lead to an oscillating effect known as "flapping."

Bad Example:

```
scale_out_threshold = 600  # When thread count >= 600
scale_in_threshold = 600   # When thread count <= 600
```

Better Practice:

```
scale_out_threshold = 80  # When CPU% >= 80%
scale_in_threshold = 60   # When CPU% <= 60%
```

Preventing "Flapping":

- Autoscale uses estimation to prevent unnecessary scaling:
  - When considering scaling in, it calculates if scaling out would be immediate upon scaling in:

    ```
    If current CPU% is 60 across 3 instances:
    60 x 3 = 180 threads total
    180 / 2 (after scaling in) = 90 threads per instance
    ```

    - Since 90 is above the scale-out threshold, it won't scale in to avoid flapping.

Example Scenario:

- **Starting Point:** 2 instances
- **Scale Out:** CPU hits 80%, a third instance is added.
- **Scale In Consideration:** CPU drops to 60%, but the system estimates:

  ```
  60 x 3 = 180 total / 2 = 90 threads per instance after scaling in
  ```

  - No scale-in occurs because 90 is still near the scale-out threshold.
- **Next Check:** CPU drops further to 50%, estimation now allows scale-in:

  ```
  50 x 3 = 150 total / 2 = 75 threads per instance
  ```

  - Now, scaling in to 2 instances is performed as 75 is below the scale-out threshold.

Conclusion:

- Always ensure there's a buffer between your scale-out and scale-in thresholds to prevent rapid back-and-forth scaling actions. This not only saves resources but also keeps your system's stability in check.

Remember, your servers should scale like a well-oiled machine, not like the mood swings of a teenager.

Azure Autoscale Considerations for Multiple Rules

Multiple Rules in a Profile:

- **Scale-Out:** Autoscale will scale out if **any** of the scale-out rules are met.
- **Scale-In:** Autoscale will only scale in if **all** scale-in rules are satisfied.

Example with Four Rules:

- **Scale-Out Rules:**
  - CPU > 75%, scale out by 1
  - Memory > 75%, scale out by 1
- **Scale-In Rules:**
  - CPU < 30%, scale in by 1
  - Memory < 50%, scale in by 1

  Scenarios:

  - **Scale-Out Cases:**
    - CPU at 76% and Memory at 50% -> Scale out (One rule met)
    - CPU at 50% and Memory at 76% -> Scale out (One rule met)
  - **No Scale-In Cases:**
    - CPU at 25% but Memory at 51% -> No scale in (Both rules not met)
  - **Scale-In Case:**
    - CPU at 29% and Memory at 49% -> Scale in (Both rules met)

Default Instance Count:

- **Importance:** Set a safe default instance count. This count is used when metrics are unavailable for scaling decisions.
  - **Example:**

```
default_instance_count = 3  # Choose based on minimal safe operation
level
```

Autoscale Notifications:

- **Activity Log:** Autoscale logs these events:
  - When it issues a scale operation.
  - Successful completion of a scale action.

    - Failure to scale.
    - When metrics are unavailable or become available again.

- **Setting Up Alerts:**

    - Use Activity Log alerts to monitor the autoscale engine's health.

    - Configure notifications for successful scale actions:

```
notifications {
  email: ["admin@example.com", "support@example.com"],
  webhook: "https://example.com/autoscale-webhook"
}
```

Conclusion:

- When configuring multiple rules, remember that scale-out is more lenient while scale-in requires all conditions to be met to avoid unnecessary scaling. Ensure your default instance count is set to handle your workload safely when metrics fail. And don't forget to set up notifications; it's always good to know when your system decides to grow or shrink, just like keeping tabs on whether your garden needs watering or not.

Azure App Service Deployment Slots Overview

What are Deployment Slots?

- Deployment slots in Azure App Service provide a staging environment where you can deploy your app before swapping it into production. This minimizes downtime and risk during application updates.

Key Concepts:

- **Slot Swapping:** The process of switching the environment (staging to production or vice versa) without downtime.
- **Manual Traffic Routing:** Allows you to control the flow of traffic to different slots for testing purposes.
- **Automatic Traffic Routing:** Can be set up to automatically route traffic based on certain conditions or rules.

How Slot Swapping Works:

- When you swap slots:

    1. The staging slot's content becomes the new production content.
    2. The old production content moves to the staging slot, allowing for rollback if necessary.

```
# Pseudocode for a slot swap
swap_slots(sourceSlotName="staging", targetSlotName="production")
```

Manual Traffic Routing:

- You can manually route traffic to test new features or changes in a staging environment.

```
# Example of routing 50% of traffic to staging
set_traffic_route(productionSlot="production", stagingSlot="staging",
percentage=50)
```

Automatic Traffic Routing:

- Automate traffic based on conditions like:
  - Time of day (e.g., sending more traffic to staging during low-traffic hours).
  - Performance metrics (if staging performs better, gradually shift more traffic).

```
# Example of setting up an automatic rule
add_automatic_routing_rule(condition="timeOfDay", startTime="22:00",
endTime="06:00", targetSlot="staging")
```

Benefits of Using Slots:

- **Zero Downtime Deployment:** Swap environments without taking the app offline.
- **Easy Rollback:** If something goes wrong, swap back to the previous state.
- **Testing in Production:** Test new versions with real-world data and traffic patterns.

Steps to Perform a Swap:

1. **Deploy:** Push your new version to a staging slot.
2. **Test:** Validate in the staging environment.
3. **Swap:** Use Azure portal or CLI to swap the slots.

```
# CLI command for slot swapping
az webapp deployment slot swap --resource-group MyResourceGroup --name
MyWebApp --slot staging
```

Conclusion:

Deployment slots in Azure App Service are your magic wand for seamless application updates. They allow you to experiment in a production-like setting without affecting your live website, ensuring that your app deployment is as smooth as a jazz tune. Remember, with great power comes great responsibility—use slots wisely to avoid any production blues.

Azure App Service Deployment Slots Introduction

Overview:

- Deployment slots in Azure App Service facilitate a robust environment for managing application versions, allowing developers to stage, test, and deploy updates with minimal risk.

Learning Objectives:

- **Understand Benefits:** Learn why using deployment slots is advantageous for application management.
- **Slot Swapping:** Grasp the mechanism behind swapping slots in App Service.
- **Manual vs Auto Swap:** Know how to manually swap slots and set up auto-swapping.
- **Traffic Management:** Learn techniques for routing traffic both manually and automatically.

Key Points:

- **Benefits of Deployment Slots:**
  - **Zero-downtime deployments**: Update your app without interrupting service.
  - **Testing in Production Environment**: Test with real user traffic without affecting the live app.
  - **Easy Rollback**: Revert changes by swapping back if issues arise post-deployment.
- **Slot Swapping Mechanics:**
  - Swap operation moves the content from one slot (usually staging) to another (production).
  - Example swap command:

```
az webapp deployment slot swap --resource-group MyResourceGroup --name
MyWebApp --slot staging
```

- **Manual Swap and Auto Swap:**
  - **Manual Swap:** Directly control when to swap using the Azure portal, CLI, or API.
  - **Auto Swap:** Configure auto-swapping to occur when certain conditions are met, like after successful deployment.

```
# Enable auto swap for a slot
az webapp deployment slot auto-swap --name MyWebApp --resource-group
MyResourceGroup --slot staging --enable
```

- **Traffic Routing:**
  - **Manual Routing:** Use for A/B testing, where you control how much traffic goes to each slot.

```
# Route 30% traffic to staging slot
az webapp traffic-routing set --name MyWebApp --resource-group
MyResourceGroup --distribution staging=30 production=70
```

- **Automatic Routing:** Set rules or use Azure Traffic Manager for dynamic traffic distribution based on performance or other metrics.

Prerequisites:

- Prior experience with Azure portal to ensure you're familiar with managing App Service web apps.

Conclusion:

Deployment slots are like having a safety net for your app updates, allowing you to juggle different versions of your application with the finesse of a circus performer. This module sets you up to not only manage your app's lifecycle with ease but also to do it with the confidence of knowing you can always catch your app with the net if it falls.

Staging Environments in Azure App Service

Overview:

- Deployment slots are available in Standard, Premium, and Isolated tiers, allowing deployment to environments other than the default production slot.

Benefits of Using Staging Slots:

- **Pre-Deployment Testing:** Validate changes before they hit production.

```
// Pseudo-code for deploying to a staging slot
deploy_to_slot(appName="MyWebApp", slotName="staging", source="dev-branch")
```

- **Warm-Up:** Ensures all instances of the slot are warmed up before going live, avoiding downtime:

```
# Pseudo-code for warming up instances
warm_up_instances(slotName="staging")
```

- **Seamless Traffic Redirection:** Swapping slots redirects traffic instantly without losing requests.
- **Rollback Capability:** Swap back to the previous version if something goes wrong:

```
# Pseudo-code for rollback
swap_slots(sourceSlot="production", targetSlot="staging")
```

- **Automation:** Auto-swap can be configured for immediate production deployment post-validation.

```
# Pseudo-code to enable auto-swap
enable_auto_swap(slotName="staging")
```

Slot Management:

- **Slot Limitations:**
  - Each tier supports a different number of slots. Check App Service limits for your tier's capacity.
  - Scaling down requires the new tier to support the current slot number.

Slot Creation and Configuration:

- **Content:** New slots start empty but can be cloned from another slot for settings.

```
# Pseudo-code to clone settings from production
clone_slot_settings(productionSlot="production", newSlot="staging")
```

- **Deployment:** Deploy different versions or branches to staging slots for testing.

Important Notes:

- When scaling your app, ensure the target tier supports your current slot usage. For instance, Standard tier supports up to five slots.
- Deployment slots provide a safe playground for your app's updates. Think of them as your app's dress rehearsal room where it can perform without the audience until it's showtime in production.

Slot Swapping in Azure App Service

Slot Swapping Process:

1. **Configuration Application:**
   - Apply settings from the target slot (e.g., production) to the source slot (e.g., staging):
     - Slot-specific settings like app settings and connection strings.
     - Continuous deployment settings.
     - App Service authentication settings.

```
# Pseudo-code for applying settings
apply_target_settings(sourceSlot="staging")
```

   - This triggers a restart of all instances in the source slot.
2. **Restart Validation:**
   - Wait for all instances in the source slot to restart successfully. If an instance fails, the swap reverts.
3. **Local Cache Initialization:**
   - If local cache is enabled, an HTTP request to `/` on each instance initializes the cache, causing another restart.

```
# Pseudo-code for local cache initialization
init_local_cache(slot="staging")
```

4. **Application Warm-Up:**
    - If auto swap with custom warm-up is enabled, hit `/` to warm up the application:

```
# Pseudo-code for application warm-up
warm_up_application(slot="staging")
```

5. **Swap Execution:**
    - Once all instances are warmed up, swap the routing rules of the slots:

```
# Pseudo-code for executing the swap
execute_slot_swap(sourceSlot="staging", targetSlot="production")
```

6. **Post-Swap Actions:**
    - Apply the original target slot's settings to what is now the new source slot, ensuring consistency.

Key Considerations:

- **No Downtime:** The target slot remains online during the swap process, ensuring no downtime for production.
- **Swap Direction:** Always swap into the production slot as the target to minimize impact on live traffic.
- **Configuration Cloning:** When cloning config, remember:
    - **Content-Following Settings:** These settings move with the app during a swap.
    - **Slot-Specific Settings:** These stay with their respective slots after a swap.

Configuration Elements During Swap:

- **App Settings:** Slot specific unless marked to stick with content.
- **Connection Strings:** Similar to app settings, can be configured to follow content or remain slot-specific.
- **Publishing Profile:** Stays with the slot.

Conclusion:

Slot swapping in Azure is like performing a magic trick. You wave your wand (execute the swap), and suddenly your staging slot's content appears in production without the audience (your users) noticing any downtime. Just make sure you've got your hat (configuration) straight before pulling the rabbit (your app) out.

Configuration Settings in Azure App Service Slot Swapping

Swappable vs. Non-Swappable Settings:

- **Settings That Are Swapped:**
  - General settings like framework version, 32/64-bit architecture, and WebSockets.
  - **App Settings:** Can be configured to stick to a slot or follow content to a new slot.

```
# Example of an app setting that follows content
WEBSITE_NODE_DEFAULT_VERSION=12.13.0
```

  - **Connection Strings:** Similar to app settings, can be set to stick to a slot or swap with the app.

```
# Example of a connection string
SQLAZURECONNSTR_defaultConnection=Server=tcp:servername.database.window
s.net,1433;Database=databaseName;User
ID=userName;Password=password;Trusted_Connection=False;Encrypt=True;
```

  - Handler mappings.
  - Public certificates.
  - WebJobs content.
  - Path mappings.
- **Settings That Aren't Swapped:**
  - Publishing endpoints.
  - Custom domain names.
  - Non-public certificates and TLS/SSL settings.
  - Scale settings.
  - WebJobs schedulers.
  - IP restrictions.
  - Always On settings.
  - **Azure Content Delivery Network** (planned to be unswapped).
  - **Service Endpoints** (planned to be unswapped).
  - Diagnostic log settings.
  - Cross-origin resource sharing (CORS).
  - Virtual network integration.
  - Managed identities.
  - Settings ending with `_EXTENSION_VERSION`.

Making Settings Swappable:

- To override the default behavior where certain settings are sticky to slots:

```
# App setting to override sticky slot settings
WEBSITE_OVERRIDE_PRESERVE_DEFAULT_STICKY_SLOT_SETTINGS=0
```

This setting should be added to every slot. Setting it to `0` or `false` makes all settings swappable.

Configuring Slot-Specific Settings:

- To make an app setting or connection string slot-specific (not swappable):
    - Navigate to the slot's **Configuration** page.
    - Add or edit the setting.
    - Check the **Deployment slot setting** box.

```
# Setting a slot-specific app setting (example)
APPSETTING_DEBUG=true [Deployment Slot Setting]
```

Note:

- Managed identities are always slot-specific and are not affected by the override setting.

Conclusion:

In Azure App Service, you can tailor how settings behave during slot swaps, giving you the flexibility to keep certain configurations consistent across environments while allowing others to adapt. Think of it like setting the stage: you want the backdrop (slot-specific settings) to remain, but you're swapping the actors (your app) and their props (content and swappable settings) for a new scene.

Manual Slot Swapping in Azure App Service

Steps to Swap Deployment Slots:

1. **Access the Deployment Slots Page:**
    - Navigate to your app in Azure portal, then to the **Deployment slots** page.
2. **Initiate Swap:**
    - Click on **Swap** to open the swap dialog.

```
# Pseudo-code for initiating a swap
initiate_swap(sourceSlot="staging", targetSlot="production")
```

3. **Select Slots:**
    - Choose **Source** (e.g., staging) and **Target** (typically production) slots. Verify settings in both tabs:
        - **Source Changes:** Settings that will be applied to the source slot.
        - **Target Changes:** Settings that will be applied to the target slot post-swap.
4. **Immediate Swap:**

- If you're ready to swap without preview, click **Swap** to execute the operation.

5. **Swap with Preview:**
   - For validation, check the **Perform swap with preview** option. This begins a multi-phase swap:
     - **Phase 1:** Applies the target slot's settings to the source slot, then pauses.

```
# Pseudo-code for starting the swap with preview
start_swap_with_preview(sourceSlot="staging",
targetSlot="production")
```

     - **Preview:** Visit the source slot URL to test the app with new settings:

```
# URL format for previewing the swap
https://<app_name>-<source-slot-name>.azurewebsites.net
```

     - **Phase 2:** If satisfied with the preview, select **Complete Swap** to finalize the swap.

```
# Pseudo-code for completing the swap
complete_swap()
```

   - **Cancel:** If issues are found during the preview, select **Cancel Swap** to revert changes.

6. **Close Dialog:**
   - After the swap or cancellation, click **Close** to exit the swap interface.

Key Points:

- **Validation:** Always ensure the production slot is the target to minimize downtime.
- **Configuration Review:** Review settings in both slots before proceeding with the swap.
- **Warm-Up:** Swap with preview ensures the source slot is warmed up, reducing the risk of performance issues.
- **Rollback:** If a swap with preview is canceled, the configuration reverts, providing a safety net.

Conclusion:

Swapping deployment slots in Azure App Service is like changing the soup of the day in a restaurant's menu - you want to make sure the new flavor (staging environment) is just right before serving it to all the patrons (your production users). With the option to preview, you can taste the soup before making it the special of the day, ensuring it's a hit.

Configuring Auto Swap in Azure App Service

Steps to Enable Auto Swap:

1. **Navigate to the Slot Configuration:**
    - Go to your App Service's resource page in Azure portal.
    - Click on the deployment slot you're configuring for auto swap.
2. **Enable Auto Swap:**
    - Go to **Configuration** > **General settings**.
    - Turn **Auto swap enabled** to **On** and select the **target slot** for auto swap.

```
# Pseudo-code for enabling auto swap
enable_auto_swap(sourceSlot="staging", targetSlot="production")
```

    - Save the changes.
3. **Code Push:**
    - Push your code to the source slot. Auto swap will occur automatically post-warm-up.

Custom Warm-Up Configuration:

- **Using `web.config`:**
    - Insert `<applicationInitialization>` within `<system.webServer>` to specify custom warm-up actions.

```
<system.webServer>
    <applicationInitialization>
        <add initializationPage="/" hostName="[app hostname]" />
        <add initializationPage="/Home/About" hostName="[app hostname]"
/>
    </applicationInitialization>
</system.webServer>
```

- **App Settings for Customization:**
    - **WEBSITE_SWAP_WARMUP_PING_PATH:** Custom URL path to ping for warm-up.

```
# Setting custom warm-up path
WEBSITE_SWAP_WARMUP_PING_PATH=/statuscheck
```

    - **WEBSITE_SWAP_WARMUP_PING_STATUSES:** HTTP status codes considered valid for warm-up.

```
# Setting valid HTTP status codes for warm-up
WEBSITE_SWAP_WARMUP_PING_STATUSES=200,202
```

- **WEBSITE_WARMUP_PATH:** Path to ping on site restarts.

```
# Setting warm-up path for restarts
WEBSITE_WARMUP_PATH=/statuscheck
```

Notes:

- **Linux and Containers:** Auto swap is not supported for web apps on Linux or Web App for Containers.
- **Troubleshooting:** For issues with auto swap or warm-up, refer to deployment slot swap failures documentation.

Conclusion:

Configuring auto swap in Azure App Service is like setting up a conveyor belt in a factory. Once your app's code is pushed, it automatically rolls into production, ensuring zero downtime and cold starts. Custom warm-up settings ensure the machinery (your app) is fully operational before it hits the production line, keeping your customers (users) satisfied with smooth service transitions.

Rollback and Monitoring Slot Swaps in Azure App Service

Rollback Procedure:

- **Immediate Swap Back:** If issues arise post-swap:

```
# Pseudo-code for immediate rollback
swap_slots(sourceSlot="production", targetSlot="staging")
```

This command effectively swaps the slots back to their original configurations.

Monitoring the Swap Operation:

1. **Access the Activity Log:**
    - In the Azure portal, navigate to your app's resource page.
    - From the left pane, select **Activity Log**.
2. **Check Swap Operations:**
    - Look for **Swap Web App Slots** in the activity log entries. This can help you understand:
        - Duration of the swap operation.
        - Any errors or suboperations that occurred.

```
# Pseudo-code to filter for swap operations in logs
filter_activity_log(event='Swap Web App Slots')
```

3. **Detail Examination:**

    - Click on the swap event to expand it, where you can:
        - Review suboperations.
        - Access error details if the swap didn't complete as expected.

Conclusion:

Rollback in Azure App Service is like hitting an "undo" button for your environment swap. If the new production setup doesn't perform as expected, a quick re-swap can bring back the previous version while you investigate. Monitoring through the activity log is akin to reviewing the flight recorder after a flight; it helps you understand what happened during the swap journey, ensuring you can troubleshoot or optimize future swaps.

Routing Traffic in Azure App Service

Traffic Routing Overview:

- By default, all traffic goes to the production slot. However, Azure allows you to route traffic to different slots for testing or phased rollouts.

Steps for Automatic Traffic Routing:

1. **Navigate to Deployment Slots:**
    - Go to your app's **Deployment slots** section in the Azure portal.
2. **Set Traffic Percentage:**
    - In the **Traffic %** column for the desired slot (e.g., staging):
        - Enter a percentage (0-100) of traffic to be routed to this slot.
        - Click **Save**.

```
# Pseudo-code for setting traffic distribution
set_traffic_percentage(slotName="staging", percentage=10)
```

3. **Client Routing:**
    - After saving, the specified percentage of traffic will be randomly routed to the slot.
    - Clients are **pinned** to the slot for their session duration, determined by the `x-ms-routing-name` cookie.

Example of Traffic Distribution:

- If you set **staging** to receive 10% of the traffic:
    - 10% of users will be directed to the staging environment.
    - They will receive this cookie: `x-ms-routing-name=staging`
    - The remaining 90% will go to production with `x-ms-routing-name=self`.

Checking Slot Assignment:

- Use the `x-ms-routing-name` cookie from HTTP headers to confirm which slot your session is using.

Conclusion:

Routing traffic in Azure App Service is like directing a small portion of your audience to a different theater for a sneak preview. You get real user feedback without committing the entire audience. By controlling the traffic percentage, you can test updates safely, and the sticky session ensures users stay with their assigned slot throughout their visit, providing a consistent experience for testing purposes.

Manual Traffic Routing in Azure App Service

Manual Traffic Routing with Query Parameters:

- Azure App Service allows routing traffic manually to different slots using the `x-ms-routing-name` query parameter.

Opting Out of Beta:

- To return users to the production environment from a beta or testing slot:

```
<a href="<webappname>.azurewebsites.net/?x-ms-routing-name=self">Go back to production app</a>
```

- This link sets the `x-ms-routing-name` to **self**, directing the user to the production slot. Subsequent requests include this cookie, keeping the user in production for the session.

Opting Into Beta:

- To direct users to a non-production slot (like staging):

```
<webappname>.azurewebsites.net/?x-ms-routing-name=staging
```

- Replace `staging` with your slot's name.

Advanced Traffic Control:

- When setting a slot's traffic percentage to **0%**:
  - The **0%** value in black indicates that the slot is not automatically routing traffic, but manual access via the query parameter is still possible.
  - This setup is useful for hiding the slot from public traffic while allowing specific users or internal teams to access it for testing.

Key Points:

- **Manual Routing:** Useful for beta testing where users can opt in or out.
- **Cookie Persistence:** Once a user is routed, the session is "pinned" to that slot via a cookie.
- **Slot Visibility:** Setting traffic to 0% hides the slot from random routing but allows manual access.

Conclusion:

Manual traffic routing with Azure App Service is like having a secret handshake. You can give users the option to see behind the curtain (beta environment) or return to the main stage (production) without interrupting the overall show. It's a clever way to get targeted feedback or to conduct controlled testing without throwing open the doors to everyone.

AZ-204: Implement Azure Functions

Content:

- **Introduction to Azure Functions:**
  - Azure Functions are serverless compute services for running small pieces of code, or "functions," in the cloud.
  - They're great for scenarios where you need to execute code in response to events without managing infrastructure.
- **Creating and Deploying Functions:**
  - **Steps to Create a Function:**

```
- Choose runtime (e.g., .NET, Node.js, Python)
- Select a hosting plan
- Write or upload your function code
- Configure triggers and bindings
```

  - **Deployment Options:**
    - Direct publish from Visual Studio or Visual Studio Code
    - Azure DevOps for CI/CD
    - Azure Portal for quick testing and deployment
- **Hosting Options:**
  - **Consumption plan:** Pay only for compute resources when your functions are running.
  - **Premium plan:** Dedicated resources, better performance, and VNET support.
  - **App Service plan:** If you already have apps running on Azure App Service.
- **Triggers and Bindings:**
  - **Triggers:** Events that cause your function to run (e.g., HTTP requests, timers, queues).
  - **Bindings:** Simplify integration with services by defining how data is input or output (e.g., blob storage, Cosmos DB).

Prerequisites:

- **Experience:** At least one year developing scalable solutions.
- **Azure Knowledge:** Basic understanding of Azure services and portal navigation.
- **Recommendation:** If new to Azure, complete AZ-900: Azure Fundamentals first.

Tips for Learning:

- Hands-on labs in the Azure portal can give you practical experience.
- Use Azure Functions Core Tools for local development and debugging.
- Consider scenarios where serverless would benefit your applications, like real-time data processing or IoT data handling.

Remember, Azure Functions are like the Swiss Army Knife of cloud services; versatile, handy, and they might just save the day when you're in a bind. Happy coding!

Introduction to Azure Functions:

Key Concepts:

- **Azure Functions:**
  - Azure Functions enable you to build serverless applications. Essentially, you're writing small pieces of code (functions) that execute in the cloud in response to various events without needing to manage server infrastructure.

Learning Outcomes:

- **Comparison with Other Azure Services:**
  - **Azure Functions** vs **Azure Logic Apps** vs **WebJobs:**
    - **Azure Functions:** Best for event-driven, scalable compute scenarios. You write the code in response to triggers.
    - **Azure Logic Apps:** Workflow automation service, visual designer for SaaS and enterprise integration.
    - **WebJobs:** Runs background tasks within an App Service web app, but less scalable in terms of execution frequency and event types.
- **Hosting Options in Azure Functions:**
  - **Consumption Plan:** Ideal for scenarios where you need to run code sporadically. You're charged based on resource consumption.
  - **Premium Plan:** Provides pre-warmed instances, longer execution time, and VNET integration for better performance and security.
  - **Dedicated (App Service) Plan:** Best if you're already using Azure App Service for other applications, allows for constant warm-up.
- **Scalability:**
  - Azure Functions automatically scale based on the number of incoming events or triggers, up to the limits of your hosting plan. This means your application can handle load spikes without manual intervention.

Practical Considerations:

- Choose Azure Functions when:
  - You need to run code in response to specific events (like HTTP requests, timer triggers, etc.)
  - You want to pay only for the compute resources you consume.
  - Your workload is highly variable or unpredictable.

- **Example Use Case:**

```
// A basic Azure Function that responds to an HTTP GET request
[FunctionName("GetUserProfile")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", Route =
"user/{userId}")] HttpRequest req,
    string userId,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    // Your code to fetch user profile goes here
    var profile = await FetchUserProfileAsync(userId);

    return new OkObjectResult(profile);
}
```

Remember, with Azure Functions, you're not just coding; you're thinking like a cloud-native developer, letting the cloud take care of the "server" part so you can focus on the "less" part.

Discover Azure Functions:

Module Overview:

- **Status:** Completed
- **XP:** 100
- **Duration:** 3 minutes

Key Points:

- **Serverless Architecture:**
    - Azure Functions embodies the serverless computing model where you focus on writing code for business logic while Azure handles the infrastructure.
- **Use Cases for Azure Functions:**
    - **Web APIs:** Serve as endpoints for APIs that can scale automatically.
    - **Database Change Response:** Trigger functions when data changes occur.
    - **IoT Data Processing:** Handle streams of data from IoT devices.
    - **Message Queue Management:** Process messages from queues like Azure Service Bus.
- **Core Components:**
    - **Triggers:**
        - Trigger types include:
            - HTTP trigger (REST APIs)
            - Timer trigger (scheduled tasks)
            - Queue trigger (message processing from Azure Queue Storage)

        - Blob storage trigger (file uploads)

```
// Example of an HTTP trigger
[FunctionName("HttpExample")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route =
null)] HttpRequest req,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a
request.");

    string name = req.Query["name"];

    string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
        : new BadRequestObjectResult("Please pass a name on the query
string or in the request body");
}
```

- **Bindings:**
    - Simplify coding by automatically connecting to input or output data sources.
    - Examples include:
        - **Input Binding:** Fetch data from a blob storage when the function runs.
        - **Output Binding:** Send results to a database or queue after processing.

```
// Example of an output binding to send a message to a queue
[FunctionName("QueueOutputExample")]
public static void Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route =
null)] HttpRequest req,
    [Queue("outqueue"), StorageAccount("AzureWebJobsStorage")] out
string myQueueItem,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a
request.");

    string name = req.Query["name"];
    string requestBody = new StreamReader(req.Body).ReadToEnd();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;
```

```
    myQueueItem = $"New Queue Item: {name}";
}
```

- **Automation and Integration:**
    - Azure Functions can be part of a larger ecosystem of integration tools in Azure, working alongside Azure Logic Apps, Azure Event Grid, and Azure Service Bus to automate processes and integrate systems.

Remember, Azure Functions are your ticket to a world where you don't need to manage servers - just the logic that matters. It's like having a team of invisible IT staff that you never have to meet!

Comparing **Azure Functions** and **Azure Logic Apps** as well as a comparison with **WebJobs**:

Azure Functions vs Azure Logic Apps:

- **Development Approach:**
    - **Azure Functions:** Code-first (imperative)
        - You write the code for each step of the orchestration.
        - Durable Functions can be used for stateful workflows:

```
// Example of a Durable Function in Azure Functions
[FunctionName("OrchestratorFunction")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("Activity1", "Hello");
    await context.CallActivityAsync("Activity2", "World");
}
```

    - **Logic Apps:** Designer-first (declarative)
        - Use a visual designer or edit configuration files.
- **Connectivity:**
    - **Azure Functions:** Limited built-in bindings, but you can code for custom ones.
    - **Logic Apps:** Extensive library of connectors including Enterprise Integration Pack.
- **Actions:**
    - **Azure Functions:** Each action is a function you must code.
    - **Logic Apps:** Predefined actions which can be configured without coding.
- **Monitoring:**
    - **Azure Functions:** Uses Azure Application Insights.
    - **Logic Apps:** Monitored via the Azure portal and Azure Monitor logs.
- **Management:**

    - **Azure Functions:** REST API, Visual Studio.
    - **Logic Apps:** Azure portal, REST API, PowerShell, Visual Studio.

- **Execution Context:**

    - Both can run in Azure, but Logic Apps can also run locally or on-premises.

Azure Functions vs WebJobs with WebJobs SDK:

- **Serverless and Scaling:**

    - **Azure Functions:** True serverless with automatic scaling.
    - **WebJobs:** Not serverless in nature, does not scale automatically.

- **Development Environment:**

    - **Azure Functions:** Can develop and test in the browser.
    - **WebJobs:** Requires Visual Studio or similar IDE for development.

- **Pricing:**

    - **Azure Functions:** Pay-per-use, more cost-effective for intermittent workloads.
    - **WebJobs:** Part of the App Service plan, which might not be as cost-effective for sporadic workloads.

- **Integration with Logic Apps:**

    - **Azure Functions:** Can be used as actions within Logic Apps workflows.
    - **WebJobs:** Not directly integrated with Logic Apps.

- **Trigger Events:**

    - Both support a variety of triggers, but Azure Functions offers more flexibility and includes newer Azure services like Event Grid.

Conclusion:

- **Azure Functions** is generally the recommended service for new projects due to:
    - Better developer productivity
    - More programming language support
    - Greater flexibility in Azure service integration
    - More attractive pricing model for many scenarios

For existing applications with a heavy WebJobs investment, migration to Azure Functions might be considered, but for new development, Azure Functions typically provides a more modern and efficient approach to serverless compute.

Compare Azure Functions Hosting Options:

Hosting Options for Azure Functions:

1. **Consumption Plan:**

    - **Service:** Azure Functions

- **Availability:** Generally Available (GA)
- **Container Support:** None
- **Characteristics:**
  - Serverless experience with automatic scaling based on the number of events.
  - You're charged only when your code runs, which makes it ideal for sporadic workloads.
  - No support for pre-warmed instances or VNET integration.

2. **Flex Consumption Plan:**

- **Service:** Azure Functions
- **Availability:** Preview
- **Container Support:** None
- **Characteristics:**
  - Similar to Consumption Plan but with a preview feature set, potentially including faster scaling or other enhancements.

3. **Premium Plan:**

- **Service:** Azure Functions
- **Availability:** Generally Available (GA)
- **Container Support:** Linux
- **Characteristics:**
  - Offers pre-warmed instances to reduce cold start times.
  - Provides enhanced performance and more consistent execution.
  - Includes VNET integration for secure networking.
  - Priced based on vCPU and memory usage, not just execution time.

4. **Dedicated Plan:**

- **Service:** Azure Functions
- **Availability:** Generally Available (GA)
- **Container Support:** Linux
- **Characteristics:**
  - Runs on an App Service plan, allowing you to share resources with other applications.
  - Provides predictable pricing and scaling behavior.
  - Supports Always On for continuous instances.
  - Ideal when you have existing App Service resources to leverage.

5. **Container Apps:**

- **Service:** Azure Container Apps
- **Availability:** Generally Available (GA)
- **Container Support:** Linux
- **Characteristics:**
  - Allows running functions in any container, giving you full control over the environment.
  - Supports microservices architecture alongside functions.
  - Provides scaling based on events and includes features like revisions and deployments.

Impact of Hosting Options:

- **Scaling:**
    - Consumption and Flex plans scale automatically based on demand.
    - Premium and Dedicated plans offer manual scaling or automatic scaling within the App Service Plan limits.
- **Resource Availability:**
    - Consumption plans have dynamic allocation.
    - Premium and Dedicated plans have dedicated resources based on your plan selection.
- **Advanced Functionality:**
    - Premium and Dedicated plans support VNET integration for enhanced security.
    - Container Apps provide the most flexibility with container orchestration capabilities.
- **Cost Implications:**
    - **Consumption/Flex:** Pay only for the time your functions are running.
    - **Premium/Dedicated:** Pay for the resources you reserve, whether or not they are in use.
    - **Container Apps:** Billing includes the cost of containers, which can be more predictable for steady workloads.

Choosing the Right Plan:

- Consider your workload's nature:
    - **Sporadic and Event-driven:** Consumption Plan.
    - **High-performance or Networking:** Premium Plan.
    - **Resource Sharing with Web Apps:** Dedicated Plan.
    - **Complete Container Control:** Container Apps.

Each plan has its niche, and your choice should align with your application's needs, expected traffic patterns, and cost considerations. Remember, in Azure, the cloud is your playground - pick the hosting option that lets your functions play best!

Overview of Azure Functions Hosting Plans:

Consumption Plan:

- **Default Hosting Option**
- **Cost:** Pay-for-use based on the compute resources consumed during function executions.
- **Scaling:**
    - **Dynamic:** Instances are added or removed automatically according to the number of incoming events.
    - **Benefits:**
        - **No Cold Starts:** Functions scale from zero to handle new events.
        - **Cost Efficiency:** Ideal for applications with intermittent or unpredictable loads.

Flex Consumption Plan:

- **High Scalability**
- **Features:**

    - **Compute Options:** Offers different compute configurations.
    - **Virtual Networking:** Includes networking features like VNET integration.
    - **Concurrency Control:** Scales based on configured per instance concurrency along with event load.
    - **Pre-provisioned Instances:** Can set a number of instances that are always ready, reducing cold start latency.
    - **Benefits:**
        - **Cost Effective with Performance:** Similar pay-as-you-go model but with more control over scaling behavior.
        - **Reduced Latency:** Pre-warmed instances help with responsiveness.

Premium Plan:

- **Advanced Features**
- **Scaling & Performance:**
    - **Automatic:** Scales based on demand.
    - **Prewarmed Workers:** Instances are kept warm to reduce latency after idle periods.
    - **Powerful Instances:** Access to higher CPU and memory options.
    - **Virtual Network Connectivity:** Allows secure communication with other resources in a VNET.
    - **Longer Execution Time:** Supports applications that need to run for extended periods.
    - **Custom Linux Images:** Can use custom images for more control over the runtime environment.
- **When to Use:**
    - **Continuous Operations:** For function apps that run most of the time or need to be always on.
    - **Multi-App Plans:** When multiple function apps need to share resources with event-driven scaling.
    - **High Execution Count:** If you're facing high execution costs but low resource consumption on the Consumption Plan.
    - **Resource Intensive:** When you need more CPU or memory than is available in the Consumption Plan.
    - **Network Requirements:** For apps requiring secure network access.

Syntax Example for Azure Functions (Simple HTTP Trigger in C# for any plan):

```
[FunctionName("HttpTriggerFunction")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
HttpRequest req,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    return name != null
        ? (ActionResult)new OkObjectResult($"Hello, {name}")
```

```
            : new BadRequestObjectResult("Please pass a name on the query string or in
the request body");
}
```

Remember, choosing the right plan is like picking the right spaceship for your interstellar journey - you need to consider the load you're carrying, the conditions you'll face, and how much fuel you're willing to spend!

Dedicated Plan and **Container Apps** along with the **Function App Timeout Duration**:

Dedicated Plan:

- **Hosting:** Functions run within an App Service plan.
- **Billing:** Regular App Service plan rates.
- **Use Cases:**
  - **Predictable Billing:** When you need a consistent billing model.
  - **Manual Scaling:** For scenarios where you want to manually control scaling.
  - **Resource Sharing:** If you're running multiple web and function apps on the same plan.
  - **Large Compute:** Access to larger compute sizes.
  - **Isolation:** When you need full compute isolation and secure network access via an App Service Environment (ASE).
  - **High Memory/Scale:** Suited for applications with high memory usage or those requiring high scale.

Container Apps:

- **Hosting:** Fully managed environment for containerized function apps.
- **Benefits:**
  - **Simplified Operations:** Avoid managing Kubernetes clusters.
  - **Microservices:** Functions can run alongside other cloud-native services.
  - **Custom Libraries:** Package custom libraries with your function code.
  - **Legacy Migration:** Migrate from on-premises or legacy to cloud-native containerized workloads.
  - **Dedicated CPU:** When high-end CPU resources are needed for function execution.

Function App Timeout Duration:

- **host.json Configuration:** The `functionTimeout` property sets the execution timeout for functions.
- **Behavior:**
  - Functions must respond within the timeout duration after being triggered.

Timeout Values by Plan:

Plan	Default (minutes)	Maximum (minutes)
Consumption Plan	5	10
Flex Consumption Plan	30	Unlimited
Premium Plan	30	Unlimited
Dedicated Plan	30	Unlimited

Plan	Default (minutes)	Maximum (minutes)
Container Apps	30	Unlimited

- **HTTP Trigger Limit:** Even if the function app timeout is set to unlimited, HTTP-triggered functions have a hard limit of 230 seconds to respond.
- **Version 1.x Default:** No timeout limit for version 1.x of the Functions runtime.
- **Guaranteed Execution:** Up to 60 minutes for Premium, Dedicated, and Container Apps, with caveats for OS updates, patching, or scale-in.
- **Flex Consumption Caveat:** No enforced limit, but termination can occur due to platform actions.
- **Container Apps with Zero Replicas:** Default timeout varies based on triggers when minimum replicas are set to zero.

When configuring your function app, remember that setting these timeouts is like setting the timer on your space suit's air supply - too short and you might run into issues mid-task, too long and you might end up paying for air you don't use. Balance is key!

Scale Azure Functions:

Scaling Behaviors by Hosting Plan:

Plan	Scale Out Behavior	Max # Instances
Consumption Plan	- Event-driven scaling - Automatically scales out during high load. - Scales based on incoming trigger events.	- **Windows:** 200 - **Linux:** 100[^1]
Flex Consumption Plan	- Per-function scaling - Deterministic scaling based on individual function triggers. - Scales by adding instances for each function.	- Limited by total memory usage across a region.
Premium Plan	- Event-driven - Automatically scales based on function triggers.	- **Windows:** 100 - **Linux:** 20-100[^2]
Dedicated Plan	- Manual or Autoscale - Scaling is controlled by user-defined rules or manually.	- **Standard:** 10-30 - **ASE:** 100
Container Apps	- Event-driven - Automatically scales by adding more instances based on event triggers.	- **Range:** 10-300[^4]

Notes:

- **Consumption Plan:**
    - During scale-out, there's a limit of 500 instances per subscription per hour for Linux apps.
- **Premium Plan:**
    - The ability to scale to 100 instances for Linux apps is region-dependent.

- **Dedicated Plan:**
  - Specific limits depend on the App Service plan options. For more details, refer to App Service plan limits.
- **Container Apps:**
  - You can configure the maximum number of replicas, which is respected provided there are enough cores available in the quota.

Example Code for Scaling Configuration in `host.json` (if applicable):

```
{
  "version": "2.0",
  "extensions": {
    "http": {
      "routePrefix": "api",
      "maxOutstandingRequests": 200, // Example configuration for HTTP triggers
      "maxConcurrentRequests": 35
    }
  },
  "functionTimeout": "00:05:00", // Example timeout setting
  "healthMonitor": {
    "enabled": true,
    "healthCheckInterval": "00:00:10",
    "healthCheckWindow": "00:02:00",
    "healthCheckThreshold": 6,
    "counterThreshold": 0.80
  }
}
```

This configuration snippet shows how you might set up scaling-related parameters in Azure Functions, focusing on HTTP triggers and health monitoring.

Remember, scaling in Azure Functions is like managing a fleet of interstellar vessels; you want each ship (instance) to be ready to handle the load, and sometimes you need to call in reinforcements or send some ships home when the job's done!

Develop Azure Functions:

Unit Topics (to be covered):

1. **Introduction to Azure Functions:**
   - Understanding the core concepts of serverless computing with Azure Functions.
   - The advantages and use cases for using Azure Functions.
2. **Setting up the Development Environment:**
   - Tools and SDKs required for Azure Functions development.
   - Configuring Visual Studio, Visual Studio Code, or Azure Functions Core Tools.

3. **Creating Your First Function:**
    - Steps to create a simple HTTP triggered function.
    - Example code for an HTTP trigger function:

```
using System.Net;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
HttpRequestData req,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    string responseMessage = string.IsNullOrEmpty(name)
        ? "This HTTP triggered function executed successfully. Pass a name in the
query string or in the request body for a personalized response."
        : $"Hello, {name}. This HTTP triggered function executed successfully.";

    return req.CreateResponse(HttpStatusCode.OK, responseMessage);
}
```

4. **Function Bindings and Triggers:**
    - Exploring different types of triggers (e.g., Blob, Queue, Timer, Event Hub).
    - How to configure input and output bindings.

5. **Local Testing and Debugging:**
    - Running and testing functions locally.
    - Using Azure Storage Emulator or local resources for testing.

6. **Deployment Strategies:**
    - Deploying functions from local to Azure.
    - Continuous Integration/Continuous Deployment (CI/CD) with Azure DevOps or GitHub Actions.

7. **Monitoring and Scaling:**
    - Setting up Application Insights for monitoring.
    - Understanding how Azure Functions scale automatically in different plans.

Tips for Efficient Development:

- **Use Azure Functions Core Tools** for a streamlined local development experience.
- **Leverage the Azure portal** for quick prototyping and testing.
- **Understand the asynchronous nature** of function execution for better design.
- **Implement proper error handling** to manage unexpected scenarios.

Remember, while developing Azure Functions, you're not just coding; you're orchestrating a symphony of cloud events. Keep your functions light, your dependencies lean, and your code clean!

Introduction to Developing Azure Functions:

Key Concepts of Azure Functions:

1. **Function App:**
    - A function app is the container for your functions in Azure. It defines the execution context for a set of functions, including runtime settings.
2. **Function:**
    - A function is the unit of work, the actual piece of code that runs when triggered. Functions can be stateless or stateful (using Durable Functions for orchestration).
3. **Trigger:**
    - A trigger defines how a function is invoked. Each function must have exactly one trigger.
    - Common triggers include:
        - **HTTP**: Invoked via HTTP requests.
        - **Timer**: Scheduled execution.
        - **Blob Storage**: Triggered when files are uploaded or changed.
4. **Bindings:**
    - Bindings are how functions connect to and interact with data sources, services, and other resources.
    - There are **Input** bindings (like reading from a queue) and **Output** bindings (like writing to a database).
    - Examples:
        - **Input Binding:** Automatically fetch data from Cosmos DB when the function runs.
        - **Output Binding:** Send a message to a Service Bus queue after processing.
5. **Function.json:**
    - Configuration file for each function, defining triggers, bindings, and other settings.

Learning Outcomes:

- **Explain the Key Components:**
    - Understanding how function apps, functions, triggers, and bindings work together.
- **Create Triggers and Bindings:**

    - You'll learn how to set up triggers to define when functions run, and bindings to handle input and output data.

- **Connect to Azure Services:**
    - Functions can integrate seamlessly with other Azure services for both triggers and bindings.
- **Create Functions Using Visual Studio Code & Azure Functions Core Tools:**
    - Visual Studio Code is a lightweight, powerful tool for developing Azure Functions.
    - **Azure Functions Core Tools** allows for local debugging, testing, and deployment.

Practical Example:

```
{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ],
  "scriptFile": "../dist/FunctionApp/index.js"
}
```

This `function.json` configuration describes an HTTP trigger function. It awaits HTTP GET or POST requests and responds via an HTTP output binding.

Note: When developing, remember that Azure Functions are like the Swiss Army Knife of Azure services; they can slice through data, connect disparate services, and scale like a pro, all without you worrying about the infrastructure.

Explore Azure Functions Development:

Function App:

- **Definition:** A function app is the deployment and management unit in Azure for your functions.
- **Composition:** It contains one or more functions that are managed, deployed, and scaled as a group.
- **Shared Resources:** Functions within an app share:
    - The same pricing plan
    - Deployment method

    - Runtime version
- **Version Consideration:**
    - **Functions 2.x:** All functions within a function app must use the same programming language.
    - **Previous Versions:** Did not require all functions to be in the same language.

Local Development:

- **Advantages:**
    - Use your preferred code editor and development tools.
    - Easily test and debug functions on your local machine using the full Functions runtime.
    - Connect to live Azure services for testing purposes.
- **Development Environments:**
    - Choice depends on your language and tool preferences:
        - **Visual Studio:** For .NET developers.
        - **Visual Studio Code:** Cross-platform and supports various languages.
        - **Azure Functions Core Tools:** Command-line interface for all languages, good for automation and CI/CD.
- **Important Note:**
    - Due to portal limitations, local development is recommended. Editing function code in the Azure portal has limitations, so:
        - Develop functions locally.
        - Publish them to a function app in Azure.

Local Development Configuration:

```
// local.settings.json
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet"
  }
}
```

- The `local.settings.json` file contains settings used during local development. The `AzureWebJobsStorage` setting here uses the local Azure Storage Emulator, and `FUNCTIONS_WORKER_RUNTIME` specifies the runtime environment.

Guideline for Local Development:

- Use local development for building and testing your functions.
- Ensure your local setup mirrors the Azure environment as closely as possible to avoid discrepancies post-deployment.
- Leverage local debugging tools for efficient troubleshooting before pushing to Azure.

Remember, with Azure Functions, you're not just writing code; you're crafting small, powerful pieces of cloud-native logic that can be tested at home and then sent out into the cosmos of Azure to do their job.

Local Project Files in Azure Functions:

Essential Local Project Files:

- **host.json:**
  - **Purpose:** This file contains global configuration options for all functions within a function app instance.
  - **Environment:**
    - **Azure:** Configuration is managed via application settings.
    - **Local:** Configuration is managed in the `local.settings.json` file.
  - **Bindings:** Configuration for bindings in `host.json` applies to all functions in the app.
- **local.settings.json:**
  - **Purpose:** Stores app settings and local development tools settings.
  - **Usage:** Only used when running the project locally.
  - **Security Consideration:**

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet",
    "MySecretSetting": "secretValue"
  },
  "ConnectionStrings": {
    "SQLDBConnectionString":
"Server=tcp:yourserver.database.windows.net,1433;Initial
Catalog=yourdatabase;Persist Security Info=False;User
ID=youruserid;Password=yourpassword;MultipleActiveResultSets=False;Encr
ypt=True;TrustServerCertificate=False;Connection Timeout=30;"
  }
}
```

    - **Note:** Secrets like connection strings should be kept out of source control. Consider encrypting this file or using a secrets management system.

Synchronizing Settings:

- **Local to Azure:** After local development, ensure that all necessary settings are replicated in your Azure function app's application settings before deployment.
- **Azure to Local:** You can download your Azure function app's current settings to synchronize with your local development environment.

Best Practices:

- **Never commit `local.settings.json`:** This file can contain sensitive information. Use `.gitignore` or similar to exclude it from version control.
- **Use Environment Variables:** In Azure, use app settings for configuration which can be accessed as environment variables.
- **Secrets Management:** Consider using Azure Key Vault or another secrets management service for handling sensitive information.

Developers should treat local settings files with care, recognizing that while they're vital for local development, they pose a security risk if accidentally shared or committed to source control.

Create Triggers and Bindings:

Triggers and Bindings:

- **Trigger:**
  - **Defines:** How a function is invoked.
  - **Requirement:** Must have exactly one per function.
  - **Data:** Associated data provided as the payload.
- **Bindings:**
  - **Purpose:** Declaratively connect to resources for input or output.
  - **Types:**
    - **Input:** Data provided as function parameters.
    - **Output:** Data sent via function return value or out parameters.
  - **Flexibility:** Can have multiple bindings or none at all.
  - **Benefits:**
    - Reduces hardcoding service access.
    - Simplifies integration with other Azure services.

Configuring Triggers and Bindings:

- **C# Class Library:**
  - Use **C# attributes** to decorate methods and parameters.

```
// Example of HTTP Trigger in C#
public static class Function1
{
    [FunctionName("HttpTriggerCSharp")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
HttpRequest req,
        ILogger log)
    {
        // Function logic here
    }
}
```

- **Java:**

    - Use **Java annotations** for configuration.

```
// Example of Queue Trigger in Java
@FunctionName("QueueTriggerJava")
public void run(
    @QueueTrigger(name = "message", queueName = "myqueue-items", connection =
"AzureWebJobsStorage") String message,
    final ExecutionContext context
) {
    context.getLogger().info("Java Queue trigger function processed a message: " +
message);
}
```

- **JavaScript/PowerShell/Python/TypeScript:**
    - **function.json** schema is updated.

```
{
    "bindings": [
        {
            "name": "req",
            "type": "httpTrigger",
            "direction": "in",
            "dataType": "binary",
            "methods": ["get", "post"]
        },
        {
            "name": "response",
            "type": "http",
            "direction": "out"
        }
    ]
}
```

- **Portal Configuration:**
    - For dynamically typed languages, set `dataType` in `function.json` to handle different data formats (e.g., `binary`, `stream`, `string`).

Binding Direction:

- **Triggers:** Always `in`.
- **Bindings:**
    - `in` for input.
    - `out` for output.
    - `inout` for special bidirectional bindings, limited to Advanced editor in the Azure portal.

C# and Java Notes:

- **Parameter Type:** Defines the data type for input in strongly typed languages.

- **Portal Limitations:** Functions defined with attributes can't be edited in the portal as they don't use `function.json`.

Remember, configuring triggers and bindings is like setting up the plumbing for your function; you're telling Azure how to funnel data in and out, ensuring your function can do its job without drowning in boilerplate code.

Azure Functions Trigger and Binding Example:

Scenario Description:

- Trigger a function when a new message appears in Azure Queue storage.
- Write the message content as a new row to Azure Table storage.

function.json Configuration:

```
{
  "disabled": false,
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "myQueueItem",
      "queueName": "myqueue-items",
      "connection": "MyStorageConnectionAppSetting"
    },
    {
      "tableName": "Person",
      "connection": "MyStorageConnectionAppSetting",
      "name": "tableBinding",
      "type": "table",
      "direction": "out"
    }
  ]
}
```

- **Queue Trigger Binding:**
  - **Type:** `queueTrigger`
  - **Direction:** `in` (input)
  - **Name:** `myQueueItem` (function parameter)
  - **QueueName:** `myqueue-items` (name of the queue to monitor)
  - **Connection:** Points to an app setting for the storage account connection string.
- **Table Output Binding:**
  - **Type:** `table`
  - **Direction:** `out` (output)
  - **Name:** `tableBinding` (used to identify the output)
  - **TableName:** `Person` (the table where data will be stored)

- **Connection:** Points to the same app setting as the trigger for storage account access.

C# Function Example:

```
public static class QueueTriggerTableOutput
{
    [FunctionName("QueueTriggerTableOutput")]
    [return: Table("outTable", Connection = "MY_TABLE_STORAGE_ACCT_APP_SETTING")]
    public static Person Run(
        [QueueTrigger("myqueue-items", Connection =
"MY_STORAGE_ACCT_APP_SETTING")] JObject order,
        ILogger log)
    {
        return new Person() {
            PartitionKey = "Orders",
            RowKey = Guid.NewGuid().ToString(),
            Name = order["Name"].ToString(),
            MobileNumber = order["MobileNumber"].ToString()
        };
    }
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

- **FunctionName:** Names the function for easy reference and deployment.
- **QueueTrigger Attribute:** Specifies the queue name and connection string app setting for the trigger.
- **Table Attribute:** On the return value, specifies the output table and its connection setting.
- **Person Class:** Represents the data structure for the table entry.

Key Points:

- The `QueueTrigger` attribute on the `order` parameter indicates that the function runs when a new message is added to the queue.
- The function processes the queue message data and constructs a `Person` object.
- The return of the `Person` object is bound to the Table Storage output, automatically inserting this data as a new row in the 'outTable'.

This setup demonstrates how Azure Functions can automate data processing workflows with minimal code, leveraging triggers and bindings to interact with Azure services.

Connect Functions to Azure Services:

Secure Connection Practices:

- **Application Settings:**

    - Use Azure App Service's application settings to store sensitive information like connection strings.
    - These settings are encrypted and treated as environment variables at runtime.
    - For bindings, reference these settings rather than hardcoding the actual connection details.

Using Environment Variables:

- **Local Development:** Use `local.settings.json` for local environment variables.
- **Azure Deployment:** Use Azure Application Settings for production environment variables.

Identity-Based Connections:

- **Managed Identity:** Preferred over secrets where supported.
    - **Default:** Uses system-assigned managed identity.
    - **User-Assigned:** Available with `credential` and `clientID` properties.
    - **Limitation:** Can't configure using resource ID for user-assigned identities.
- **Local Development:** Uses the developer's identity by default, which can be customized.

Example of configuring managed identity for Azure Key Vault:

```
{
  "bindings": [
    {
      "type": "keyVault",
      "direction": "in",
      "name": "mySecret",
      "keyVaultName": "mykeyvault",
      "secretName": "MySecret",
      "credential": "ManagedIdentity"
    }
  ]
}
```

Permissions Management:

- **Role-Based Access Control (RBAC):**
    - Assign necessary roles to the managed identity via Azure RBAC.
    - Roles should match the minimum requirements for the function's operations.
- **Access Policies:**
    - For services like Azure Key Vault, specify the identity in an access policy.

```
# Example of granting permissions via Azure CLI
az role assignment create --assignee <managed-identity-principal-id> --role <role-name> --scope /subscriptions/<subscription-id>/resourceGroups/<resource-group>/providers/Microsoft.Storage/storageAccounts/<storage-account-name>
```

Important Considerations:

- **Azure Files Limitation:**
  - In Consumption or Elastic Premium plans, storage account access for Azure Files uses predefined connection settings, which do not support managed identity.
- **Least Privilege Principle:**
  - Grant only the permissions required for the function to operate, avoiding over-provisioning access rights.

When setting up connections, think of it like giving out keys to your interstellar spaceship. You wouldn't give every crew member the key to the command deck; you'd only give them access to their specific stations. Similarly, in Azure Functions, manage identities and permissions with care to keep your cloud operations secure and efficient.

AZ-204: Develop solutions that use Blob storage

Overview:

- **Objective:** Learn to manage Azure Blob storage resources, lifecycle management, and interaction with containers and items using Azure Blob storage client library V12 for .NET.

Prerequisites:

- **Experience:** At least 1 year developing scalable solutions across all software development phases.
- **Knowledge:** Basic understanding of Azure services, cloud concepts, and navigation within the Azure portal.
- **Recommendation:** Complete AZ-900: Azure Fundamentals if new to Azure or cloud computing.

Key Learning Points:

1. **Create Azure Blob Storage Resources:**
   - **Code Example for Creating a Blob Container:**

```
BlobServiceClient blobServiceClient = new
BlobServiceClient(connectionString);
await blobServiceClient.CreateBlobContainerAsync(containerName);
```

2. **Manage Blob Storage Lifecycle:**
   - Understand lifecycle policies to automatically transition blobs to cooler storage tiers or delete them.
3. **Work with Containers and Items:**
   - **Uploading a Blob:**

```
BlobContainerClient containerClient =
blobServiceClient.GetBlobContainerClient(containerName);
BlobClient blobClient = containerClient.GetBlobClient(blobName);
await blobClient.UploadAsync(fileStream);
```

- **Downloading a Blob:**

```
BlobDownloadInfo download = await blobClient.DownloadAsync();
using (FileStream fileStream = File.OpenWrite(localFilePath))
{
    await download.Content.CopyToAsync(fileStream);
}
```

Notes:

- Ensure you handle exceptions and implement proper error logging in production code.
- Consider using asynchronous methods for better performance in blob operations.
- Remember to manage security, like using least privilege access for your blob storage accounts.

These notes provide a quick reference for developing with Azure Blob Storage, focusing on practical code examples and key concepts for lifecycle management and interaction with Blob storage services.

Introduction to Azure Blob Storage

Overview:

Azure Blob Storage is a scalable cloud storage solution by Microsoft for handling vast amounts of unstructured data. It's designed for:

- Serving images or documents directly to browsers
- Storing files for distributed access
- Streaming video and audio content

Key Topics Covered:

1. **Understanding Azure Blob Storage:**
   - **Features:**
     - Scalability: Store hundreds of terabytes or even petabytes of data.
     - Durability: Geo-redundant storage options for data protection.
     - Flexibility: Multiple data access tiers for cost optimization.
   - **Types of Storage Accounts:**
     - General-purpose v2: Most versatile, supports all storage services.

        - Blob Storage: Optimized for storing unstructured data as blobs.
        - General-purpose v1: Legacy, not recommended for new applications.

    - **Access Tiers:**

        - Hot: Frequently accessed data, higher storage costs but lower access costs.
        - Cool: Infrequently accessed data, lower storage costs but higher access costs.
        - Archive: Rarely accessed data, lowest storage costs, highest retrieval costs.

2. **Storage Accounts, Containers, and Blobs:**

    - **Storage Accounts** are like a parent namespace in Azure that can contain multiple containers.
    - **Containers** are used to organize blobs, similar to directories in a filesystem.
    - **Blobs** are the actual files or data stored within these containers.

    ```
    Storage Account
    ├── Container
    │   ├── Blob (File1)
    │   └── Blob (File2)
    └── Container 2
        └── Blob (File3)
    ```

3. **Azure Storage Security and Encryption Features:**

    - **Authentication:** Shared Key, Azure AD, and SAS (Shared Access Signature) for secure access.
    - **Encryption:** Data at rest encryption using Microsoft-managed keys or customer-managed keys.
    - **Network Security:** Private Endpoints, Firewalls, and VNet service endpoints for secure network access.

    Code Example for Creating a Container with .NET:

    ```
    BlobServiceClient blobServiceClient = new
    BlobServiceClient(connectionString);
    BlobContainerClient containerClient = await
    blobServiceClient.CreateBlobContainerAsync("mycontainer");
    ```

    Note: Always ensure you follow the principle of least privilege when setting permissions for blobs and containers.

These notes provide a quick overview of Azure Blob Storage's capabilities, structure, and security features, with a brief example of how to interact with Blob Storage programmatically.

Explore Azure Blob Storage

Overview:

Azure Blob Storage is a cloud-based object storage service ideal for managing large volumes of unstructured data like text, binary data, documents, or media files.

Primary Use Cases:

- **Content Delivery:** Direct serving of images or documents to web browsers.
- **File Storage:** Storing files accessible by multiple users or applications.
- **Media Streaming:** Efficient streaming of video and audio content.
- **Logging:** Writing log files for applications.
- **Backup and Recovery:** Data archiving, disaster recovery, and backup solutions.
- **Data Analysis:** Storing data for analysis by local or cloud-based services.

Accessing Blob Storage:

- **Protocols:** HTTP/HTTPS
- **Methods:**
  - Azure Storage REST API
  - Azure PowerShell
  - Azure CLI
  - Azure Storage client libraries (e.g., for .NET, Python, Java, etc.)

Code Example for Uploading a Blob using .NET:

```
BlobServiceClient blobServiceClient = new BlobServiceClient(connectionString);
BlobContainerClient containerClient =
blobServiceClient.GetBlobContainerClient("mycontainer");
BlobClient blobClient = containerClient.GetBlobClient("myblob");

using (FileStream uploadFileStream = File.OpenRead("myfile.txt"))
{
    await blobClient.UploadAsync(uploadFileStream, true);
    Console.WriteLine("File uploaded successfully.");
}
```

Code Example for Listing Blobs in a Container:

```
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine(blobItem.Name);
}
```

Azure Storage Account:

- **Purpose:** Acts as the top-level organizational unit and provides a unique namespace for your storage resources.
- **Global Accessibility:** Data in your storage account can be accessed worldwide via HTTP or HTTPS.

These notes summarize the functionalities and access methods of Azure Blob Storage, along with practical examples for interacting with it, showcasing its utility for developers and system administrators managing cloud storage.

Types of Storage Accounts

Azure Storage offers two main performance levels:

- **Standard:** General-purpose v2 account, suitable for most scenarios.
- **Premium:** Offers higher performance using SSDs, with three specific types:

Type of storage account	Supported storage services	Redundancy options	Usage
Standard general-purpose v2	Blob, Queue, Table, Azure Files	LRS, GRS, RA-GRS, ZRS, GZRS, RA-GZRS	Default for most scenarios. Supports NFS in Azure Files with premium type.
Premium block blobs	Blob Storage	LRS, ZRS	For high transaction rates, smaller objects, or low latency.
Premium file shares	Azure Files	LRS, ZRS	For enterprise or high-performance file share applications.
Premium page blobs	Page blobs	LRS, ZRS	Exclusively for page blobs.

Access Tiers for Block Blob Data

Azure Storage provides different access tiers for efficient cost management:

- **Hot Access Tier:**
    - Optimized for frequent access.
    - Highest storage cost, lowest access cost.
    - Default tier for new accounts.
- **Cool Access Tier:**
    - For data accessed infrequently, stored for at least 30 days.
    - Lower storage costs, higher access costs than Hot.
- **Cold Access Tier:**
    - For data accessed even less frequently, stored for at least 90 days.
    - Lower storage costs than Cool, higher access costs.
- **Archive Tier:**

- For data with retrieval latency of several hours, stored for at least 180 days.
- Most cost-effective for storage, but expensive for retrieval compared to others.

Code Example for Changing Blob's Access Tier in .NET:

```
BlobClient blobClient = new BlobClient(connectionString, containerName, blobName);
await blobClient.SetAccessTierAsync(AccessTier.Hot);
Console.WriteLine("Blob access tier changed to Hot.");
```

Note:

- Transitioning between tiers can be done at any time based on data usage changes.
- Consider the cost implications when moving data between tiers, especially to and from the Archive tier.

These notes encapsulate the different Azure Storage account types and the access tiers for block blob data, providing insights on when to use each along with a practical example for tier management.

Discover Azure Blob Storage Resource Types

Overview:

Blob storage utilizes a hierarchical structure composed of:

1. **Storage Account**
2. **Container**
3. **Blob**

Storage Accounts:

- **Purpose:** Provides a unique namespace for your data in Azure.
- **Address Structure:** Combines account name with Azure Storage blob endpoint, e.g.:

```
http://mystorageaccount.blob.core.windows.net
```

Containers:

- **Function:** Organizes blobs, akin to directories in a file system.
- **Naming Rules:**
  - Length: 3 to 63 characters.
  - Must start with a letter or number.
  - Can contain only lowercase letters, numbers, and dashes (`-`).
  - No consecutive dashes allowed.
- **Container URI Example:**

```

```

```
https://myaccount.blob.core.windows.net/mycontainer
```

Blobs:

Azure supports three blob types:

- **Block Blobs:**
  - **Use Case:** Best for storing text or binary data.
  - **Composition:** Comprised of blocks, which can be managed individually.
  - **Size Limit:** Up to approximately 190.7 TiB.
- **Append Blobs:**
  - **Use Case:** Optimized for append operations, useful for logging.
  - **Composition:** Similar to block blobs but optimized for append.
- **Page Blobs:**
  - **Use Case:** For random access files up to 8 TB, like VHD files for Azure VMs.
- **Blob URI Examples:**

```
# Basic URI
https://myaccount.blob.core.windows.net/mycontainer/myblob

# With virtual directory
https://myaccount.blob.core.windows.net/mycontainer/myvirtualdirectory/myblob
```

These notes provide a clear understanding of how Blob storage is structured within Azure, detailing the significance of each resource type and how they interact with each other.

Explore Azure Storage Security Features

Overview:

Azure Storage employs comprehensive security measures, particularly through encryption.

Service-Side Encryption (SSE):

- **Default Encryption:** Azure Storage uses Service-Side Encryption (SSE) to automatically encrypt data at the server level when it's persisted to the cloud.
- **Encryption Standard:** Utilizes 256-bit AES encryption, which is FIPS 140-2 compliant.
- **Scope:**
  - Enabled for all storage accounts by default and cannot be disabled.
  - Encrypts all data types including block blobs, append blobs, page blobs, disks, files, queues, and tables.

- Covers all performance tiers, access tiers, and deployment models.
- Applies to both primary and secondary regions when geo-replication is used.

Note: Azure Storage encryption is akin to BitLocker on Windows, providing transparent encryption and decryption.

Client-Side Encryption:

- **Client-Side Option:** For scenarios requiring client-side encryption, Azure Storage client libraries for Blob Storage and Queue Storage support this feature.

Key Points:

- **Security and Compliance:** This encryption helps in meeting organizational security and regulatory compliance requirements.
- **No Additional Cost:** Encryption through Azure Storage comes at no extra cost.
- **Transparent:** Users do not need to alter their applications or code to leverage this encryption; it works seamlessly in the background.

These notes summarize the encryption mechanisms in Azure Storage, highlighting how data security is handled both at rest and potentially on the client side, ensuring data integrity and confidentiality across various storage scenarios without additional configuration or costs.

Encryption Key Management

Default: Data in new storage accounts is encrypted with **Microsoft-managed keys**.

Key Management Options:

1. **Customer-Managed Keys:**
   - **Use Case:** For encryption and decryption of data in Blob Storage and Azure Files.
   - **Storage:** Keys must be stored in Azure Key Vault or Azure Key Vault Managed HSM.
   - **Responsibility:** Customers manage key rotation and control.
2. **Customer-Provided Keys:**
   - **Use Case:** For blob storage operations, allowing granular control over encryption.
   - **Storage:** Keys are provided by the client at the time of operation, stored externally.
   - **Responsibility:** Customers control and rotate keys.

Comparison of Key Management Options:

Key Management Parameter	Microsoft-managed keys	Customer-managed keys	Customer-provided keys
Encryption/decryption operations	Azure	Azure	Azure

Key Management Parameter	Microsoft-managed keys	Customer-managed keys	Customer-provided keys
Azure Storage services supported	All	Blob Storage, Azure Files	Blob Storage
Key storage	Microsoft key store	Azure Key Vault or Key Vault HSM	Customer's own key store
Key rotation responsibility	Microsoft	Customer	Customer
Key control	Microsoft	Customer	Customer
Key scope	Account, container, or blob	Account, container, or blob	N/A

Client-Side Encryption:

- **Supported Languages:** .NET, Java, and Python for Blob Storage; .NET and Python for Queue Storage.
- **Encryption Method:** AES (Advanced Encryption Standard).
  - **Version 2:** Uses **Galois/Counter Mode (GCM)**.
    - Supported by Blob Storage and Queue Storage SDKs.
  - **Version 1:** Uses **Cipher Block Chaining (CBC)**.
    - Supported by Blob Storage, Queue Storage, and Table Storage SDKs.

Code Example for Client-Side Encryption (Conceptual):

```
// This is a conceptual example, real implementations will vary based on the
language and library used.
BlobServiceClient blobServiceClient = new BlobServiceClient(connectionString);

// Create or get a client-side encryption key
// var encryptionKey = ... // Key generation or retrieval would go here

// Create a BlobClient with client-side encryption
BlobClient encryptedBlobClient = blobServiceClient.GetBlobClient(containerName,
blobName)
    .WithClientSideEncryptionOptions(new
ClientSideEncryptionOptions(encryptionKey));

// Upload a file with encryption
using (var fileStream = File.OpenRead(localFilePath))
{
    await encryptedBlobClient.UploadAsync(fileStream, true);
}

// Download and decrypt the blob
var blobContent = await encryptedBlobClient.DownloadAsync();
using (var memoryStream = new MemoryStream())
{
    await blobContent.Value.Content.CopyToAsync(memoryStream);
```

```
    // memoryStream now contains the decrypted blob content
}
```

These notes provide a concise overview of the key management options for Azure Storage encryption, including client-side encryption capabilities, helping developers and administrators understand how to manage security in Azure Storage.

Manage the Azure Blob Storage Lifecycle

Objective:

To understand and implement lifecycle management strategies for data in Azure Blob Storage, ensuring data availability and cost efficiency.

Key Concepts:

1. **Lifecycle Management:**
    - **Purpose:** Automate actions like transitioning data to cooler access tiers or deleting it after a set period to optimize costs and manage storage efficiently.
2. **Lifecycle Policies:**
    - Policies can be set to:
        - **Transition:** Move data from hot to cool, cold, or archive tiers based on access patterns or time since last modification.
        - **Delete:** Automatically remove blobs that are no longer needed.
3. **Example of a Lifecycle Policy in JSON:**

```
{
  "rules": [
    {
      "name": "transitionToCoolRule",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": {
              "daysAfterModificationGreaterThan": 30
            }
          }
        },
        "filters": {
          "blobTypes": ["blockBlob"],
          "prefixMatch": ["container1/prefix1"]
        }
```

```
        }
      },
      {
        "name": "deleteOldBlobsRule",
        "enabled": true,
        "type": "Lifecycle",
        "definition": {
          "actions": {
            "baseBlob": {
              "delete": {
                "daysAfterModificationGreaterThan": 365
              }
            }
          },
          "filters": {
            "blobTypes": ["blockBlob"],
            "prefixMatch": ["container2/prefix2"]
          }
        }
      }
    ]
  }
```

- **Explanation:**
  - The first rule transitions blobs to the cool tier 30 days after last modification.
  - The second rule deletes blobs after they are 365 days old.

4. **Implementation Steps:**
   - **Create/Edit Policy:** Use Azure Portal, Azure CLI, Azure PowerShell, or REST API to set up lifecycle management policies.
   - **Review:** Regularly check and adjust policies based on data usage patterns and business requirements.

5. **Benefits:**
   - **Cost Management:** Reduces storage costs by placing data on appropriate access tiers.
   - **Data Governance:** Automates compliance with data retention policies.

Remaining Units:

- Likely will cover more detailed scenarios, policy creation, monitoring, and advanced configurations.

These notes outline the basics for managing the lifecycle of data in Azure Blob Storage, providing a foundation for understanding how to automate data management tasks for efficiency and cost optimization.

Introduction to Azure Blob Storage Lifecycle Management

Overview:

Data in Azure Blob Storage goes through different lifecycle stages where its access frequency varies:

- **Initial Stage:** Data is frequently accessed.
- **Aging Stage:** Access drops significantly.
- **Idle Stage:** Data becomes rarely accessed or completely idle.

Learning Objectives:

Upon completing this module, you will:

1. **Understand Access Tiers:**
    - **Hot Tier:** Optimized for frequent access, with the highest storage cost but lowest access cost.
    - **Cool Tier:** For data accessed infrequently, offering lower storage costs but higher access costs compared to the Hot tier.
    - **Cold Tier:** For data that's accessed even less often, with even lower storage costs.
    - **Archive Tier:** For data that might not be accessed for a long time, with the lowest storage cost but highest retrieval cost.
2. **Create and Implement Lifecycle Policies:**
    - **Purpose:** Automate the transition of data through different access tiers or deletion based on time or other criteria.
    - **Example Policy Creation (Conceptual):**

```
{
  "rules": [
    {
      "name": "transitionToCoolTier",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": { "daysAfterModificationGreaterThan": 30 }
          }
        },
        "filters": {
          "blobTypes": ["blockBlob"],
          "prefixMatch": ["mycontainer/logs/"]
        }
      }
    }
  ]
}
```

        - This policy moves blobs in `mycontainer/logs/` to the Cool tier 30 days after modification.

3. **Rehydrate Blob Data from Archive:**
    - **Process:** Moving data from the Archive tier to a more accessible tier like Hot or Cool for retrieval.
    - **Note:** Rehydration involves a waiting period due to the nature of the Archive tier.

Key Takeaways:

- Data lifecycle management in Azure Blob Storage helps in optimizing costs and managing data efficiently.
- Policies can be set up to automatically handle data transitions or deletions.
- Rehydration is necessary when needing to access archived data but comes with delays and costs.

These notes encapsulate the essentials of managing data lifecycle in Azure Blob Storage, focusing on understanding access tiers, implementing lifecycle policies, and the process of data rehydration from the archive tier.

Explore the Azure Blob Storage Lifecycle

Overview:

Data in Azure Blob Storage has varying lifecycle patterns:

- **Frequent Access:** Initially, data might be accessed often.
- **Infrequent Access:** Access decreases as data ages.
- **Long-term Storage:** Some data remains idle or archived with very low access frequency.
- **Expiration:** Some data has a short lifespan, expiring quickly after creation.
- **Active Use:** Other datasets are actively used throughout their existence.

Access Tiers:

Azure provides tiered storage options to match different data access patterns:

1. **Hot Tier:**
    - **Use Case:** For data that is accessed frequently.
    - **Characteristics:** Highest storage cost, lowest access cost. Best for data needing quick access.
2. **Cool Tier:**
    - **Use Case:** For data that is infrequently accessed but needs to be online, stored for at least 30 days.
    - **Characteristics:** Lower storage costs than Hot, higher access costs.
3. **Cold Tier:**
    - **Use Case:** For data accessed even less often, stored for at least 90 days.
    - **Characteristics:** Further reduces storage costs compared to Cool, with higher retrieval costs.
4. **Archive Tier:**

- **Use Case:** For data that can tolerate several hours of latency when retrieved and is stored for at least 180 days.
- **Characteristics:** The lowest storage cost but the highest access cost due to the offline nature of the tier.

Important Note on Storage Limits:

- Storage limits apply at the account level, not per tier. This means you can distribute your storage usage across tiers as needed without worrying about tier-specific limits.

These notes provide a summary of how data lifecycle management works with Azure Blob Storage, focusing on the different access tiers available to optimize for access frequency and cost efficiency.

Manage the Data Lifecycle in Azure Blob Storage

Azure Blob Storage lifecycle management allows for defining policies to:

- **Optimize Performance:** Transition blobs from cool to hot tier when accessed.
- **Optimize Costs:** Move blobs to cooler tiers based on inactivity.
- **Lifecycle End Management:** Delete blobs at the end of their lifecycle.

Lifecycle Management Capabilities:

1. **Tier Transition:**
   - **Cool to Hot:** Automatically move blobs to the hot tier for immediate access optimization.
   - **Hot/Cool to Cooler Tiers:** Transition blobs, their previous versions, or snapshots to a cooler tier (like cool, cold, or archive) if they haven't been accessed or modified for a specified time, reducing costs.
2. **Data Expiration:**
   - **Deletion:** Automatically delete blobs, previous versions, or snapshots when they reach the end of their lifecycle.
3. **Policy Scope:**
   - Policies can be applied at:
     - **Account Level:** Affect all blobs within the storage account.
     - **Container Level:** Target specific containers.
     - **Blob Level:** Use filters like name prefixes or blob index tags to apply rules to subsets of blobs.

Example Scenario:

- **Initial Stage:** Data is frequently accessed, hence stored in the **Hot** tier for optimal access speed.
- **Two Weeks Later:** Access becomes occasional, so data is transitioned to the **Cool** tier to balance access costs with performance.

- **After One Month:** Data is rarely accessed, making the **Archive** tier ideal for long-term storage at the lowest cost.

Sample Lifecycle Management Policy (JSON):

```
{
  "rules": [
    {
      "name": "accessTierTransitionRule",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": { "daysAfterModificationGreaterThan": 14 },
            "tierToArchive": { "daysAfterModificationGreaterThan": 30 }
          }
        },
        "filters": {
          "blobTypes": ["blockBlob"],
          "prefixMatch": ["container1/folder1/"]
        }
      }
    },
    {
      "name": "deleteOldBlobsRule",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "delete": { "daysAfterModificationGreaterThan": 365 }
          }
        },
        "filters": {
          "blobTypes": ["blockBlob"],
          "prefixMatch": ["container2/folder2/"]
        }
      }
    }
  ]
}
```

- **Explanation:**
  - The first rule transitions blobs to **Cool** 14 days after modification and to **Archive** after 30 days within `container1/folder1/`.
  - The second rule deletes blobs after one year in `container2/folder2/`.

Key Takeaway:

By leveraging lifecycle management policies, you can dynamically manage your data's storage based on its lifecycle phase, ensuring cost efficiency without compromising on performance when needed.

Discover Blob Storage Lifecycle Policies

Overview:

A lifecycle management policy in Azure Blob Storage is defined as a JSON document consisting of rules. Each rule specifies actions to be taken on blobs based on filters.

Policy Structure:

```
{
  "rules": [
    {
      "name": "rule1",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {
        "actions": {
          // Actions to be performed
        },
        "filters": {
          // Filters that define the scope of blobs this rule applies to
        }
      }
    }
  ]
}
```

Policy Parameters:

- **rules**:
  - **Type:** Array of rule objects
  - **Notes:** A policy must have at least one rule, with a maximum of 100 rules allowed.

Rule Parameters:

Parameter Name	Parameter Type	Notes	Required
name	String	Can include up to 256 alphanumeric characters. Case-sensitive and must be unique within a policy.	True
enabled	Boolean	Defaults to `true`. Can be used to temporarily disable a rule.	False
type	Enum	Must be set to `"Lifecycle"`.	True
definition	Object	Contains `actions` and `filters` to define the lifecycle rule.	True

Key Points:

- **Filters:** Allow rules to target specific sets of blobs within a container or by name prefixes.
- **Actions:** Define what happens to the filtered blobs, like tiering (moving to a different access tier) or deletion.

This structure allows for fine-grained control over how data is managed over time, enabling cost optimization by moving data to appropriate storage tiers or removing unnecessary data.

Rules in Azure Blob Storage Lifecycle Management

Each rule in a lifecycle management policy consists of:

- **Filter Set:** Defines the scope of blobs the rule applies to.
- **Action Set:** Specifies what actions (tiering or deletion) are performed on the filtered blobs.

Sample Rule Example:

```
{
  "rules": [
    {
      "enabled": true,
      "name": "sample-rule",
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "version": {
            "delete": { "daysAfterCreationGreaterThan": 90 }
          },
          "baseBlob": {
            "tierToCool": { "daysAfterModificationGreaterThan": 30 },
            "tierToArchive": {
              "daysAfterModificationGreaterThan": 90,
              "daysAfterLastTierChangeGreaterThan": 7
            },
            "delete": { "daysAfterModificationGreaterThan": 2555 }
          }
        },
        "filters": {
          "blobTypes": ["blockBlob"],
          "prefixMatch": ["sample-container/blob1"]
        }
      }
    }
  ]
}
```

Explanation of Actions:

- **tierToCool:** Moves the blob to the Cool tier 30 days after last modification.

- **tierToArchive:** Moves to Archive tier 90 days after last modification, ensuring at least 7 days have passed since the last tier change.
- **delete:** Deletes the blob after 2,555 days (about 7 years) from last modification.
- **Delete Snapshots:** Deletes blob snapshots 90 days after they're created.

Rule Filters:

- **blobTypes:** Must be defined, specifies the type of blob this rule applies to (e.g., `blockBlob`).
- **prefixMatch:** Optional, matches blobs starting with a specified prefix, up to 10 prefixes per rule.
- **blobIndexMatch:** Optional, matches blobs based on blob index tag conditions.

Note: Multiple filters use a logical AND operation.

Rule Actions:

- **Action Types:**
  - **tierToCool, tierToCold, tierToArchive:** For moving blobs to respective tiers.
  - **enableAutoTierToHotFromCool:** Automatically moves blobs back to Hot from Cool when accessed (not for snapshots).
  - **delete:** Deletes blobs, snapshots, or previous versions.

Action Priority:

- If multiple actions are defined, the least expensive action is applied. For example, `delete` is cheaper than `tierToArchive`.

Action Run Conditions:

- **daysAfterModificationGreaterThan:** Used for base blob actions.
- **daysAfterCreationGreaterThan:** For blob snapshot actions.
- **daysAfterLastAccessTimeGreaterThan:** For current versions with access tracking enabled.
- **daysAfterLastTierChangeGreaterThan:** Ensures a blob stays in a tier for a minimum duration before archiving.

These conditions help in defining precise lifecycle management rules to optimize storage costs based on data usage patterns.

Implement Blob Storage Lifecycle Policies

Methods to Manage Lifecycle Policies:

- **Azure Portal**
- **Azure PowerShell**
- **Azure CLI**
- **REST APIs**

Azure Portal Implementation:

1. **Code View Method:**

- Navigate to your **Storage Account** in the Azure portal.
- Under **Data management**, select **Lifecycle Management**.
- Switch to the **Code View** tab to define your policy in JSON format.

Example Policy in JSON:

```
{
  "rules": [
    {
      "enabled": true,
      "name": "move-to-cool",
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": {
              "daysAfterModificationGreaterThan": 30
            }
          }
        },
        "filters": {
          "blobTypes": ["blockBlob"],
          "prefixMatch": ["sample-container/log"]
        }
      }
    }
  ]
}
```

- This policy targets block blobs starting with "log" in the "sample-container" and moves them to the Cool tier if they haven't been modified for 30 days.

2. **List View Method:** (Not detailed here, but involves using the UI to set rules visually.)

Azure CLI Implementation:

- **Policy Creation:**

  You first need to define your policy in a JSON file (e.g., `policy.json`). Then use the Azure CLI to apply the policy:

```
az storage account management-policy create \
    --account-name <storage-account> \
    --policy @policy.json \
    --resource-group <resource-group>
```

- **Notes:**
  - Replace `<storage-account>` with your storage account name.

- Replace `<resource-group>` with the name of the resource group where your storage account resides.
- The `@policy.json` reads the policy from a local file named `policy.json`.

Important:

- Lifecycle policies need to be read or written in their entirety; partial updates are not supported. If you need to modify a policy, you must replace the entire policy document.

Rehydrate Blob Data from the Archive Tier

Overview:

Archived blobs are offline and cannot be accessed directly. Rehydration is necessary to make them accessible again.

Rehydration Methods:

1. **Copy to Online Tier:**
   - **Operation:** Use `Copy Blob` or `Copy Blob from URL` to make a new online version of the blob.
   - **Recommended:** Preferred method by Microsoft for most scenarios.

```
# Example using Azure CLI to copy an archived blob to a hot tier
az storage blob copy start \
  --account-name <storage-account> \
  --destination-blob <destination-blob-name> \
  --source-blob <source-blob-name> \
  --destination-container <destination-container> \
  --source-container <source-container> \
  --tier Hot
```

2. **Change Blob's Access Tier:**
   - **Operation:** Use `Set Blob Tier` to change the tier directly.

```
# Example using Azure CLI to change the tier of an archived blob to Hot
az storage blob tier \
  --account-name <storage-account> \
  --container-name <container-name> \
  --name <blob-name> \
  --tier Hot
```

Rehydration Considerations:

- **Time:** The process can take several hours.
- **Performance:** Rehydrating larger blobs is recommended for better performance.

- **Multiple Small Blobs:** Rehydrating numerous small blobs at the same time might increase completion time.

Rehydration Priority:

- **Standard Priority:**
  - **Time:** Might take up to 15 hours.
  - **Usage:** Default option, processed in the order received.
- **High Priority:**
  - **Time:** Can complete in under an hour for blobs smaller than 10 GB.
  - **Usage:** Use when you need the data quickly.

Checking Rehydration Status:

- Use `Get Blob Properties` to check the `x-ms-rehydrate-priority` header, which will show either `Standard` or `High`.

```
# Example using Azure CLI to get blob properties
az storage blob show \
  --account-name <storage-account> \
  --container-name <container-name> \
  --name <blob-name> \
  --query "properties.rehydratePriority"
```

These notes outline the process and considerations for rehydrating archived blobs in Azure Blob Storage, providing examples of how to perform these operations using Azure CLI.

Rehydration Methods for Archived Blobs

Copy an Archived Blob to an Online Tier:

- **Method:** Use the `Copy Blob` operation to create a new online blob in either the Hot or Cool tier.
  - The source archived blob remains unchanged.
  - **Restrictions:**
    - For service versions before 2021-02-12, copying is limited within the same storage account.
    - From service version 2021-02-12, cross-account copying is allowed if both accounts are in the same region.

Example Command:

```
az storage blob copy start \
  --account-name <source-account> \
  --destination-blob <new-blob-name> \
```

```
  --source-blob <archived-blob-name> \
  --destination-container <destination-container> \
  --source-container <source-container> \
  --tier Hot
```

Change a Blob's Access Tier to an Online Tier:

- **Operation:** `Set Blob Tier` allows you to directly change the tier of an archived blob to Hot or Cool.
  - **Note:** This operation cannot be canceled once started.
  - **Blob Properties:** During rehydration, the blob's access tier still appears as "archived" until the process completes.

Example Command:

```
az storage blob tier \
  --account-name <storage-account> \
  --container-name <container-name> \
  --name <blob-name> \
  --tier Cool
```

Caution:

- Changing the tier does not update the last modified time.
- If there's a lifecycle management policy, the blob might be automatically moved back to the archive tier post-rehydration if the policy's conditions are met based on the unchanged last modified time.

Work with Azure Blob Storage

Objective:

This module focuses on utilizing the Azure Blob Storage client library to manage Blob storage resources effectively.

Key Topics to Learn:

1. **Introduction to Azure Blob Storage Client Library:**
   - Overview of the library, its versions, and compatibility.
2. **Setting Up the Development Environment:**
   - Installing necessary SDKs or libraries for your chosen programming language (e.g., .NET, Java, Python, JavaScript/TypeScript).
3. **Creating Storage Accounts and Containers:**
   - **Code Example for Creating a Container in .NET:**

```
BlobServiceClient blobServiceClient = new
BlobServiceClient(connectionString);
await blobServiceClient.CreateBlobContainerAsync("mycontainer");
```

4. **Uploading Blobs:**
    - **Uploading a File:**

```
BlobContainerClient containerClient =
blobServiceClient.GetBlobContainerClient("mycontainer");
BlobClient blobClient = containerClient.GetBlobClient("mypicture.jpg");
await blobClient.UploadAsync("path/to/local/mypicture.jpg", true);
```

5. **Downloading Blobs:**
    - **Downloading Blob Content:**

```
using (var memoryStream = new MemoryStream())
{
    await blobClient.DownloadToAsync(memoryStream);
    memoryStream.Position = 0;
    using (var fileStream =
File.Create("path/to/download/mypicture.jpg"))
    {
        memoryStream.CopyTo(fileStream);
    }
}
```

6. **Managing Blob Metadata:**
    - Adding, retrieving, or updating metadata associated with blobs.
7. **Blob Properties and Permissions:**
    - Setting and getting blob properties like content type, lease status, etc.
    - Managing permissions, including setting access policies for containers.
8. **Blob Snapshots and Leases:**
    - Creating snapshots for point-in-time copies.
    - Implementing blob leases for exclusive access.

Practical Exercises:

- Exercises might include tasks like uploading different types of data, setting lifecycle policies, or managing blob properties programmatically.

Best Practices:

- Understanding how to optimize blob uploads/downloads for performance.
- Secure handling of storage account keys and connection strings.
- Implementing proper error handling and retry policies.

This module will equip you with the skills to programmatically interact with Azure Blob Storage, enhancing your ability to develop applications that leverage cloud storage effectively.

Introduction to Azure Storage Client Libraries for .NET

Overview:

The Azure Storage client libraries for .NET provide an intuitive interface to interact with Azure Storage services, particularly useful for Blob storage operations.

Learning Objectives:

Upon completing this module, you will be able to:

1. **Create and Manipulate Blob Data:**
    - Use the Azure Blob Storage client library to:
        - Create new blobs
        - Upload data to blobs
        - Download blobs
        - Modify blob content or properties

    Example for Uploading a Blob:

    ```
    BlobServiceClient blobServiceClient = new
    BlobServiceClient(connectionString);
    BlobContainerClient containerClient =
    blobServiceClient.GetBlobContainerClient("mycontainer");
    BlobClient blobClient = containerClient.GetBlobClient("myblob");

    await blobClient.UploadAsync("path/to/local/file.txt", overwrite: true);
    ```

2. **Manage Container Properties and Metadata:**
    - Utilize .NET to:
        - Create, list, or delete containers
        - Set and retrieve container metadata
        - Manage access policies for containers

    Example for Setting Container Metadata:

```
Dictionary<string, string> metadata = new Dictionary<string, string>
{
    {"key1", "value1"},
    {"key2", "value2"}
};
await containerClient.SetMetadataAsync(metadata);
```

REST Operations:

- While the module focuses on .NET, understanding REST API operations can enhance your ability to manage Azure Storage resources:
    - **Get Container Properties:**

      ```
      GET /mycontainer?restype=container
      ```

    - **Set Container Metadata:**

      ```
      PUT /mycontainer?restype=container&comp=metadata
      x-ms-meta-key1: value1
      x-ms-meta-key2: value2
      ```

These notes provide a foundation for understanding how to work with Azure Blob Storage using .NET client libraries, including basic operations and metadata management, with examples illustrating common tasks.

Explore Azure Blob Storage Client Library for .NET

Overview:

The Azure Storage client libraries for .NET provide a high-level interface for interacting with Azure Blob Storage. **Version 12.x** is the latest and recommended version for new applications.

Core Classes:

Class	Description
BlobClient	Enables operations on individual blobs like uploading, downloading, copying, and deleting.
BlobClientOptions	Configures client behavior, including retry policies, logging, and telemetry when connecting to Blob Storage.
BlobContainerClient	Used for managing blob containers, including creating, deleting, or listing containers, and performing operations on all blobs within a container.

Class	Description
BlobServiceClient	Manages operations at the storage account level, allowing access to containers within the account.
BlobUriBuilder	Helps in constructing URIs to Azure Blob Storage resources dynamically. This is useful for building URLs to blobs, containers, or the storage service itself.

Relevant Packages for Blob Storage:

- **Azure.Storage.Blobs:**
  - Includes primary client objects (`BlobClient`, `BlobContainerClient`, `BlobServiceClient`).
  - Used for general operations on Blob Storage resources.
- **Azure.Storage.Blobs.Specialized:**
  - Contains specialized classes for specific blob types like `BlockBlobClient`, `AppendBlobClient`, and `PageBlobClient`.
  - Useful for operations tailored to the unique characteristics of different blob types.
- **Azure.Storage.Blobs.Models:**
  - Encompasses additional classes, structures, and enums that serve as utility types for Blob Storage operations.

Using the Library:

Example to Create a BlobContainerClient:

```
BlobServiceClient blobServiceClient = new BlobServiceClient(connectionString);
BlobContainerClient containerClient =
blobServiceClient.GetBlobContainerClient(containerName);
```

Uploading a Blob:

```
BlobClient blobClient = containerClient.GetBlobClient("myblob");
await blobClient.UploadAsync("path/to/local/file.txt", true);
```

Remember:

- Always use the latest stable version of the library to take advantage of new features, improvements, and security patches.
- The client library abstracts much of the complexity of dealing with Azure Storage directly via REST APIs, making development more straightforward.

Create a Client Object with Azure Blob Storage SDK

Overview:

To interact with Azure Blob Storage resources (storage accounts, containers, blobs) using the SDK, you first need to create client objects.

Key Points:

1. **Client Creation:**
    - Client objects are created by passing a URI and credentials to the respective client constructor.
    - URIs can be manually constructed or dynamically fetched using Azure Storage management libraries.
2. **Authentication:**
    - `DefaultAzureCredential` is used for authentication, providing an access token for Azure Entra (formerly Azure AD) security principal.
    - The security principal must have the necessary Azure RBAC role assignments for blob data access.
3. **BlobServiceClient:**
    - This client interacts with the storage account level, allowing operations like:
        - Retrieving and configuring account properties.
        - Listing, creating, or deleting containers.

Example of Creating a `BlobServiceClient`:

```
using Azure.Identity;
using Azure.Storage.Blobs;

public BlobServiceClient GetBlobServiceClient(string accountName)
{
    // Construct the URI for the Blob storage endpoint
    BlobServiceClient client = new(
        new Uri($"https://{accountName}.blob.core.windows.net"), // Endpoint URI
        new DefaultAzureCredential()); // Credential for authentication

    return client;
}
```

Notes:

- `DefaultAzureCredential` tries various credential sources, making it versatile for development, testing, and production environments.
- The `BlobServiceClient` created here can then be used to perform operations on containers or blobs within the specified storage account.

Client Objects for Specific Blob Storage Resources

Create a `BlobContainerClient` Object:

- **From `BlobServiceClient`:**
  - This method is useful when you need to manage multiple containers or perform operations at the account level before focusing on a specific container.

```
public BlobContainerClient GetBlobContainerClient(
    BlobServiceClient blobServiceClient,
    string containerName)
{
    // Create a container client from the service client
    BlobContainerClient client =
blobServiceClient.GetBlobContainerClient(containerName);
    return client;
}
```

- **Directly:**
  - Ideal for scenarios where your operations are focused on a single container.

```
public BlobContainerClient GetBlobContainerClient(
    string accountName,
    string containerName,
    BlobClientOptions clientOptions)
{
    // Directly create a container client with the full URI
    BlobContainerClient client = new(
        new Uri($"https://{accountName}.blob.core.windows.net/{containerName}"),
        new DefaultAzureCredential(),
        clientOptions);

    return client;
}
```

Create a `BlobClient` Object:

- `BlobClient` is used for operations on individual blobs, like uploading, downloading, or deleting a blob.

```
public BlobClient GetBlobClient(
    BlobServiceClient blobServiceClient,
    string containerName,
    string blobName)
{
    // Create a blob client from the container client
    BlobClient client =

blobServiceClient.GetBlobContainerClient(containerName).GetBlobClient(blobName);
```

```
    return client;
}
```

Notes:

- `BlobContainerClient` provides methods to manage the container and its blobs, like creating, listing, or deleting blobs.
- `BlobClient` is tailored for operations on a single blob, offering fine-grained control over blob-specific tasks.
- Using `BlobClientOptions` allows you to customize the behavior of the client, like setting retry policies or specifying how the client should handle HTTP requests.

Exercise: Create Blob Storage Resources Using .NET Client Library

Objective:

Demonstrate how to interact with Azure Blob Storage using the .NET client library within a console application, covering:

- Creating a container
- Uploading blobs
- Listing blobs
- Downloading blobs
- Deleting a container

Prerequisites:

- An **Azure account** with an active subscription.
- **Visual Studio Code** installed.
- **.NET 8** as the target framework.
- **C# extension** for Visual Studio Code installed.
- **Azure CLI** installed.

Setup Instructions:

1. **Open Visual Studio Code:**
    - Launch Visual Studio Code and open a terminal via **Terminal > New Terminal** from the top menu.
2. **Sign in to Azure:**

    ```
    az login
    ```

    - This command will prompt you to sign in through a browser window.
3. **Create Resource Group:**

```
az group create --location <myLocation> --name az204-blob-rg
```

   - Replace `<myLocation>` with your preferred Azure region.

4. **Create Storage Account:**

```
az storage account create --resource-group az204-blob-rg --name
<myStorageAcct> --location <myLocation> --sku Standard_LRS
```

   - Replace `<myStorageAcct>` with a unique name for your storage account. Remember:
     - The name should be 3-24 characters long.
     - Only lowercase letters and numbers are allowed.
     - It must be unique across Azure.

5. **Retrieve Storage Account Credentials:**

   - Go to the **Azure Portal**.
   - Find your storage account under the **az204-blob-rg** resource group.
   - Navigate to **Security + networking > Access keys**.
   - Copy the **Connection string** from **key1** for later use in your application.

6. **Prepare for Exercise:**

   - In the storage account overview, navigate to the **Blobs** section and select **Containers** to monitor changes during the exercise.

Next Steps:

- Proceed with coding your console application using the copied connection string to interact with Blob Storage, performing the operations listed in the objective.

Prepare the .NET Project for Azure Blob Storage

Important Security Considerations:

- The example uses a connection string for authentication, which is not optimal for security.
- For production scenarios, consider using managed identities for Azure resources to authorize data access.

Create and Configure the Project:

1. **Create the Project:**

```
dotnet new console -n az204-blob
```

- This creates a console app named `az204-blob`.

2. **Navigate and Build:**

```
cd az204-blob
dotnet build
```

- Move into the project directory and ensure it builds correctly.

3. **Create Data Directory:**

```
mkdir data
```

- This directory will hold blob data files.

4. **Install Azure Blob Storage Client Library:**

```
dotnet add package Azure.Storage.Blobs
```

5. **Modify `Program.cs`:** Replace the contents of `Program.cs` with:

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;

Console.WriteLine("Azure Blob Storage exercise\n");

// Run the examples asynchronously, wait for the results before proceeding
ProcessAsync().GetAwaiter().GetResult();

Console.WriteLine("Press enter to exit the sample application.");
Console.ReadLine();

static async Task ProcessAsync()
{
    // Copy the connection string from the portal in the variable below.
    string storageConnectionString = "CONNECTION STRING";

    // Create a client that can authenticate with a connection string
    BlobServiceClient blobServiceClient = new
BlobServiceClient(storageConnectionString);

    // COPY EXAMPLE CODE BELOW HERE
}
```

Next Steps:

- **Set Connection String:** Replace `"CONNECTION STRING"` with the actual connection string you copied from the Azure portal.
- **Implement Blob Storage Operations:**
    - You'll need to add code within the `ProcessAsync` method to perform operations like creating containers, uploading, listing, downloading, and deleting blobs.

Ensure the terminal remains open for building and running the application as you progress through the exercise.

Build the Full App for Azure Blob Storage Interaction

Create a Container:

- **Objective:** Create a uniquely named container in your Azure Blob Storage account.
- **Code Snippet to Add to `Program.cs`:**

```
// Create a unique name for the container
string containerName = "wtblob" + Guid.NewGuid().ToString();

// Create the container and return a container client object
BlobContainerClient containerClient = await
blobServiceClient.CreateBlobContainerAsync(containerName);
Console.WriteLine("A container named '" + containerName + "' has been created. " +
    "\nTake a minute and verify in the portal." +
    "\nNext a file will be created and uploaded to the container.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Notes:

- A `GUID` is used to ensure the container name is unique, preventing conflicts if the app is run multiple times.
- `CreateBlobContainerAsync` is used to create the container. If a container with the same name exists, this method will throw an exception. Ensure you handle or check for existing containers if this behavior is not desired.

Next Steps:

- After this snippet, continue by adding code for uploading blobs to the newly created container. Remember to keep appending new functionalities sequentially in the `ProcessAsync` method.

Upload Blobs to a Container:

- **Objective:** Upload a local file to the container created earlier.

- **Code Snippet to Add to `Program.cs`:**

```
// Create a local file in the ./data/ directory for uploading and downloading
string localPath = "./data/";
string fileName = "wtfile" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to the blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n", blobClient.Uri);

// Open the file and upload its data
using (FileStream uploadFileStream = File.OpenRead(localFilePath))
{
    await blobClient.UploadAsync(uploadFileStream);
    uploadFileStream.Close();
}

Console.WriteLine("\nThe file was uploaded. We'll verify by listing" +
        " the blobs next.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Notes:

- A new file is created with a unique name using `Guid`.
- `UploadAsync` is used to upload the file to Blob Storage, creating a new blob if it doesn't exist or overwriting if it does.

List the Blobs in a Container:

- **Objective:** List all blobs within the container to verify the upload.
- **Code Snippet to Add to `Program.cs`:**

```
// List blobs in the container
Console.WriteLine("Listing blobs...");
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}

Console.WriteLine("\nYou can also verify by looking inside the " +
        "container in the portal." +
        "\nNext the blob will be downloaded with an altered file name.");
```

```
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Notes:

- `GetBlobsAsync` is used to retrieve an asynchronous enumerable of `BlobItem` which represents blobs in the container.
- This snippet will list the name of the blob just uploaded, confirming its presence in the container.

Download Blobs:

- **Objective:** Download the previously uploaded blob to the local file system.
- **Code Snippet to Add to `Program.cs`:**

```
// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
string downloadFilePath = localFilePath.Replace(".txt", "DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
BlobDownloadInfo download = await blobClient.DownloadAsync();

using (FileStream downloadFileStream = File.OpenWrite(downloadFilePath))
{
    await download.Content.CopyToAsync(downloadFileStream);
}
Console.WriteLine("\nLocate the local file in the data directory created earlier
to verify it was downloaded.");
Console.WriteLine("The next step is to delete the container and local files.");
Console.WriteLine("Press 'Enter' to continue.");
Console.ReadLine();
```

Notes:

- A new file name is created by appending "DOWNLOADED" to distinguish it from the original local file.
- `DownloadAsync` retrieves the blob content, which is then written to a local file using a `FileStream`.

Delete a Container:

- **Objective:** Clean up by deleting the container and removing local files.
- **Code Snippet to Add to `Program.cs`:**

```
// Delete the container and clean up local files created
Console.WriteLine("\n\nDeleting blob container...");
```

```
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);

Console.WriteLine("Finished cleaning up.");
```

Notes:

- `DeleteAsync` is called on the container client to delete the container and all blobs inside it.
- Local files created during the exercise are also deleted to ensure a clean state afterward.

Run the Code:

- **Objective:** Execute the completed application to see the operations in action.
- **Steps:**
    1. **Ensure you're in the application directory.**
    2. **Build the application:**

    ```
    dotnet build
    ```

    3. **Run the application:**

    ```
    dotnet run
    ```

Notes:

- The application will pause at various points with prompts, allowing you to check the Azure portal for changes after each operation (container creation, blob upload, listing, download, and deletion).

Clean Up Other Resources:

- **Objective:** Remove any remaining Azure resources used in the exercise.
- **Command to Delete Resource Group:**

```
az group delete --name az204-blob-rg --no-wait
```

Notes:

- This command deletes the entire resource group `az204-blob-rg`, which includes the storage account and any other resources created within it.
- The `--no-wait` flag allows the command to execute without waiting for the operation to complete, which can be useful for long-running operations or when you want to continue with other tasks immediately. However, it's worth noting that this means you won't receive immediate feedback on whether the deletion was successful.

Manage Container Properties and Metadata by Using .NET

Overview:

Azure Blob Storage containers support:

- **System Properties:** These are inherent properties of Blob storage resources, managed by Azure. Some can be read or modified, while others are read-only.
- **User-Defined Metadata:** Custom name-value pairs that users can set on Blob storage resources for additional context or data.

System Properties:

- They relate to standard HTTP headers and are managed by the Azure Storage client library.

User-Defined Metadata:

- Can be set by the user.
- Must follow HTTP header rules:
  - Names must be valid HTTP headers and C# identifiers.
  - Only ASCII characters are allowed in names.
  - Case-insensitive.
  - Non-ASCII values should be encoded (Base64 or URL-encoded).

Retrieve Container Properties:

- Use `GetProperties` or `GetPropertiesAsync` methods of `BlobContainerClient` to fetch container properties.

Example Code to Read Container Properties:

```
private static async Task ReadContainerPropertiesAsync(BlobContainerClient
container)
{
    try
    {
        // Fetch some container properties and write out their values.
        var properties = await container.GetPropertiesAsync();
        Console.WriteLine($"Properties for container {container.Uri}");
        Console.WriteLine($"Public access level:
{properties.Value.PublicAccess}");
```

```
        Console.WriteLine($"Last modified time in UTC:
{properties.Value.LastModified}");
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Notes:

- This example shows how to retrieve and display the `PublicAccess` level and `LastModified` time of a container.
- Error handling is implemented to catch and display information about any `RequestFailedException` that might occur during the operation.

Set and Retrieve Metadata for Azure Blob Storage Containers

Setting Metadata:

- Metadata for containers can be set using name-value pairs stored in an `IDictionary<string, string>`.
- Use `SetMetadata` or `SetMetadataAsync` of `BlobContainerClient` to apply the metadata.

Example Code to Set Container Metadata:

```
public static async Task AddContainerMetadataAsync(BlobContainerClient container)
{
    try
    {
        IDictionary<string, string> metadata = new Dictionary<string, string>();

        // Add some metadata to the container.
        metadata.Add("docType", "textDocuments");
        metadata.Add("category", "guidance");

        // Set the container's metadata.
        await container.SetMetadataAsync(metadata);
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Key Points:

- Metadata names should conform to C# identifier naming rules but are case-insensitive when accessed.
- If duplicate metadata names are provided, values are concatenated with commas.

Retrieving Metadata:

- Metadata can be retrieved along with properties using `GetProperties` or `GetPropertiesAsync`.

Example Code to Read Container Metadata:

```
public static async Task ReadContainerMetadataAsync(BlobContainerClient container)
{
    try
    {
        var properties = await container.GetPropertiesAsync();

        // Enumerate the container's metadata.
        Console.WriteLine("Container metadata:");
        foreach (var metadataItem in properties.Value.Metadata)
        {
            Console.WriteLine($"\tKey: {metadataItem.Key}");
            Console.WriteLine($"\tValue: {metadataItem.Value}");
        }
    }
    catch (RequestFailedException e)
    {
        Console.WriteLine($"HTTP error code {e.Status}: {e.ErrorCode}");
        Console.WriteLine(e.Message);
        Console.ReadLine();
    }
}
```

Notes:

- The example iterates over the metadata dictionary to display each key-value pair.
- Error handling is included in both examples to manage potential issues like network errors or unauthorized access.

Set and Retrieve Properties and Metadata for Blob Resources Using REST

Metadata Header Format:

- Metadata headers follow the format:

```
x-ms-meta-name:string-value
```

- **Since version 2009-09-19**, metadata names must follow C# identifier naming conventions.
- Names are case-insensitive when set or read, but the original case is preserved.
- Duplicate metadata names cause a `400 (Bad Request)` error.

Key Points:

- Total metadata size can be up to 8 KB.
- Metadata must adhere to HTTP header constraints.

Operations on Metadata:

- Metadata can be added, updated, or retrieved without altering the resource's content.
- Full metadata must be read or written; partial updates aren't supported.

Retrieving Properties and Metadata:

- **For Containers:** Use `GET` or `HEAD` with the following URI:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer?restype=container
```

- **For Blobs:** Use `GET` or `HEAD` with:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metadata
```

Notes:

- These operations return only headers, not the actual content of the blob or container.
- `GET` will fetch the metadata, whereas `HEAD` will return only the headers without downloading the resource's content, which is more efficient for just checking metadata.

Setting Metadata Headers for Blob Storage Resources

PUT Operation for Metadata:

- **For Containers:** Use `PUT` with the following URI:

```
PUT https://myaccount.blob.core.windows.net/mycontainer?
comp=metadata&restype=container
```

- **For Blobs:** Use `PUT` with:

```
PUT https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=metadata
```

Notes:

- The `PUT` operation with metadata headers will overwrite existing metadata or clear all metadata if no headers are provided in the request.

Standard HTTP Properties:

- Both containers and blobs support standard HTTP properties alongside custom metadata:
  - **For Containers:**
    - `ETag`
    - `Last-Modified`
  - **For Blobs:**
    - `ETag`
    - `Last-Modified`
    - `Content-Length`
    - `Content-Type`
    - `Content-MD5`
    - `Content-Encoding`
    - `Content-Language`
    - `Cache-Control`
    - `Origin`
    - `Range`

Key Points:

- Metadata headers start with `x-ms-meta-`, whereas property headers use standard HTTP names as defined in the HTTP/1.1 protocol.
- Properties like `ETag` and `Last-Modified` are crucial for versioning and synchronization, while others like `Content-Type` and `Content-Encoding` define how the blob's content should be interpreted or handled.

AZ-204: Develop Solutions with Azure Cosmos DB

Focus:

- Creating Azure Cosmos DB resources
- Understanding consistency levels
- Using .NET SDK V3 for data operations

Prerequisites:

- **Experience:** At least 1 year in scalable solution development
- **Azure Knowledge:** Basic understanding of Azure, cloud concepts, services, and the Azure portal
- **Recommendation:** Complete AZ-900: Azure Fundamentals if you're new to Azure or cloud computing

Key Learning Points:

1. **Resource Creation:**
    - Learn to set up Azure Cosmos DB databases, containers, and items.
    - Understand different consistency levels and choose appropriately.
2. **.NET SDK V3:**

```
// Example of creating a container in Azure Cosmos DB using .NET SDK V3
using Microsoft.Azure.Cosmos;

// Initialize client
CosmosClient client = new CosmosClient("your-account-endpoint", "your-account-
key");
Database database = await client.CreateDatabaseIfNotExistsAsync("YourDatabaseId");

// Create a container
ContainerProperties containerProperties = new
ContainerProperties("YourContainerId", "/partitionKey");
Container container = await
database.CreateContainerIfNotExistsAsync(containerProperties, 400);
```

- Perform CRUD operations on your data.

Note: Ensure you have the Azure Cosmos DB SDK installed in your environment for this to work.

Tips:

- Experiment with different consistency levels to see how they impact your application's performance and data integrity.
- Always consider the partition key strategy for efficient data distribution and query performance.

This course will transform you from a 'Cloud Curious' to a 'Cosmos Conqueror' in no time, ready to wield the power of Azure Cosmos DB like a true intergalactic developer!

Azure Cosmos DB Introduction

Key Benefits of Azure Cosmos DB:

- **Global Distribution:** Data can be read from and written to local replicas, enhancing performance.
- **Automatic Replication:** Data is automatically synchronized across regions.

Understanding Azure Cosmos DB Structure:

- **Account:** The top-level container in Azure Cosmos DB, which can span multiple regions.
- **Database:** A logical namespace for grouping containers.
- **Container:** Holds data items, stored procedures, triggers, and UDFs. Containers can be collections, tables, or graphs depending on the API used.

Consistency Levels:

- **Strong:** Guarantees data consistency at the cost of higher latency.
- **Bounded Staleness:** Balances between strong consistency and availability.
- **Session:** Ensures read-your-own-writes consistency within a session.
- **Consistent Prefix:** Ensures that writes are seen in the order they were written.
- **Eventual:** Offers the highest availability with eventual consistency.

Choosing the Right Consistency Level:

- Select based on your application's need for consistency vs. availability.

APIs in Azure Cosmos DB:

- **SQL API** - For document databases, similar to JSON storage.
- **MongoDB API** - MongoDB wire protocol compatible.
- **Cassandra API** - For high-scale, low-latency needs.
- **Gremlin API** - For graph databases.
- **Table API** - Key-value store, similar to Azure Table storage.
- **Etcd API** - Distributed key-value store.

Request Units (RUs):

- **Impact on Costs:** RUs are the unit of measure for throughput. Understanding and optimizing RU usage directly affects the cost efficiency of your database operations.

```
// Example: Querying data in Azure Cosmos DB with SQL API
using Microsoft.Azure.Cosmos;

// Initialize client
var client = new CosmosClient("your-account-endpoint", "your-account-key");

// Query data
var queryText = "SELECT * FROM c WHERE c.status = 'active'";
var result = client.GetContainer("databaseId",
"containerId").GetItemQueryIterator<dynamic>(queryText);

while (result.HasMoreResults)
{
    var response = await result.ReadNextAsync();
    foreach (var item in response)
    {
        Console.WriteLine(item);
    }
}
```

Creating Cosmos DB Resources:

- Use the Azure portal to:

    - Set up new accounts
    - Create databases
    - Define containers with specific throughput settings

Note: Always consider the global distribution settings when creating resources to align with your application's geographic needs.

This summar- should give you a solid foundation for what Azure Cosmos DB offers, how it's structured, and how to start using it effectively. Remember, in the vast galaxy of databases, Azure Cosmos DB is like the starship Enterprise - it can take you to new worlds of data management with global reach!

Key Benefits of Azure Cosmos DB

Design Principles:

- **Fully Managed NoSQL:** No need for database administration.
- **Low Latency:** Optimized for fast data access.
- **Elastic Scalability:** Scale throughput up or down instantly.
- **Consistent Data:** Offers well-defined consistency models.
- **High Availability:** Designed for maximum uptime.

Global Distribution:

- **Multi-Region Deployment:**
    - Deploy your database across multiple Azure regions.
    - Choose regions based on user locations for reduced latency.
- **Dynamic Region Management:**
    - Add or remove regions without application downtime.
    - No need for redeployment when adjusting regions.

Multi-Master Replication:

- **Unlimited Scalability:**
    - Both read and write operations can scale independently.
- **High Availability:**
    - 99.999% availability for reads and writes globally.
- **Low Latency:**
    - Reads and writes are served in less than 10ms at the 99th percentile.

Example of Adding Region via Azure CLI:

```
az cosmosdb create \
    --name myCosmosDbAccount \
    --resource-group myResourceGroup \
    --kind GlobalDocumentDB \
    --locations regionName1=Primary regionName2=Secondary
```

Code Example for Multi-Region Writes in .NET:

```
using Microsoft.Azure.Cosmos;

// Initialize client with multiple regions
var client = new CosmosClient("your-account-endpoint", "your-account-key", new
CosmosClientOptions
{
    ApplicationRegion = Regions.EastUS,
    ApplicationPreferredRegions = new List<string> { "East US", "West US" }
});

// Perform write operation, which can go to any region
var container = client.GetContainer("databaseId", "containerId");
await container.CreateItemAsync(newItem);
```

Automatic Replication:

- Azure Cosmos DB takes care of data replication across regions based on your selected consistency level.

Disaster Recovery:

- **Failover:** If one region becomes unavailable, others can handle requests, ensuring continuous operation.

Note: Remember, with Azure Cosmos DB, you're not just managing data; you're orchestrating a symphony of global replication that would make even the most seasoned database conductor nod in approval!

Azure Cosmos DB Resource Hierarchy

Azure Cosmos DB Account:

- **DNS Name:** Unique identifier for your Cosmos DB account.
- **Management Tools:**
  - Azure Portal
  - Azure CLI

```
# Example Azure CLI command to create a Cosmos DB account
az cosmosdb create --name myCosmosDbAccount --resource-group myResourceGroup
```

- **Global Distribution:**
  - Add or remove Azure regions dynamically for data distribution.

Resource Limits:

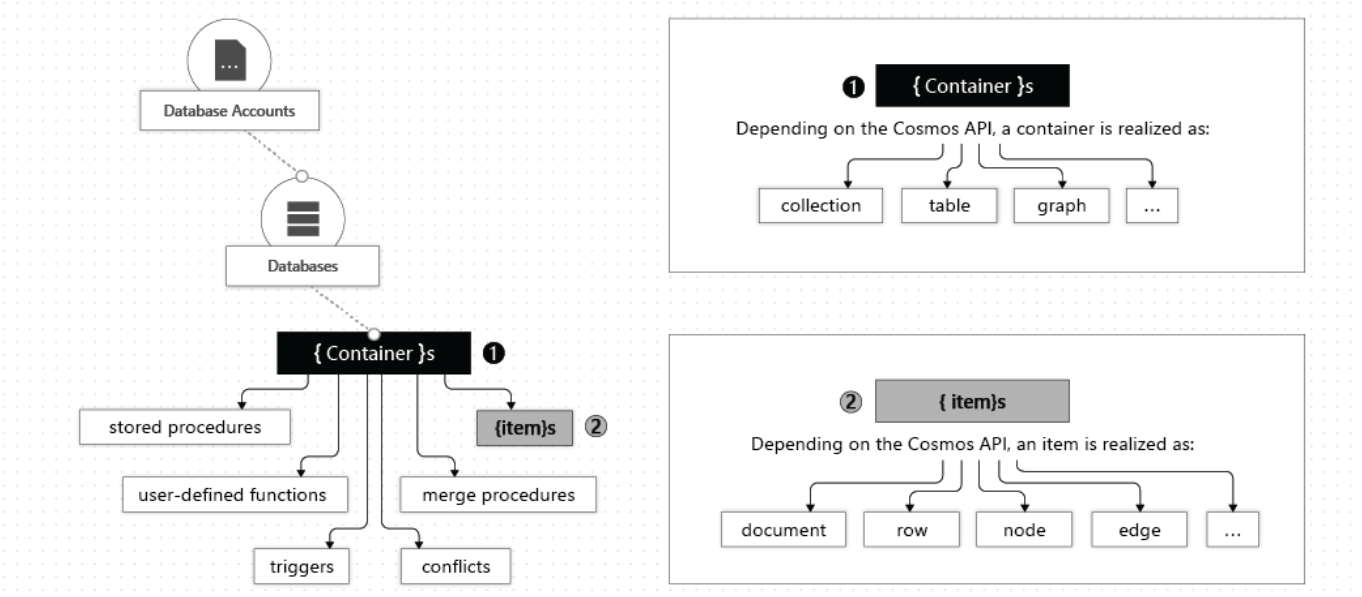
- **Account Limit:** Up to 50 accounts per Azure subscription (support request can increase this limit).

Hierarchy Elements:

1. **Account**
   - Top-level entity in Azure Cosmos DB structure.
   - Contains databases, containers, and items.
2. **Database**
   - Logical grouping of containers.
   - Acts as a namespace.
3. **Container**
   - **Scalability Unit:** Central to provisioned throughput and storage.
   - Uses **logical partition keys** for data partitioning and scaling.

Visual Representation of Hierarchy:

```
Account
|-- Database 1
|   |-- Container A
|   |-- Container B
|-- Database 2
    |-- Container C
```



Code Example for Creating Database and Container:

```
using Microsoft.Azure.Cosmos;

// Initialize client
var client = new CosmosClient("your-account-endpoint", "your-account-key");

// Create database
Database database = await client.CreateDatabaseIfNotExistsAsync("MyDatabase");

// Create container with partition key
ContainerProperties containerProperties = new ContainerProperties("MyContainer",
"/partitionKey");
Container container = await
database.CreateContainerIfNotExistsAsync(containerProperties, 400);
```

Notes:

- Containers in Azure Cosmos DB are where the magic happens, where your data gets the VIP treatment of scalability and throughput.
- The logical partition key is your secret sauce for data distribution, making sure data is evenly spread across the cosmos of your database.

This structure allows for efficient management and scalability of data within Azure Cosmos DB, ensuring that your applications can grow without bounds, much like the universe itself!

Azure Cosmos DB Resource Structure

Databases:

- **Creation:** Create one or multiple under an account.
- **Function:** Serves as a namespace for grouping containers.

Containers:

- **Data Storage:** Primary place where data resides.
- **Scalability:**
    - Unlike vertical scaling in relational DBs, Cosmos DB scales horizontally by adding partitions.
- **Partitioning:**
    - **Physical Partitions:** Each can handle up to 10,000 RU/s and store 50 GB.
    - **Logical Partitions:** Abstracted for easier management, can store up to 20 GB.
- **Partition Key:**
    - Required when creating a container.

- **Function:** Distributes data across partitions for load balancing and efficient retrieval.
- **Example Use:**

```
SELECT * FROM c WHERE c.partitionKey = 'someKey'
```

Throughput Modes:

Dedicated Throughput:

- **Standard:** Fixed amount of throughput assigned.
- **Autoscale:** Throughput automatically scales based on usage.

Shared Throughput:

- **Database Level:** Throughput shared across up to 25 containers.
- **Exclusions:** Containers with dedicated throughput don't share this pool.

Azure Cosmos DB Items:

- **Data Representation:** Varies by API used:
  - **API for NoSQL:** Items
  - **API for Cassandra:** Rows
  - **API for MongoDB:** Documents
  - **API for Gremlin:** Nodes or edges
  - **API for Table:** Items

Example of Creating a Container with Partition Key in .NET:

```
using Microsoft.Azure.Cosmos;

// Initialize client
var client = new CosmosClient("your-account-endpoint", "your-account-key");

// Create a database
Database database = await client.CreateDatabaseIfNotExistsAsync("MyDatabase");

// Define container properties with partition key
ContainerProperties containerProperties = new ContainerProperties("MyContainer",
"/partitionKey");

// Create container with shared throughput
await database.CreateContainerIfNotExistsAsync(containerProperties, throughput:
400);
```

Notes:

- Think of the partition key as the sorting hat in Hogwarts, deciding which partition your data will call home for optimal performance.

- Throughput in Azure Cosmos DB is like the energy drink for your database, giving it the power to handle more requests or scale back when it's time to chill.

Consistency Levels in Azure Cosmos DB

Azure Cosmos DB provides a spectrum of consistency levels, allowing developers to balance between consistency, availability, and performance:

Consistency Levels:

1. **Strong Consistency:**
    - **Guarantee:** The read will return the most recent write.
    - **Trade-offs:** Highest latency, lowest availability among the options.
2. **Bounded Staleness:**
    - **Guarantee:** Reads lag behind writes by a specified amount (staleness window).
    - **Trade-offs:** Moderate latency, good availability.
3. **Session Consistency:**
    - **Guarantee:** Read-your-own-writes consistency within a single client session.
    - **Use Case:** Ideal for user-specific data where write-after-read consistency is needed.
4. **Consistent Prefix:**
    - **Guarantee:** Reads never see out-of-order writes, but they might not reflect all writes.
    - **Trade-offs:** Provides a balance between strong and eventual consistency.
5. **Eventual Consistency:**
    - **Guarantee:** Reads will eventually reflect all writes, but with no guaranteed order or timing.
    - **Trade-offs:** Highest availability, lowest latency among the options.

Visual Representation:

```
Strong → Bounded Staleness → Session → Consistent Prefix → Eventual
```

Key Features:

- **Region Agnostic:** Consistency levels apply globally, regardless of regions involved.
- **Operation Scope:** Consistency is per read operation within a partition-key range.

Example of Setting Consistency Level in .NET:

```
using Microsoft.Azure.Cosmos;
```

```
// Initialize client with session consistency
CosmosClientOptions options = new CosmosClientOptions
{
    ConsistencyLevel = ConsistencyLevel.Session
};

CosmosClient client = new CosmosClient("your-account-endpoint", "your-account-
key", options);
```

Notes:

- When choosing a consistency level, think of it as picking your favorite flavor of ice cream for your data; it's all about the taste, or in this case, the balance of consistency versus performance you're willing to accept.
- The level you choose affects how your data behaves like a time traveler through your application, from the immediate past to the eventual future of your data's journey.

Choosing the Right Consistency Level in Azure Cosmos DB

Considerations for Consistency Levels:

- **Strong:**
  - **Use Case:** Financial transactions, where every read must reflect the latest write.
- **Bounded Staleness:**
  - **Use Case:** Applications needing near-real-time consistency but can tolerate a small lag, like stock trading apps.
- **Session:**
  - **Use Case:** User profile management where users should see their own updates immediately but don't need global consistency.
- **Consistent Prefix:**
  - **Use Case:** Collaborative editing or chat applications where seeing the writes in order is crucial, but not necessarily the latest.
- **Eventual:**
  - **Use Case:** Social media feeds, content delivery networks where eventual consistency is acceptable.

Configuring Default Consistency:

- **Setting the Default:**
  - You can set a default consistency level for your entire Azure Cosmos DB account.

- This level applies to all databases and containers within that account unless overridden.

Example of Configuring Default Consistency via Azure Portal:

1. Navigate to your Cosmos DB account in the Azure portal.
2. Go to **Settings** -> **Default consistency**.
3. Select the consistency level from the dropdown.

Or using Azure CLI:

```
az cosmosdb update --name myCosmosDbAccount --resource-group myResourceGroup --
default-consistency-level Session
```

Scope of Read Consistency:

- **Operation Scope:** A single read operation within a logical partition.

Code Example for Client-side Consistency Override in .NET:

```
using Microsoft.Azure.Cosmos;

// Initialize client
CosmosClient client = new CosmosClient("your-account-endpoint", "your-account-
key");

// For a specific operation, override the default consistency
var container = client.GetContainer("databaseId", "containerId");

// Override to Strong consistency for this read operation
var options = new ItemRequestOptions { ConsistencyLevel = ConsistencyLevel.Strong
};
var item = await container.ReadItemAsync<dynamic>("itemId", new
PartitionKey("partitionKeyValue"), options);
```

Notes:

- Think of the default consistency level as setting the mood for your data party; it sets the tone unless someone specifically decides to change the playlist.
- In the grand scheme of things, choosing the right consistency level is like picking the right pair of shoes for a hike; it needs to match the terrain of your data needs without tripping you up on performance or availability.

Guarantees of Consistency Levels in Azure Cosmos DB

Strong Consistency:

- **Guarantee:** Linearizability, ensuring reads return the most recent committed version of an item.

- **Behavior:**
  - Concurrent requests are served in real-time.
  - No uncommitted or partial writes are visible to clients.
  - Clients always see the latest write.

Bounded Staleness Consistency:

- **Guarantee:** Data lag between regions is controlled within specified bounds.
- **Configuration Options:**
  - **K Versions:** Limits how many versions behind a read can be from the latest write.
  - **T Time Interval:** Limits how far behind in time a read can be.
- **Example Configuration:**

```
ConsistencyLevel.BoundedStaleness newLevel = new
ConsistencyLevel.BoundedStaleness
{
    MaxStalenessPrefix = 1000,   // K versions
    MaxStalenessIntervalInSeconds = 300 // T time interval in seconds
};
```

- **Behavior:**
  - **Multi-Region Accounts:** Helps manage staleness across regions.
    - If the lag exceeds the specified bounds, writes are throttled until the staleness drops below the threshold.
  - **Single-Region Accounts:**
    - Mimics the behavior of Session and Eventual consistency for writes.
    - Data is replicated to a local majority within the single region.

Notes:

- **Strong Consistency:** Imagine your data as a freshly baked cookie; every bite (read) is guaranteed to be from the most recent batch (write).
- **Bounded Staleness:** Like a news broadcast with a slight delay; you're not getting live coverage, but it's still pretty fresh, controlled by how many stories (versions) or how much time (interval) you're willing to wait.

Consistency Levels in Azure Cosmos DB (Continued)

Session Consistency:

- **Guarantee:**

    - **Read-Your-Writes:** Within one session, a client sees its own writes immediately.
    - **Write-Follows-Reads:** Subsequent writes in the same session see previous reads.

- **Behavior:**

    - Assumes a single writer or shared session token for multiple writers.
    - Writes are replicated to at least three replicas locally, with asynchronous replication to other regions.

Consistent Prefix Consistency:

- **Guarantee:**

    - **Single Document Updates:** Eventual consistency.
    - **Batch Transactions:** All changes within a transaction are seen together or not at all.

- **Behavior:**

    - Provides a guarantee of monotonic reads (reads never see older data after seeing newer data).
    - **Example Scenario:**
        - If `Doc 1` and `Doc 2` are updated in transactions `T1` and `T2` respectively, reads will either see both at their latest version or at an earlier version, never a mix.

```
// Example of ensuring consistent prefix in a transaction (pseudo-code)
using Microsoft.Azure.Cosmos.Scripts;
var transaction = container.Scripts.ExecuteStoredProcedureAsync<dynamic>(
    "sprocId",
    new PartitionKey("key"),
    new dynamic[] { "Doc 1", "Doc 2" });
```

Eventual Consistency:

- **Guarantee:** No ordering of reads, but in the absence of writes, replicas will eventually converge.
- **Behavior:**
    - Weakest consistency level, where reads might not reflect the latest writes.
    - Suitable for applications where data order isn't critical, like social media counters.

Notes:

- **Session Consistency:** It's like a diary where you can write and then read your own entries right away, ensuring your personal timeline makes sense.
- **Consistent Prefix:** Like watching a movie with a friend, you'll either see the whole scene together or not at all, maintaining the story's integrity.
- **Eventual Consistency:** It's like rumors in a small town; eventually, everyone will hear about it, but not necessarily in the order it happened.

Supported APIs in Azure Cosmos DB

Overview:

- **APIs Offered:** NoSQL, MongoDB, PostgreSQL, Cassandra, Gremlin, and Table.
- **Data Models:** Supports documents, key-value, graph, and column-family models.
- **Benefits:**
    - **No Management Overhead:** Azure Cosmos DB handles scaling and management.
    - **Flexibility:** Use familiar ecosystems, tools, and skills.

API Considerations:

API for NoSQL:

- **Native:** Designed specifically for Azure Cosmos DB.
- **Use Case:** Ideal for new applications or where you want to leverage Azure Cosmos DB's native features.

API for MongoDB:

- **Compatibility:** Follows MongoDB wire protocol.
- **Best When:**
    - Migrating from MongoDB.
    - Wanting to leverage MongoDB's ecosystem without changing the application code.

API for PostgreSQL:

- **Compatibility:** PostgreSQL wire protocol.
- **Best When:**
    - Migrating from PostgreSQL.
    - Needing relational database features with distributed scaling.

API for Cassandra:

- **Compatibility:** Cassandra Query Language (CQL).
- **Best When:**
    - Migrating from Cassandra.
    - Requiring columnar data storage with high scalability.

API for Gremlin:

- **Graph Database:** Uses Gremlin query language.
- **Best When:**
    - Working with complex, interconnected data structures.
    - Existing graph database applications.

API for Table:

- **Key-Value Store:** Similar to Azure Table Storage.
- **Best When:**
    - Migrating from Azure Table Storage.
    - Needing simple key-value data model with global distribution.

Example of Choosing an API Based on Data Model (Pseudo-decision logic):

```
if (dataModel == "Document")
{
    Console.WriteLine("Choose API for NoSQL or MongoDB");
}
else if (dataModel == "Graph")
{
    Console.WriteLine("Choose API for Gremlin");
}
else if (dataModel == "Column-family")
{
    Console.WriteLine("Choose API for Cassandra");
}
else if (dataModel == "Key-Value")
{
    Console.WriteLine("Choose API for Table");
}
else if (dataModel == "Relational")
{
    Console.WriteLine("Choose API for PostgreSQL");
}
```

Notes:

- Choosing an API in Azure Cosmos DB is like deciding what genre of music to play at your data party; each has its own vibe, and you pick based on what fits the mood or existing setup of your application.
- If you're looking to keep the dance moves (code) the same, stick with the API that matches your existing partner (database).

Azure Cosmos DB APIs Overview

API for NoSQL:

- **Data Storage:** Document format.
- **Features:**
  - Full control over interface, service, and SDKs.
  - First to receive new Azure Cosmos DB features.
  - **Querying:** Supports SQL syntax for querying items.

Example Query:

```
SELECT * FROM c WHERE c.status = 'active'
```

API for MongoDB:

- **Data Storage:** Document format via BSON.

- **Compatibility:** MongoDB wire protocol without native MongoDB code.
- **Use Case:** Leverages MongoDB ecosystem while using Azure Cosmos DB features.

Example MongoDB Query:

```
db.collection.find({ status: "active" })
```

API for PostgreSQL:

- **Data Storage:** Relational or distributed data with Citus extension.
- **Configuration:** Single or multi-node for distributed scaling.

Example of Creating a Distributed Table:

```
SELECT create_distributed_table('table_name', 'distribution_column');
```

API for Apache Cassandra:

- **Data Storage:** Column-oriented schema.
- **Compatibility:** Wire protocol compatible with Apache Cassandra.
- **Philosophy:** Embraces Cassandra's approach to distributed NoSQL.

Example CQL Query:

```
SELECT * FROM keyspace.table WHERE partition_key = 'some_key';
```

API for Apache Gremlin:

- **Data Storage:** Graph database with vertices and edges.
- **Use Cases:**
  - Dynamic data with complex relations.
  - Data too intricate for relational modeling.
  - Leveraging existing Gremlin ecosystem.

Example Gremlin Query:

```
g.V('person').out('knows').valueMap(true)
```

API for Table:

- **Data Storage:** Key/value format.
- **Benefits Over Azure Table Storage:**
  - Better latency, scaling, throughput, and distribution.

- Improved index management and query performance.
- **Supports:** Only OLTP (Online Transaction Processing) scenarios.

Example of Creating a Table:

```
var account = new CloudStorageAccount(new StorageCredentials("AccountName",
"AccountKey"), true);
var client = account.CreateCloudTableClient();
var table = client.GetTableReference("myTable");
table.CreateIfNotExists();
```

Notes:

- **API for NoSQL:** Like choosing to write a novel in your own language, giving you full creative control.
- **API for MongoDB:** It's like moving to a neighborhood where you can speak the local dialect without changing your house.
- **API for PostgreSQL:** Imagine having a supercharged version of a standard car, allowing you to drive in both city streets and the autobahn of data distribution.
- **API for Cassandra:** Like organizing a library where books are chapters, allowing for efficient access to specific pages.
- **API for Gremlin:** Navigating the complex web of social connections or any intricate relationships in data, like plotting a course through a dense forest.
- **API for Table:** A straightforward move from a simple flat to a penthouse with better views and services.

Azure Cosmos DB - Request Units (RUs)

Understanding Request Units:

- **Function:** RUs represent the system resources (CPU, IOPS, memory) needed for database operations.
- **Unit of Measure:** All operations in Azure Cosmos DB are measured in RUs.

Basic RU Cost Example:

- **Point Read:**
  - Fetching a 1-KB item by ID and partition key = **1 RU**.

Charging Modes:

Provisioned Throughput Mode:

- **Throughput Provisioning:** Pay for a fixed amount of RUs per second, in 100 RU increments.

Example of Changing Throughput with Azure CLI:

```
az cosmosdb sql container throughput update \
  --throughput 500 \
  --account-name myCosmosDbAccount \
```

```
    --resource-group myResourceGroup \
    --database-name myDatabase \
    --name myContainer
```

- **Scaling:** Adjustable in 100 RU increments/decrements at any time.

Serverless Mode:

- **Billing:** Charged for the RUs consumed rather than pre-provisioned.
- **Use Case:** Ideal for unpredictable workloads or when you want to avoid upfront throughput commitments.

Autoscale Mode:

- **Scaling:** Automatically scales throughput based on usage without impacting performance.
- **Use Case:** Best for mission-critical applications with variable traffic, needing high performance and scale guarantees.

Visual Concept:

The image provided shows how different database operations consume RUs, illustrating that even though operations vary in complexity, they all are measured in RUs.

Notes:

- **RUs as Currency:** Think of RUs like energy credits in a futuristic world; every action in your database "costs" you in terms of these credits.
- **Choosing the Right Mode:**
    - **Provisioned:** Like renting a fixed amount of power for your spaceship.
    - **Serverless:** Paying only for the energy your spaceship uses during its journey.
    - **Autoscale:** Your spaceship's energy output adjusts automatically as you encounter different space environments, ensuring you never run out of juice but also not wasting it.

Working with Azure Cosmos DB

Module Overview:

- **Goal:** Learn to develop both client-side and server-side programming solutions with Azure Cosmos DB.

Key Areas to Cover:

1. **Client-Side Development:**
    - Learn how to interact with Azure Cosmos DB from your application code.
    - Understand how to use different SDKs to perform CRUD operations.

Example of Client-Side Operation with .NET SDK:

```
using Microsoft.Azure.Cosmos;

// Initialize client
var client = new CosmosClient("your-account-endpoint", "your-account-key");

// Perform a point read
var container = client.GetContainer("databaseId", "containerId");
var itemResponse = await container.ReadItemAsync<dynamic>("itemId", new
PartitionKey("partitionKeyValue"));

// Process the item
var item = itemResponse.Resource;
Console.WriteLine($"Item ID: {item.id}");
```

2. **Server-Side Development:**
    - Explore server-side logic like stored procedures, triggers, and user-defined functions (UDFs).

Example of Creating a Stored Procedure:

```
function sampleSproc() {
    var collection = getContext().getCollection();
    var response = getContext().getResponse();

    collection.createDocument(collection.getSelfLink(),
      { id: "sampleId", name: "sampleName" },
      function(err, documentCreated) {
        if (err) throw new Error('Error' + err.message);
        response.setBody('Document created successfully');
      });
}
```

3. **Data Modeling:**
    - How to design your data schema for optimal performance and scalability.
4. **Consistency Levels:**
    - Understanding and applying appropriate consistency levels for your application needs.
5. **Partitioning Strategy:**
    - Choosing the right partition key for efficient data distribution.
6. **Querying Data:**
    - Write and optimize queries for both SQL API and other supported APIs.
7. **Security and Access Control:**
    - Managing access through Azure AD, keys, or token-based authentication.

8. **Monitoring and Optimization:**
    - Use Azure Monitor and Cosmos DB metrics for performance tuning.

Reminder:

- As you progress through the module, focus on both the theoretical aspects like data modeling and practical implementations like code examples for client and server interactions.
- Keep in mind the balance between consistency, availability, and partition tolerance when designing your Azure Cosmos DB solutions. Remember, in the cosmos of databases, it's all about choosing your stars wisely to navigate your data galaxy!

Introduction to Azure Cosmos DB Programming

Learning Objectives:

Client-Side Programming:

- **Identify Classes and Methods:**
    - Understand which `CosmosClient` classes and methods are used for resource creation within the Azure Cosmos DB .NET v3 SDK.
- **Resource Creation:**
    - Learn to programmatically create Azure Cosmos DB resources like databases, containers, and items.

Example of Creating a Container:

```
using Microsoft.Azure.Cosmos;

// Initialize client
CosmosClient client = new CosmosClient("your-account-endpoint", "your-account-
key");

// Create a database
Database database = await client.CreateDatabaseIfNotExistsAsync("MyDatabase");

// Define container properties
ContainerProperties containerProperties = new ContainerProperties("MyContainer",
"/partitionKey");

// Create a container
Container container = await
database.CreateContainerIfNotExistsAsync(containerProperties, 400);
```

Server-Side Programming:

- **Stored Procedures, Triggers, and UDFs:**

- Write JavaScript code for server-side logic:
  - **Stored Procedures:** Execute transactions within the database.
  - **Triggers:** Run automatically before or after document operations.
  - **User-Defined Functions:** Custom operations within queries.

Example of a Stored Procedure:

```
function createNewItem() {
    var collection = getContext().getCollection();
    var response = getContext().getResponse();

    var newItem = {
        id: "itemId",
        name: "New Item"
    };

    collection.createDocument(collection.getSelfLink(), newItem,
        function(err, documentCreated) {
            if (err) throw new Error('Error while creating the item: ' +
err.message);
            response.setBody('Item created successfully');
        });
}
```

Implementing Change Feed:

- **Change Feed Notifications:**
  - Set up notifications to track changes made to your Azure Cosmos DB containers.

Example of Setting Up Change Feed Processor:

```
using Microsoft.Azure.Cosmos;
using Microsoft.Azure.Cosmos.ChangeFeed;

// Initialize client
CosmosClient client = new CosmosClient("your-account-endpoint", "your-account-
key");

// Set up Change Feed Processor
var database = client.GetDatabase("MyDatabase");
var leaseContainer = database.GetContainer("MyLeases");
var monitoredContainer = database.GetContainer("MyContainer");

var changeFeedProcessor = database.GetContainer("MyContainer")
    .GetChangeFeedProcessorBuilder<dynamic>("MyProcessor", HandleChanges)
    .WithInstanceName("instance1")
    .WithLeaseContainer(leaseContainer)
    .Build();

changeFeedProcessor.StartAsync().Wait();
```

```
// Define the change handler
private static async Task HandleChanges(ChangeFeedProcessorContext context,
IReadOnlyCollection<dynamic> docs, CancellationToken cancellationToken)
{
    foreach (var doc in docs)
    {
        Console.WriteLine($"Detected change for document with id: {doc.id}");
    }
}
```

Notes:

- This module acts as a gateway to understanding how to interact with Azure Cosmos DB from both sides of the fence - client applications and within the database itself.
- Think of it as learning the language of cosmos, where you can command both the stars (client-side) and the planets (server-side) to dance to your application's rhythm!

Microsoft .NET SDK v3 for Azure Cosmos DB

Key Points:

- **NuGet Package:** `Microsoft.Azure.Cosmos`
- **Terminology Shift:**
  - **Previous Versions:** Used terms like `collection` and `document`.
  - **SDK v3:** Uses `container` (collection, graph, table) and `item` (document, edge/vertex, row).

Resources:

- **GitHub Repository:** `azure-cosmos-dotnet-v3` for latest samples and solutions for CRUD operations and more.

CRUD Operations with .NET SDK v3:

Creating a CosmosClient:

- **Thread Safety:** `CosmosClient` is thread-safe, which means you should keep one instance per application lifetime for better performance.

```
using Microsoft.Azure.Cosmos;

// Initialize client with connection string
CosmosClient client = new CosmosClient("your-account-endpoint", "your-account-
key");
```

Creating a Database:

```
// Create or get database
Database database = await client.CreateDatabaseIfNotExistsAsync("MyDatabase");
```

Creating a Container:

```
// Define container properties
ContainerProperties containerProperties = new ContainerProperties("MyContainer",
"/partitionKey");

// Create container with provisioned throughput
Container container = await
database.CreateContainerIfNotExistsAsync(containerProperties, 400);
```

Item Operations:

- **Create an Item:**

```
// Sample item
var myItem = new MyItem { id = "itemId", name = "Item Name", partitionKey =
"keyValue" };

// Create the item
ItemResponse<MyItem> itemResponse = await container.CreateItemAsync(myItem);
```

- **Read an Item:**

```
// Read item by ID and partition key
ItemResponse<MyItem> readItemResponse = await container.ReadItemAsync<MyItem>
("itemId", new PartitionKey("keyValue"));
MyItem fetchedItem = readItemResponse.Resource;
```

- **Update an Item:**

```
// Update item
myItem.name = "Updated Item Name";
ItemResponse<MyItem> updateResponse = await container.ReplaceItemAsync<MyItem>
(myItem, "itemId", new PartitionKey("keyValue"));
```

- **Delete an Item:**

```
// Delete item
await container.DeleteItemAsync<MyItem>("itemId", new PartitionKey("keyValue"));
```

Note:

- The examples use the `async` and `await` pattern, which is recommended for non-blocking operations to handle the asynchronous nature of Azure Cosmos DB operations.
- Remember, like a good space traveler, your `CosmosClient` should be your trusty companion throughout your application's journey, not created anew at every turn.

Azure Cosmos DB Database and Container Operations

Database Operations:

Create a Database:

- **CreateDatabaseAsync:** Throws an exception if a database already exists.

```
Database database1 = await client.CreateDatabaseAsync(id: "adventureworks-1");
```

- **CreateDatabaseIfNotExistsAsync:** Checks for existence and creates if not present.

```
Database database2 = await client.CreateDatabaseIfNotExistsAsync(id:
"adventureworks-2");
```

Read a Database:

- Read the database from the service.

```
DatabaseResponse readResponse = await database.ReadAsync();
```

Delete a Database:

- Delete the database asynchronously.

```
await database.DeleteAsync();
```

Container Operations:

Create a Container:

- **CreateContainerIfNotExistsAsync:** Checks for existence and creates if not present.

```
// Set throughput to the minimum value of 400 RU/s
ContainerResponse simpleContainer = await
database.CreateContainerIfNotExistsAsync(
    id: "myContainer",
    partitionKeyPath: "/partitionKey",
    throughput: 400);
```

Get a Container:

- Retrieve container properties by ID.

```
Container container = database.GetContainer("myContainer");
ContainerProperties containerProperties = await container.ReadContainerAsync();
```

Delete a Container:

- Delete the container asynchronously.

```
await database.GetContainer("myContainer").DeleteContainerAsync();
```

Notes:

- **Database Creation:** Use `CreateDatabaseIfNotExistsAsync` to safely create databases without worrying about duplicates.
- **Container Throughput:** When creating, you can specify the throughput. Here, 400 RU/s is set as an example, which is the minimum allowed.
- **Asynchronous Operations:** All operations shown here are asynchronous, ensuring non-blocking calls which are crucial for maintaining application responsiveness, especially in I/O-bound scenarios like database operations.

Item Operations in Azure Cosmos DB

Creating an Item:

- Use `CreateItemAsync` to insert a new item into a container.

```
ItemResponse<SalesOrder> response = await container.CreateItemAsync(salesOrder,
new PartitionKey(salesOrder.AccountNumber));
```

Reading an Item:

- Retrieve a specific item using `ReadItemAsync` with its ID and partition key.

```
string id = "someId";
string accountNumber = "someAccountNumber";
ItemResponse<SalesOrder> response = await container.ReadItemAsync<SalesOrder>(id,
new PartitionKey(accountNumber));
```

Querying Items:

- Query items using SQL-like syntax with `GetItemQueryIterator`.

```
QueryDefinition query = new QueryDefinition(
    "select * from sales s where s.AccountNumber = @AccountInput ")
    .WithParameter("@AccountInput", "Account1");

FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
    query,
    requestOptions: new QueryRequestOptions()
    {
        PartitionKey = new PartitionKey("Account1"),
        MaxItemCount = 1
    });

while (resultSet.HasMoreResults)
{
    FeedResponse<SalesOrder> response = await resultSet.ReadNextAsync();
    foreach (var item in response)
    {
        Console.WriteLine(item.ToString());
    }
}
```

Additional Resources:

- **GitHub Repository:** Check out `azure-cosmos-dotnet-v3` for complete examples of CRUD operations and more.
- **Documentation:** Visit Azure Cosmos DB .NET V3 SDK examples for direct links to specific examples on GitHub.

Notes:

- **Partition Key:** Always include the partition key in operations as it is crucial for routing your requests efficiently.
- **Query Execution:** Remember to iterate over the `FeedIterator` to process all results, since the query might return results in batches.
- These operations showcase the power and flexibility of Azure Cosmos DB for handling data in a distributed system, making your data operations as smooth as a spacecraft gliding through the cosmos.

Azure Cosmos DB Stored Procedures

Overview:

Azure Cosmos DB supports JavaScript for executing stored procedures, triggers, and user-defined functions (UDFs) within the database, providing transactional integrity.

Writing Stored Procedures:

- **Functionality:** Stored procedures can perform CRUD operations inside a container.
- **Registration:** Must be registered per collection.

Simple Stored Procedure Example:

```
var helloWorldStoredProc = {
    id: "helloWorld",
    serverScript: function () {
        var context = getContext();
        var response = context.getResponse();

        response.setBody("Hello, World");
    }
}
```

- **Context Object:** Grants access to Azure Cosmos DB operations, request, and response objects.

Creating an Item with Stored Procedure:

- **Operations:** Asynchronous via callbacks.
- **Error Handling:** Callback includes error handling and returning the created item's ID.

```
var createDocumentStoredProc = {
    id: "createMyDocument",
    body: function createMyDocument(documentToCreate) {
        var context = getContext();
        var collection = context.getCollection();
        var accepted = collection.createDocument(collection.getSelfLink(),
              documentToCreate,
              function (err, documentCreated) {
                  if (err) throw new Error('Error' + err.message);
                  context.getResponse().setBody(documentCreated.id)
              });
        if (!accepted) return;
    }
}
```

Notes:

- **Asynchronous Nature:** Operations like creating documents are non-blocking, using callbacks for handling results or errors.
- **Error Management:** Proper error handling within the callback functions ensures that the application can respond appropriately to failures.
- **Flexibility:** Stored procedures can enforce business logic at the database level, reducing the need for round trips between client and server, thus enhancing performance by executing multiple operations within a single transaction.
- Remember, writing stored procedures in Azure Cosmos DB is like scripting the behavior of celestial bodies in your data universe, where each script can govern the lifecycle of your data items.

Arrays as Input Parameters for Stored Procedures

- **Input Conversion:** Azure Cosmos DB converts input parameters to strings when passed to stored procedures.
- **Workaround:** Parse string inputs as arrays within the stored procedure.

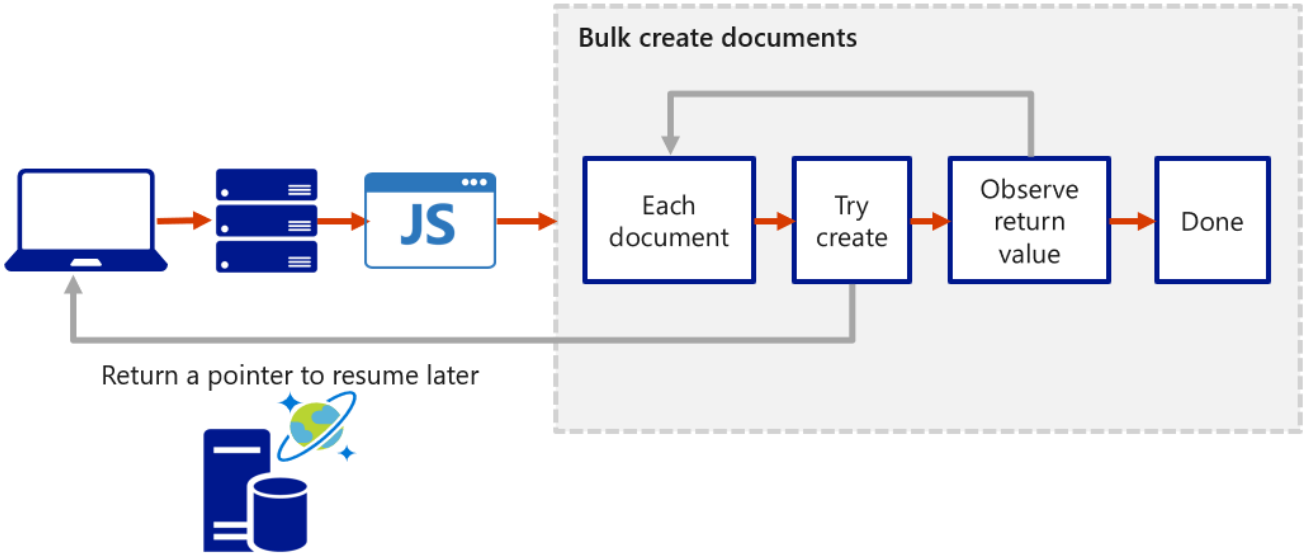
```
function sample(arr) {
    if (typeof arr === "string") arr = JSON.parse(arr);

    arr.forEach(function(a) {
        // Perform operations on each array element
        console.log(a);
    });
}
```

Bounded Execution:

- **Time Constraints:** Operations in Azure Cosmos DB must complete within a time limit.
- **Stored Procedure Execution:** Stored procedures have a server-side execution time limit.
- **Completion Indicator:** Collection functions return a Boolean value indicating if the operation completed.

Transactions within Stored Procedures:

- **Batch Processing:** Use stored procedures to process multiple items in a transaction.
- **Continuation Model:**
    - Stored procedures can implement a continuation pattern to manage long-running operations or resume execution.
    - **Continuation Value:** Can be any value to help in resuming the transaction.



Example of a Transaction with Continuation:

```
function processTransactions(continuation) {
    var context = getContext();
    var collection = context.getCollection();
    var response = context.getResponse();

    // If continuation is provided, it means we are resuming the transaction
    if (continuation) {
        // Use continuation to determine where to resume from
    } else {
        continuation = 'initialContinuation';
    }

    // Perform transactional operations here
    // ...

    // If not complete, set a continuation value
    if (!isCompleted) {
        response.setBody(continuation);
    } else {
        response.setBody("Transaction completed");
    }
}
```

Diagram Explanation:

- The diagram illustrates how a server-side function uses a continuation model:
  - Start a transaction.
  - Process a batch of work.
  - If the work isn't complete, the stored procedure returns a continuation value.
  - The client can then use this value to resume the transaction, ensuring the process repeats until completion.

Notes:

- The capability to handle input parameters as arrays allows for complex operations on collections of data within a single call, enhancing efficiency.
- Bounded execution ensures that no single operation can monopolize server resources, promoting fairness in resource allocation.
- The transaction continuation model is like planning a journey across the cosmos; you might not reach your destination in one leap, but with each continuation, you get closer until the mission is complete.

Triggers and User-Defined Functions in Azure Cosmos DB

Types of Triggers:

- **Pretriggers:** Execute before an item is modified.
- **Post-triggers:** Execute after an item is modified.
- **Execution:** Triggers are not automatic; they must be explicitly called with each relevant database operation.

Pretrigger Example:

Adding a Timestamp Pretrigger:

- **Purpose:** Ensures a timestamp is added to a newly created item if it's missing.

```
function validateToDoItemTimestamp() {
    var context = getContext();
    var request = context.getRequest();

    // Retrieve the item from the request body
    var itemToCreate = request.getBody();

    // Check if the item has a timestamp property
    if (!("timestamp" in itemToCreate)) {
        var ts = new Date();
        itemToCreate["timestamp"] = ts.getTime();
    }

    // Update the request body with the modified item
    request.setBody(itemToCreate);
}
```

- **No Input Parameters:** Pretriggers do not accept input parameters.
- **Request Object:** Used to interact with and modify the request before the operation.

Trigger Registration:

- When registering a trigger, you specify which operation it should execute with:

- **TriggerOperation.Create:** For creation operations only.
- **Note:** This trigger cannot be used for replace operations.

Additional Information:

- For detailed examples on registering and invoking triggers, refer to the official Azure Cosmos DB documentation for pretriggers.

Notes:

- **Use Case:** Triggers are perfect for ensuring data integrity, adding default values, or enforcing business rules at the time of data modification.
- **Execution Control:** By manually specifying when to execute triggers, you maintain control over performance and system load, allowing you to use triggers judiciously for critical operations.

Post-Triggers and User-Defined Functions in Azure Cosmos DB

Post-Trigger Example:

Updating Metadata Post-Trigger:

- **Purpose:** Updates a metadata document with details of a newly created item.

```
function updateMetadata() {
    var context = getContext();
    var container = context.getCollection();
    var response = context.getResponse();

    // Retrieve the newly created item
    var createdItem = response.getBody();

    // Query for the metadata document
    var filterQuery = 'SELECT * FROM root r WHERE r.id = "_metadata"';
    var accept = container.queryDocuments(container.getSelfLink(), filterQuery,
updateMetadataCallback);
    if(!accept) throw "Unable to update metadata, abort";

    function updateMetadataCallback(err, items, responseOptions) {
        if(err) throw new Error("Error" + err.message);
        if(items.length != 1) throw 'Unable to find metadata document';

        var metadataItem = items[0];

        // Update metadata
        metadataItem.createdItems += 1;
        metadataItem.createdNames += " " + createdItem.id;
        var accept = container.replaceDocument(metadataItem._self, metadataItem,
function(err, itemReplaced) {
            if(err) throw "Unable to update metadata, abort";
```

```
        });
        if(!accept) throw "Unable to update metadata, abort";
        return;
    }
}
```

- **Transactional Nature:** Post-triggers run within the same transaction as the item operation. If the trigger fails, the entire transaction fails, ensuring data consistency.

User-Defined Functions (UDFs):

Example: Income Tax Calculator:

- **Purpose:** To calculate income tax based on income brackets.
- **Sample Document Structure:**

```
{
   "name": "User One",
   "country": "USA",
   "income": 70000
}
```

- **UDF Definition:**

```
function tax(income) {
    if(income == undefined) throw 'no input';

    if (income < 1000) return income * 0.1;
    else if (income < 10000) return income * 0.2;
    else return income * 0.4;
}
```

Using UDF in Query:

```
SELECT *, udf.tax(c.income) AS taxAmount
FROM c
WHERE c.country = "USA"
```

Notes:

- **Post-Triggers:** Useful for operations that need to occur after the primary action, like logging or updating related documents.
- **User-Defined Functions:** Allow for custom logic to be executed within queries, enhancing the database's ability to perform complex calculations or transformations without needing to retrieve data

to the client for processing.
- **Error Handling:** Both examples include basic error handling, which is crucial in server-side scripts to prevent transaction failures or incorrect data handling.

Azure Cosmos DB Change Feed

Overview:

- **Functionality:** Provides a persistent record of changes to items in an Azure Cosmos DB container, ordered by occurrence.
- **Usage:** Ideal for scenarios like triggering actions based on data changes, real-time analytics, or data synchronization between systems.

Key Features:

Change Tracking:

- **Operations Tracked:** Inserts and updates are logged in the change feed.
- **Order:** Changes are output in the order they occur.

Limitations:

- **No Delete Tracking:** Deletes are not logged in the change feed.

Workaround for Deletes:

- **Soft Delete:** Implement a soft delete mechanism:
  - Add a "deleted" attribute to the item.
  - Set the value to "true".
  - Optionally, set a time-to-live (TTL) to automatically remove the item after a period.

Example of Implementing Soft Delete:

```
{
    "id": "itemId",
    "name": "Example Item",
    "deleted": true,
    "ttl": 86400 // 24 hours TTL
}
```

Processing:

- **Asynchronous:** The change feed can be processed asynchronously.
- **Incremental:** Allows for incremental processing of changes, meaning you don't need to process the entire dataset at once.
- **Distributed Consumers:** Changes can be distributed across multiple consumers for parallel processing, enhancing scalability.

Notes:

- **Use Cases:** Change feed is particularly useful for building event-driven architectures, where actions are triggered by changes in the data, similar to how a space station might react to cosmic events in real-time.
- **Considerations:** Since deletes are not tracked, implementing a soft delete strategy is crucial if you need to know when items are removed. This approach also allows for potential recovery before the actual deletion occurs.

Azure Cosmos DB Change Feed Reading

Models for Reading Change Feed:

Push Model:

- **Functionality:** The change feed processor pushes changes to a client for processing.
- **Advantages:**
  - Simplifies the process of checking for changes.
  - Manages the state of the last processed change.
  - Handles complexity like error management and load balancing.

Recommended: Preferred due to its ease of use and built-in management features.

Pull Model:

- **Functionality:** The client actively pulls changes from the server.
- **Control:**
  - Allows for reading changes from specific partition keys.
  - Provides control over the rate at which changes are processed.
  - Useful for one-time reads or data migrations.

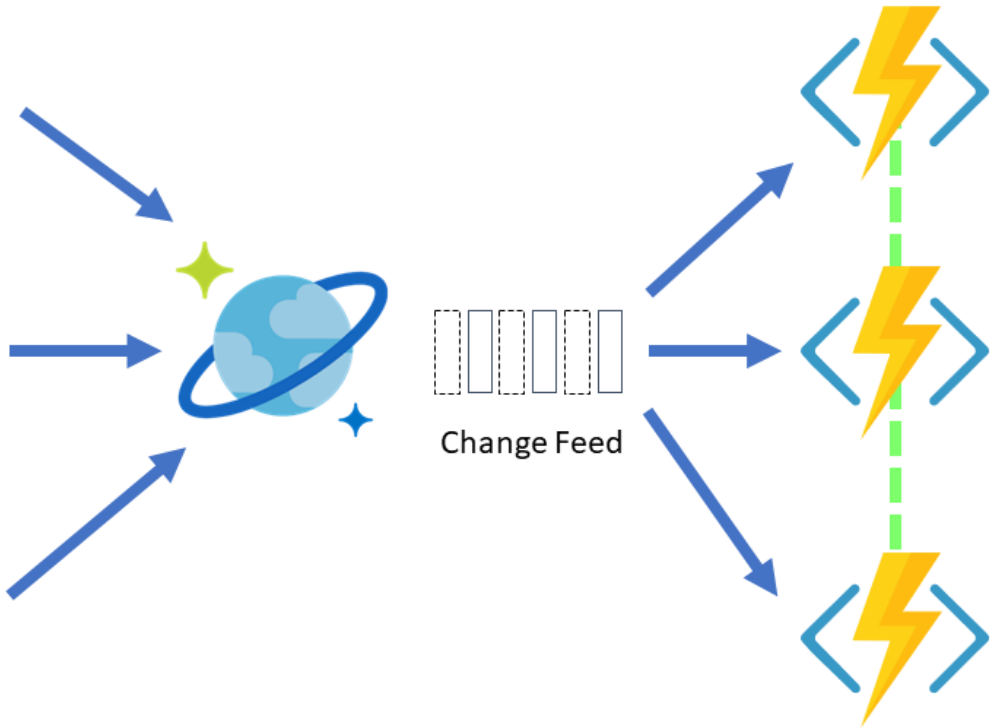
Using the Push Model:

1. Azure Functions with Cosmos DB Triggers:

- **Overview:** Automatically triggered when new events occur in your Azure Cosmos DB container.
- **Benefits:**
  - No need to manage worker infrastructure.
  - Leverages the change feed processor for scaling and reliability.
  - Parallel processing across container partitions is managed internally.

Visual Representation:

```
[Change Feed] --> [Azure Functions Trigger] --> [Function App]
```



Example of Azure Function with Cosmos DB Trigger:

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

public static void Run([CosmosDBTrigger(
    databaseName: "YourDatabase",
    collectionName: "YourContainer",
    ConnectionStringSetting = "CosmosDBConnection",
    LeaseCollectionName = "leases")]IReadOnlyList<Document> documents,
    ILogger log)
{
    if (documents != null && documents.Count > 0)
    {
        foreach (var document in documents)
        {
            log.LogInformation($"Change detected in document {document.Id}");
            // Process the document here
        }
    }
}
```

2. Change Feed Processor Library:

- **Overview:** Directly use the library to process changes without Azure Functions.
- **Benefits:**
  - Fine-tuned control over processing logic.
  - Still provides automatic scaling and state management for processing.

Notes:

- **Push vs. Pull:** Push model is generally recommended for ease and efficiency unless there's a specific need for the control offered by the pull model.
- **Azure Functions:** Acts as a hosting environment for the change feed processor, making it an effortless way to react to data changes in real-time.

Azure Cosmos DB Change Feed Processor

Components:

1. **Monitored Container:**
    - Source of the change feed where data changes are tracked.
2. **Lease Container:**
    - Stores state and coordinates change processing across multiple workers.
3. **Compute Instance:**
    - Hosts the change feed processor. Can be any compute resource like a VM, Kubernetes pod, or Azure App Service.
4. **Delegate:**
    - The business logic to process each batch of changes from the change feed.

Implementation Steps:

Initialization:

- **GetChangeFeedProcessorBuilder:** Called from a `Container` instance to set up the processor.

```
private static async Task<ChangeFeedProcessor> StartChangeFeedProcessorAsync(
    CosmosClient cosmosClient,
    IConfiguration configuration)
{
    string databaseName = configuration["SourceDatabaseName"];
    string sourceContainerName = configuration["SourceContainerName"];
    string leaseContainerName = configuration["LeasesContainerName"];

    Container leaseContainer = cosmosClient.GetContainer(databaseName,
leaseContainerName);
    ChangeFeedProcessor changeFeedProcessor =
cosmosClient.GetContainer(databaseName, sourceContainerName)
        .GetChangeFeedProcessorBuilder<ToDoItem>(processorName:
"changeFeedSample", onChangesDelegate: HandleChangesAsync)
            .WithInstanceName("consoleHost")
            .WithLeaseContainer(leaseContainer)
            .Build();
```

```
    Console.WriteLine("Starting Change Feed Processor...");
    await changeFeedProcessor.StartAsync();
    Console.WriteLine("Change Feed Processor started.");
    return changeFeedProcessor;
}
```

Delegate:

- Processes each batch of changes.

```
static async Task HandleChangesAsync(
    ChangeFeedProcessorContext context,
    IReadOnlyCollection<ToDoItem> changes,
    CancellationToken cancellationToken)
{
    Console.WriteLine($"Started handling changes for lease
{context.LeaseToken}...");
    Console.WriteLine($"Change Feed request consumed
{context.Headers.RequestCharge} RU.");

    // SessionToken if needed for Session consistency enforcement
    Console.WriteLine($"SessionToken ${context.Headers.Session}");

    // Diagnostics for long-running operations
    if (context.Diagnostics.GetClientElapsedTime() > TimeSpan.FromSeconds(1))
    {
        Console.WriteLine($"Change Feed request took longer than expected.
Diagnostics:" + context.Diagnostics.ToString());
    }

    foreach (ToDoItem item in changes)
    {
        Console.WriteLine($"Detected operation for item with id {item.id}, created
at {item.creationTime}.");
        // Asynchronous operation simulation
        await Task.Delay(10);
    }

    Console.WriteLine("Finished handling changes.");
}
```

Setup Details:

- **Processor Name:** A distinct name for the processor.
- **Instance Name:** Unique identifier for each compute instance.
- **Lease Container:** Where the processor stores its state.

Life Cycle:

1. Read changes from the change feed.
2. If there are no changes, the processor sleeps for a configurable interval.
3. If changes are detected, they are sent to the delegate.
4. After processing, the lease is updated to mark the point up to which changes have been processed.

Notes:

- The change feed processor abstracts away much of the complexity of managing the change feed, making it easier to scale and maintain your event-driven applications.
- Ensure that each compute instance has a unique instance name to avoid conflicts in lease management when scaling out.

AZ-204: Implement Containerized Solutions

Course Overview:

- Learn to deploy containerized applications using Azure services:
  - **Azure Container Registry (ACR)**: Manage and store your container images.
  - **Azure Container Instances (ACI)**: Run containers without managing servers.
  - **Azure Container Apps**: Develop, deploy, and scale containerized applications.

Prerequisites:

- **Experience:** At least one year in developing scalable solutions, covering all phases of software development.
- **Azure Knowledge:** Basic understanding of Azure services, cloud concepts, and familiarity with the Azure portal.
- **Recommended Course:** If new to Azure or cloud computing, start with AZ-900: Azure Fundamentals.

Key Takeaways:

- Understand how to push and pull container images from Azure Container Registry.
- Deploy containers quickly using Azure Container Instances for testing and development environments.
- Utilize Azure Container Apps for more robust container application management, including scaling and lifecycle management.

Next Steps:

- **Complete the modules** in the learning path to gain hands-on experience with Azure's container services.
- **Experiment** with deploying your own applications in containers to get practical experience.

Note: Remember, the universe doesn't run on Azure... or does it? Keep learning, because in the vast expanse of cloud services, knowledge is the only currency that doesn't depreciate.

Manage Container Images in Azure Container Registry

Module Overview:

- **Purpose:** Learn the essentials of using Azure Container Registry (ACR) for storing, managing, and automating container image processes.

Key Learning Outcomes:

1. **Storage and Management:**
   - Understand how to push and pull container images to/from ACR.
   - Secure your images with Azure AD authentication and role-based access control.
2. **Building Images:**
   - Use **ACR Tasks** to automate the build process.

```
# Example command to build an image using ACR Tasks
az acr build --image myimage:v1 --registry myregistry --file Dockerfile .
```

3. **Automation:**
   - Automate builds and deployments through triggers like source code updates or base image updates.

```
# Automate builds on Git commit
az acr task create --name myTask --registry myregistry --context
https://github.com/myuser/myrepo.git --git-access-token <my_token> --branch
master --file Dockerfile
```

4. **Geo-Replication:**
   - Learn about geo-replication for serving images with low latency across multiple regions.
5. **Security:**
   - Implement Docker Content Trust for signing and verifying image integrity.
6. **Integration:**
   - Integrate with Azure services like Azure Kubernetes Service (AKS) for deployment.
7. **Cost Management:**
   - Understand different tiers of ACR (Basic, Standard, Premium) and their capabilities for cost optimization.
8. **CI/CD Pipeline:**
   - How to incorporate ACR into a CI/CD pipeline for continuous deployment of containerized applications.

Note: Remember, managing container images isn't just about keeping your virtual ships afloat; it's about ensuring they're seaworthy, secure, and ready to deploy at a moment's notice. As you progress, keep in mind that in the world of containers, organization is not just key, it's the entire locksmith industry.

Introduction to Azure Container Registry (ACR)

Overview: Azure Container Registry (ACR) provides a managed, private Docker registry service, built on the open-source Docker Registry 2.0. This allows you to:

- **Store and Manage**: Private Docker container images securely in the cloud.
- **Integrate**: With other Azure services for a comprehensive container solution.

Learning Objectives:

1. **Features and Benefits of ACR:**
    - **Geo-Replication**: Serve images with low latency across regions.
    - **Security**: Integration with Azure AD for authentication and RBAC for access control.
    - **Scalability**: From Basic to Premium tiers, accommodating different needs and workloads.
    - **Integration**: Seamless integration with Azure Kubernetes Service (AKS), Azure DevOps, and more.
2. **ACR Tasks:**
    - Automate container image builds and deployments.
    - Trigger builds based on source code updates or base image updates.
    - Example command to create an ACR Task:

```
az acr task create --name myTask --registry myregistry --context
https://github.com/myuser/myrepo.git --git-access-token <my_token> --
branch master --file Dockerfile
```

3. **Dockerfile Elements:**
    - **FROM**: Specifies the base image.
    - **RUN**: Executes commands in the container to install applications or update software.
    - **COPY**: Copies new files or directories from the context into the container's filesystem.
    - **CMD**: Provides default command to run when the container starts.

```
# Example Dockerfile snippet
FROM ubuntu:latest
RUN apt-get update && apt-get install -y nginx
COPY ./index.html /var/www/html/index.html
CMD ["nginx", "-g", "daemon off;"]
```

4. **Building and Running an Image in ACR via Azure CLI:**
    - **Build an image** directly in ACR:

```
az acr build --image myimage:v1 --registry myregistry --file Dockerfile
.
```

- **Run an image**:

```
az acr run --registry myregistry myimage:v1 --cmd "echo 'Container is
running!'"
```

Conclusion: ACR isn't just another container in the ocean of cloud services; it's your private yacht where your Docker images live in luxury, securely away from public eyes, with the ability to sail smoothly across Azure's vast seas of compute resources.

Discover the Azure Container Registry

Overview: Azure Container Registry (ACR) is a managed, private Docker registry service based on Docker Registry 2.0, designed for storing and managing container images and other container-related artifacts.

Key Features:

- **Integration with Development Pipelines:**
  - Seamlessly integrates with existing container development and deployment pipelines.
  - Use ACR Tasks to build container images directly in Azure, enhancing your CI/CD practices.

Automation with ACR Tasks:

- **Trigger-Based Builds:**
  - Automatically trigger builds upon source code commits or base image updates.

```
# Example to create a task triggered on Git commit
az acr task create --name myTask --registry myregistry --context
https://github.com/myuser/myrepo.git --git-access-token <my_token> --
branch master --file Dockerfile
```

- **On-Demand Builds:**
  - Build images on demand without needing local Docker installation.

```
# Example to build an image on demand
az acr build --image webapp:v1 --registry myregistry --file Dockerfile
.
```

- **Multi-step Tasks:**

- Create tasks that involve multiple steps like building, testing, and patching container images simultaneously.

```
# Example of creating a multi-step task
az acr task create --name multistepTask --registry myregistry --file
task.yaml
```

task.yaml content:

```
version: v1.0.0
steps:
  - id: build
    cmd: docker build -t {{.Run.Registry}}/webapp:{{.Run.ID}} .
  - id: push
    cmd: docker push {{.Run.Registry}}/webapp:{{.Run.ID}}
  - id: test
    cmd: echo "Running tests"
```

Use Cases for ACR:

1. **Deployment Targets:**
   - **Scalable Orchestration:** Compatible with Kubernetes, DC/OS, Docker Swarm for managing containerized apps across clusters.
   - **Azure Services:** Integrates with AKS, App Service, Batch, and Service Fabric for large-scale deployments.
2. **Development Workflow:**
   - **Image Push:** Developers can push images to ACR from development environments or CI/CD tools like Azure Pipelines or Jenkins.

Conclusion: ACR is not just a place to park your containers; it's like a high-tech garage where your images are not only stored but also polished, tuned, and ready for a race across the Azure landscape. Whether you're automating your build pipeline or deploying at scale, ACR is your pit crew in the cloud.

Azure Container Registry Service Tiers

Overview: Azure Container Registry (ACR) provides various service tiers tailored to different needs regarding cost, storage, throughput, and feature sets:

- **Basic:**
  - **Description:** Ideal for developers exploring ACR.
  - **Features:**
    - Core capabilities like Microsoft Entra authentication, image deletion, and webhooks.
    - Suitable for lower usage scenarios with less storage and throughput.

- **Standard:**
    - **Description:** Balances cost with performance for most production environments.
    - **Features:**
        - Includes all Basic features.
        - Increased storage and image throughput for general production needs.
- **Premium:**
    - **Description:** Designed for high-volume, high-availability scenarios.
    - **Features:**
        - Highest storage capacity and concurrent operations.
        - **Enhanced Features:**
            - **Geo-Replication:** Manage a single registry across multiple regions.
            - **Content Trust:** Sign image tags for security.
            - **Private Link:** Use private endpoints for restricted registry access.

Supported Images and Artifacts:

- **Image Types:** Both Windows and Linux images can be stored.
- **Formats:**
    - **Docker Images:** Standard container images.
    - **Helm Charts:** For Kubernetes applications.
    - **OCI Images:** Images compliant with Open Container Initiative specifications.

Automated Image Builds with ACR Tasks:

- **Purpose:** Simplify the process of building, testing, pushing, and deploying container images.
- **Capabilities:**
    - Automate container patching by triggering builds when base images are updated or when code is committed to source control.

```
# Example to create an ACR Task for automatic builds
az acr task create --name myBuildTask --registry myregistry --context
https://github.com/myuser/myapp.git --file Dockerfile
```

Usage Scenarios:

- **Basic:** Learning, testing, or small-scale projects.
- **Standard:** General production environments where performance and reliability are key.
- **Premium:** For enterprises needing high throughput, multi-region deployments, or enhanced security features.

Conclusion: Choosing the right ACR tier is like picking the right spaceship for your interstellar journey. Whether you're just taking off with a small project or commanding a fleet across the stars of Azure, there's a tier that fits your mission profile, ensuring your containers have the resources they need to thrive in the cloud cosmos.

Explore Storage Capabilities of Azure Container Registry

Overview: Azure Container Registry (ACR) leverages advanced Azure storage features for security, compliance, and high availability:

Security Features:

- **Encryption-at-Rest:**
  - **Default:** All images and artifacts are encrypted before storage and decrypted when accessed.
  - **Option:** Apply additional encryption with customer-managed keys for enhanced security.

Data Residency and Compliance:

- **Regional Storage:**
  - Data is stored in the region where the registry is created.
  - Special considerations for Brazil South and Southeast Asia regions to comply with data residency laws.

High Availability and Performance:

- **Geo-Replication (Premium Tier):**
  - Replicates registry data across multiple regions for:
    - **High Availability:** Ensures access to registry data during regional outages.
    - **Performance:** Reduces latency for image pulls and pushes in global deployments.

```
# Example to enable geo-replication for a registry
az acr replication create --location "East US" --registry myregistry --
resource-group myResourceGroup
```

- **Zone Redundancy (Premium Tier):**
  - Uses Azure Availability Zones to maintain three copies of your registry data within the same region, enhancing local redundancy and availability.

Scalability:

- **Scalable Storage:**
  - Allows creation of numerous repositories, images, layers, or tags within the storage limits of your registry tier.

Performance Considerations:

- **Managing Repository Size:**
  - Large numbers of repositories, tags, and images can affect performance.
  - **Maintenance:** Regularly clean up unused items to optimize registry performance.

```
# Example to delete an unused repository
az acr repository delete --name myregistry --repository myapp --yes
```

Data Recovery:

- **Irrecoverable Deletion:** Once deleted, registry resources like repositories, images, and tags cannot be recovered. Implement a backup strategy if necessary.

Conclusion: ACR's storage capabilities are like having a Fort Knox for your digital container assets. With encryption, geo-redundancy, and zone redundancy, your data is not just stored; it's fortified. Remember to keep your registry tidy to ensure it performs like a well-oiled machine, because in the cloud, as in life, more isn't always merrier when it comes to clutter.

Build and Manage Containers with Azure Container Registry (ACR) Tasks

Overview: ACR Tasks offer a suite of features for building and managing container images in Azure:

- **Cloud-Based Building:** Suitable for Linux, Windows, and Arm platforms.
- **On-Demand Builds:** Streamline the development cycle by building containers in the cloud.
- **Automated Builds:** Trigger builds based on various events like source code updates or scheduled timers.

Task Scenarios:

1. **Quick Task:**
    - **Functionality:** Build and push a single container image to ACR without local Docker installation.
    - **Usage:** Ideal for immediate testing or validation.

```
# Example command for a quick task
az acr build --image myimage:v1 --registry myregistry --file Dockerfile
.
```

2. **Automatically Triggered Tasks:**
    - **Trigger Types:**
        - **Source Code Update:** Build when code is committed or a pull request is made.
        - **Base Image Update:** Automatically rebuild when a base image changes.
        - **Schedule:** Periodic builds based on a cron-like schedule.

```
# Example to create a task triggered on Git commit
az acr task create --name myTask --registry myregistry --context
https://github.com/myuser/myrepo.git --git-access-token <my_token> --
branch master --file Dockerfile
```

3. **Multi-Step Task:**
    - **Purpose:** Automate complex workflows involving multiple steps or containers.
    - **Benefits:** Allows for build, test, and push operations in a single task.

Source Code Context:

- ACR Tasks need a source from which to build, like:
    - Git repositories
    - Local file systems

Quick Task for Inner-Loop Development:

- **Definition:** The process of rapid code iteration, building, and testing before committing to source control.
- **Benefit:** Prevents build issues in source control by allowing developers to test builds in the cloud before committing.

```
# Example of using ACR for inner-loop development
az acr build --image myapp:dev --registry myregistry --file Dockerfile .
```

Conclusion: ACR Tasks are your automated build and deployment wizards in the cloud. They turn the mundane task of building containers into a magical, efficient process. Whether you're doing quick checks or setting up a sophisticated CI/CD pipeline, ACR Tasks are there to make your container life cycle management as smooth as the surface of an undisturbed pond in the morning. Remember, with ACR Tasks, your builds can be as quick as a flash or as reliable as the sunrise, all without cluttering your local workspace.

Trigger Task on Source Code Update

Overview: ACR Tasks can automate the building of container images or multi-step tasks in response to changes in source code, base image updates, or scheduled times:

1. Trigger on Source Code Update:

- **Functionality:** Automatically build when there's a code commit or pull request update in a Git repository.
- **Example Configuration:**

```
# Create an ACR Task to trigger on source code update
az acr task create --name mySourceTask --registry myregistry --context
https://github.com/myuser/myrepo.git --git-access-token <my_token> --branch
master --file Dockerfile
```

2. Trigger on Base Image Update:

- **Functionality:** Set up tasks to automatically rebuild application images when their base image changes.

```
# Example command to create a task dependent on base image updates
az acr task create --name myBaseTask --registry myregistry --file Dockerfile
--base-image mcr.microsoft.com/dotnet/runtime:3.1
```

3. Schedule a Task:

- **Functionality:** Schedule tasks for regular execution, useful for maintenance or periodic testing.

```
# Example to schedule a task to run daily at midnight
az acr task create --name myScheduledTask --registry myregistry --context .
--file Dockerfile --schedule "0 0 * * *"
```

Multi-Step Tasks:

- **Definition:** Use a YAML file to define complex workflows involving multiple steps of builds, tests, or deployments.

  Example YAML for a multi-step task:

```
version: v1.0.0
steps:
  - id: build-app
    cmd: docker build -t {{.Run.Registry}}/myapp .
  - id: push-app
    cmd: docker push {{.Run.Registry}}/myapp
  - id: build-test
    dependsOn: [build-app]
    cmd: docker build -t {{.Run.Registry}}/myapp-test -f Dockerfile.test .
  - id: run-test
    dependsOn: [push-app, build-test]
    cmd: docker run -d --name myapp-test {{.Run.Registry}}/myapp-test
  - id: build-helm
    dependsOn: [run-test]
    cmd: helm package -d ./helm-packages myapp
  - id: upgrade-helm
    dependsOn: [build-helm]
    cmd: helm upgrade --install -f values.yaml myapp ./helm-packages/myapp-
*.tgz
```

- **Execution:** Each step can use the resulting image from a previous step, allowing for a cohesive workflow.

Image Platforms:

- **Default:** Builds for Linux OS on AMD64 architecture.
- **Customization:**
  - Use `--platform` flag to specify different OS or architectures:
    - **Linux:** `amd64`, `arm`, `arm64`, `386`
    - **Windows:** `amd64`

```
# Example to build for ARM64v8
az acr build --image myimage:arm64v8 --registry myregistry --file Dockerfile
--platform Linux/arm64/v8 .
```

Conclusion: ACR Tasks are like your personal assistant for container builds, ensuring that your images are always up-to-date based on your codebase, your base images, or your schedule. With multi-step tasks, you can orchestrate complex workflows, making your CI/CD pipeline not just efficient but also remarkably smart, adapting to changes in your software ecosystem seamlessly.

Explore Elements of a Dockerfile

Overview: A Dockerfile is essentially a recipe for creating Docker images. It contains instructions for building the image, including:

- **Base Image:** The starting point for your container.
- **OS Updates and Software Installation:** Commands to prepare the environment.
- **Application Files:** Adding your application to the image.
- **Service Configuration:** Configuring network ports, environment variables, etc.
- **Launch Command:** What to run when the container starts.

Create a Dockerfile:

Example Dockerfile for a .NET 6 Application:

```
# Use the .NET 6 runtime as a base image
FROM mcr.microsoft.com/dotnet/runtime:6.0

# Set the working directory to /app
WORKDIR /app

# Copy the contents of the published app to the container's /app directory
COPY bin/Release/net6.0/publish/ .

# Expose port 80 to the outside world
EXPOSE 80

# Set the command to run when the container starts
CMD ["dotnet", "MyApp.dll"]
```

Explanation of Each Line:

1. `FROM mcr.microsoft.com/dotnet/runtime:6.0`
    - This specifies the base image, here it's the .NET 6 runtime environment.
2. `WORKDIR /app`
    - Sets the working directory inside the container to `/app`, where application files will be copied.
3. `COPY bin/Release/net6.0/publish/ .`
    - Copies the application's published files from the local build directory into the container's `/app` directory.
4. `EXPOSE 80`
    - Informs Docker that the container listens on the specified network ports at runtime. Here, it's port 80 for HTTP traffic.
5. `CMD ["dotnet", "MyApp.dll"]`
    - Defines the command to run when the container starts, in this case, running the .NET application with `dotnet MyApp.dll`.

Additional Notes:

- **Layering:** Each instruction in the Dockerfile creates a new layer in the image. These layers are cached which can speed up subsequent builds if only parts of the Dockerfile are changed.
- **Build Optimization:**
    - Place instructions that change often towards the end of the Dockerfile to take advantage of layer caching.
- **Security:**
    - Use multi-stage builds to reduce the size of the final image by leaving out build-time dependencies.
- **Reference:** For more detailed information, refer to the Dockerfile reference.

Conclusion: Crafting a Dockerfile is akin to writing a spell that transforms raw code into a containerized application. Each command is a step in the ritual, turning your application into a self-sufficient, deployable unit that can operate in any Docker environment. Keep in mind, the magic lies in the details, so each line needs to be precisely written to ensure your application behaves as expected inside a container.

Azure Container Instances Overview

Introduction

- **Purpose**: Understand how to deploy containers quickly with Azure Container Instances (ACI), manage environment variables, and set restart policies.

Key Concepts

- **Azure Container Instances**:
  - Simplifies the deployment of Docker containers without the need for managing underlying infrastructure like virtual machines or orchestration platforms.
  - Ideal for scenarios requiring isolated containers for simple applications, task automation, or build jobs.

Deployment Process

- **Deploying a Container:**

```
az container create --resource-group myResourceGroup --name mycontainer --
image mcr.microsoft.com/azuredocs/aci-helloworld:latest --dns-name-label
mycontainer --ports 80
```

- **Environment Variables**:
  - Setting environment variables can be done during container creation to customize the application's environment:

```
az container create --resource-group myResourceGroup --name mycontainer
--image myimage --environment-variables 'Var1=Value1' 'Var2=Value2'
```

- **Restart Policies**:
  - Define how containers should behave if they terminate:
    - **Always**: Container will keep restarting if it terminates.
    - **OnFailure**: Container restarts only if it exits with non-zero exit code.
    - **Never**: Container does not restart after termination.

```
az container create --resource-group myResourceGroup --name mycontainer --
image myimage --restart-policy OnFailure
```

Benefits of Azure Container Instances

- **Speed**: Containers can start in seconds.
- **Simplicity**: No need to manage VMs or container orchestration.
- **Isolation**: Each container runs in isolation from others.

Conclusion

- Azure Container Instances provides a serverless solution for running Docker containers, suitable for scenarios where full orchestration isn't needed, allowing for quick, on-demand deployment of

containerized applications.

This script outlines the essentials of deploying and managing containers with Azure Container Instances, focusing on deployment, environment setup, and restart policies. Remember, in the vast cosmos of cloud computing, ACI is like a small, agile spaceship - quick to launch, easy to maneuver, but not suited for long interstellar voyages where you'd need something like Kubernetes for the full space opera experience.

Introduction to Azure Container Instances (ACI)

Overview

- **What is ACI?** Azure Container Instances (ACI) provides the quickest and simplest approach to deploy containers on Azure, eliminating the need for managing virtual machines or adopting complex orchestration services.

Learning Objectives

After completing this module, you will be capable of:

- **Understanding ACI Benefits**:
  - No need for VM management.
  - No orchestration overhead.
  - Resource utilization per container group.
- **Deploying a Container**:

```
# Example command to create a container instance
az container create --resource-group myResourceGroup --name mycontainer --
image mcr.microsoft.com/azuredocs/aci-helloworld:latest --dns-name-label
mycontainer --ports 80
```

- **Managing Container Lifecycle**:
  - **Starting and Stopping Containers**:
    - Use restart policies like `Always`, `OnFailure`, or `Never` to manage container behavior post-termination.

```
# Example to set a restart policy
az container create --resource-group myResourceGroup --name mycontainer --
image myimage --restart-policy OnFailure
```

- **Environment Configuration**:
  - Setting environment variables for containers:

```
# Example to set environment variables
az container create --resource-group myResourceGroup --name mycontainer
--image myimage --environment-variables 'MY_ENV_VAR=Value'
```

- **Persistent Storage**:
    - Mounting Azure File Shares to containers for data persistence:

```
# Example command to mount an Azure File Share
az container create --resource-group myResourceGroup --name mycontainer
--image myimage --azure-file-volume-share-name mystorage --azure-file-
volume-account-name mystorageaccount --azure-file-volume-mount-path
/mnt/azureshare
```

Conclusion

Azure Container Instances stands out as an excellent service for those looking to run containerized workloads without the complexity of infrastructure management. It's akin to having a loyal robot butler for your containers, ensuring they're up and running with minimal fuss, ready to serve your applications whenever needed.

This format provides a comprehensive overview, practical examples, and a touch of humor to make learning about ACI as enjoyable as watching a good sci-fi movie where the technology just works, no questions asked.

Exploring Azure Container Instances (ACI)

Benefits of ACI

- **Fast Startup**:
    - Containers in ACI can start in mere seconds, bypassing the need for VM provisioning.
- **Direct Internet Exposure**:
    - Containers can be directly accessed from the internet via an IP and a FQDN.
- **Security**:
    - Offers hypervisor-level security, providing VM-like isolation for applications.
- **Data Handling**:
    - Minimal customer data retention to ensure container group functionality.
- **Customization**:
    - Define exact CPU cores and memory for optimal resource use.

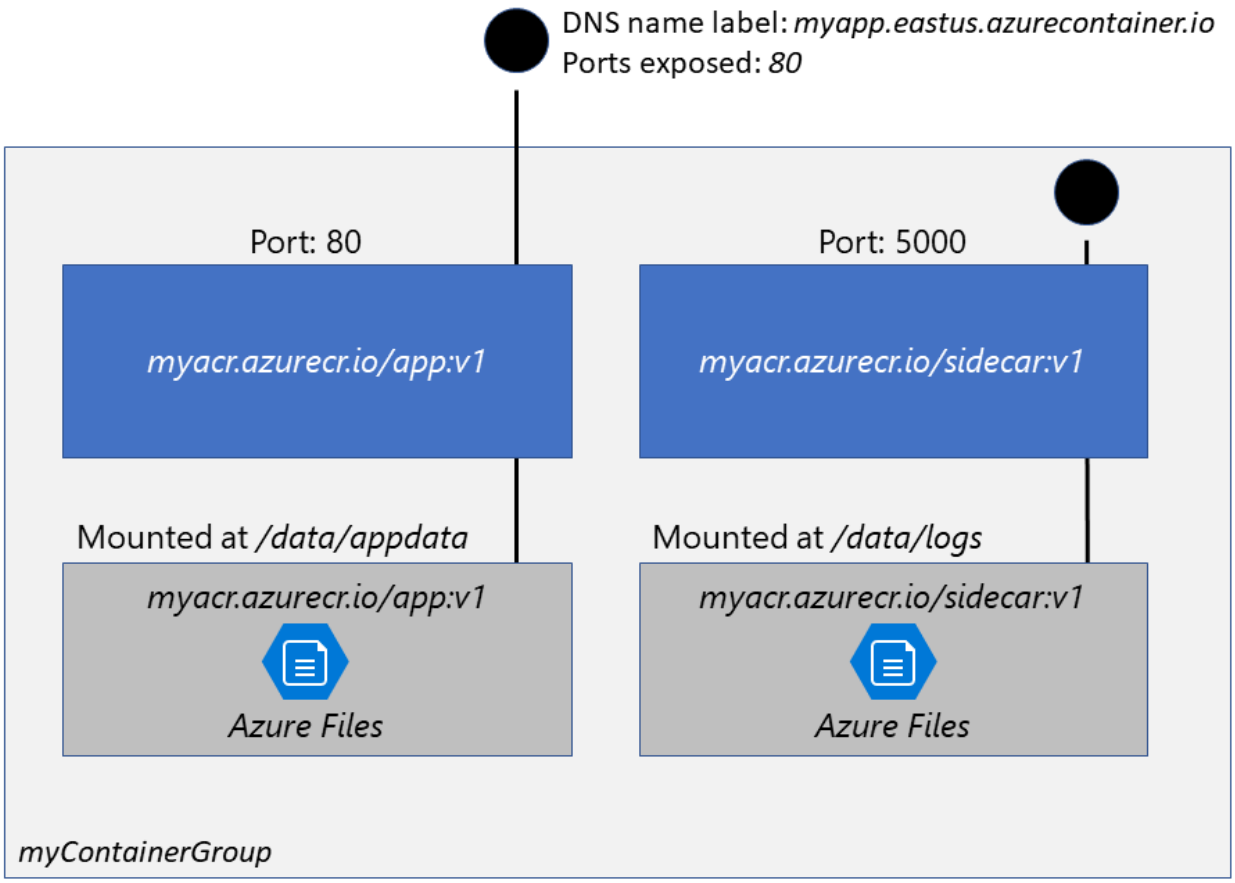
- **Storage Solutions**:
    - Integrate Azure Files for persistent data storage across container restarts.
- **Operating System Support**:
    - Supports both Linux and Windows containers, though with limitations for multi-container groups on Windows.

```
# Example to create a container with Azure Files mount
az container create --resource-group myResourceGroup --name mycontainer --image
mcr.microsoft.com/azuredocs/aci-helloworld:latest --dns-name-label mycontainer --
ports 80 --azure-file-volume-share-name myshare --azure-file-volume-account-name
mystorageaccount --azure-file-volume-mount-path /mnt/azureshare
```

Container Groups

- **Definition**:
    - A container group is the primary resource in ACI, where containers share lifecycle, resources, network, and storage.
- **Functionality**:
    - Containers within a group run on the same host, akin to Kubernetes pods.
- **Example Use Case**:
    - A container group with:
        - Two containers, each listening on different ports (80 and 5000).
        - Shared DNS name and public IP.
        - Two Azure file shares mounted for data persistence.

Diagram: (Imagine a diagram here showing two containers within a container group, each connected to different ports and sharing file storage.)



Note: Multi-container groups are currently supported only for Linux containers. For Windows, ACI supports single-instance deployments.

When to Choose ACI over AKS

- **Simple Applications**: Ideal for scenarios where full orchestration isn't necessary.
- **Complex Orchestration Needs**: For service discovery, auto-scaling, and app upgrades, consider Azure Kubernetes Service (AKS).

This format should help you understand the basics and benefits of Azure Container Instances, as well as how they differ from more complex container orchestration platforms like AKS. Remember, ACI is like using a teleportation device for your apps, swift and straightforward, while AKS is more like organizing a fleet of starships for a galactic journey.

Multi-Container Deployment in Azure Container Instances

Deployment Methods

- **Resource Manager Templates**:
  - Preferred when deploying alongside other Azure resources like Azure Files shares.

```
// Example of a simple Resource Manager Template for ACI
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "resources": [
    {
      "type": "Microsoft.ContainerInstance/containerGroups",
      "apiVersion": "2021-10-01",
      "name": "myContainerGroup",
      "location": "[resourceGroup().location]",
      "properties": {
        "containers": [
          {
            "name": "container1",
            "properties": {
              "image": "mcr.microsoft.com/azuredocs/aci-helloworld:latest",
              "ports": [
                {
                  "port": 80
                }
              ],
              "resources": {
                "requests": {
                  "cpu": 1,
                  "memoryInGB": 1
                }
              }
            }
          }
        ],
        "osType": "Linux",
        "ipAddress": {
          "type": "Public",
          "ports": [
            {
              "protocol": "TCP",
              "port": "80"
            }
          ]
        }
      }
    }
  ]
}
```

- **YAML Files**:
  - More concise, suitable for deployments involving only container instances.

```yaml
# Example YAML for ACI
apiVersion: 2018-10-01
location: eastus
name: myContainerGroup
properties:
  containers:
  - name: container1
    properties:
      image: mcr.microsoft.com/azuredocs/aci-helloworld:latest
      resources:
        requests:
          cpu: 1
          memoryInGb: 1.5
      ports:
      - port: 80
  osType: Linux
  ipAddress:
    type: Public
    ports:
    - protocol: tcp
      port: "80"
```

Resource Allocation

- **CPU, Memory, GPUs**: ACI sums up the resource requests from all containers within a group. For instance, two containers each needing 1 CPU result in a 2 CPU allocation for the group.

Networking

- **Shared IP and Port**: Containers share an IP and port namespace. External access requires port mapping on the group's IP.
- **Internal Communication**: Containers communicate via `localhost` on their respective exposed ports.

Storage

- **Volume Types**:
  - Azure File Share
  - Secret
  - Empty Directory
  - Cloned Git Repo

Common Scenarios for Multi-Container Groups

- **Separation of Concerns**: Splitting tasks like web serving from content management.
- **Logging**: A dedicated container for logging application data.
- **Monitoring**: Continuous application health checks.
- **Frontend-Backend Separation**: Enhancing modularity in application architecture.

These notes provide an overview of deploying multi-container groups in Azure Container Instances, emphasizing the deployment methods, resource management, networking specifics, and practical use cases. Remember, in the world of Azure, deploying containers is like assembling a band - each container plays its part, but together they create a symphony of functionality.

Running Containerized Tasks with Azure Container Instances

Advantages

- **Ease and Speed**: Quick deployment of containers for tasks like builds, tests, or rendering.
- **Cost Efficiency**: Pay only for the compute time the container is actively running.

Container Restart Policies

In Azure Container Instances, you can configure how containers behave upon completion or failure through three restart policies:

- **Always**
    - **Description**: Containers will always restart, regardless of exit status.
    - **Default**: Applied if no policy is specified.
- **Never**
    - **Description**: Containers never restart, they run once and then stop.
- **OnFailure**
    - **Description**: Containers restart only if they exit with a non-zero exit code, indicating a failure.

Specifying a Restart Policy

To set a restart policy, use the `--restart-policy` parameter with the `az container create` command:

```
# Example command to create a container with an OnFailure restart policy
az container create \
    --resource-group myResourceGroup \
    --name mycontainer \
    --image mycontainerimage \
    --restart-policy OnFailure
```

Run to Completion

- **Process**:
    - Azure Container Instances starts the container.
    - The container runs the specified application or script.
    - Upon completion, if the policy is `Never` or `OnFailure`, the container stops and its status is marked as **Terminated**.

Note: This setup is ideal for tasks that should run once, like:

- Building software
- Running tests
- Rendering images

Remember, in the cosmos of container management, choosing the right restart policy is like deciding whether your spaceship should auto-pilot back to base or gracefully drift into the void after its mission.

Setting Environment Variables in ACI

Overview

- **Purpose**: Environment variables provide a way to dynamically configure your container's application or script at runtime, similar to using `--env` with `docker run`.

Basic Usage

To set environment variables when creating a container:

```
# Example command to create a container with environment variables
az container create \
    --resource-group myResourceGroup \
    --name mycontainer2 \
    --image mcr.microsoft.com/azuredocs/aci-wordcount:latest \
    --restart-policy OnFailure \
    --environment-variables 'NumWords'='5' 'MinLength'='8'
```

Note: For Windows Command Prompt, use double quotes:

```
az container create --environment-variables "NumWords"="5" "MinLength"="8"
```

Secure Environment Variables

- **Purpose**: For passing sensitive information like passwords or API keys securely.
- **How to Set**:
    - Use the `secureValue` property in YAML for sensitive variables.

```
# Example YAML for setting both regular and secure environment variables
apiVersion: 2018-10-01
location: eastus
name: securetest
properties:
  containers:
```

```
    - name: mycontainer
      properties:
        environmentVariables:
          - name: 'NOTSECRET'
            value: 'my-exposed-value'
          - name: 'SECRET'
            secureValue: 'my-secret-value'
        image: nginx
        ports: []
        resources:
          requests:
            cpu: 1.0
            memoryInGB: 1.5
    osType: Linux
    restartPolicy: Always
  tags: null
  type: Microsoft.ContainerInstance/containerGroups
```

- **Security**: Secure values are not visible in container properties in the Azure portal or CLI, only accessible within the container.

Deployment with Secure Variables

Deploy the container group using the above YAML:

```
# Command to deploy container group using YAML file with secure environment
variables
az container create --resource-group myResourceGroup --file secure-env.yaml
```

Note: Remember, setting environment variables in containers is like whispering secrets to your application; use secure values when you don't want the whole universe to hear what you're saying.

Azure Container Instances with Azure Files

Purpose

- **State Persistence**: Azure Container Instances are stateless by default. Mounting external storage like Azure Files ensures data persistence beyond container lifecycle.

Azure Files Integration

- **Service**: Azure Files provides managed file shares in the cloud using SMB protocol.
- **Functionality**: Similar file-sharing capabilities as with Azure VMs.

Limitations

- **OS Support**: Only Linux containers can mount Azure Files shares.

- **Permissions**: The container must run as root to mount the volume.
- **Protocol**: Limited to Common Internet File System (CIFS) support.

Deployment with Azure File Share

To mount an Azure file share in a container:

```
# Example command to create a container with an Azure file share volume
az container create \
    --resource-group $ACI_PERS_RESOURCE_GROUP \
    --name hellofiles \
    --image mcr.microsoft.com/azuredocs/aci-hellofiles \
    --dns-name-label aci-demo \
    --ports 80 \
    --azure-file-volume-account-name $ACI_PERS_STORAGE_ACCOUNT_NAME \
    --azure-file-volume-account-key $STORAGE_KEY \
    --azure-file-volume-share-name $ACI_PERS_SHARE_NAME \
    --azure-file-volume-mount-path /aci/logs/
```

Note on DNS Name Label:

- Ensure the `--dns-name-label` is unique within the Azure region to avoid conflicts. Modify if necessary to achieve uniqueness.

Explanation:

- **--azure-file-volume-account-name**: The name of the storage account.
- **--azure-file-volume-account-key**: The storage account key for authentication.
- **--azure-file-volume-share-name**: The name of the Azure file share.
- **--azure-file-volume-mount-path**: The path within the container where the share will be mounted.

This setup allows your container to read/write data to/from an Azure file share, keeping your application's state persistent across container restarts or crashes. Remember, using Azure Files with ACI is like giving your container a persistent memory in the cloud, ensuring it doesn't forget its past lives.

Deploying Container with Volume Mounts - YAML

Single Container with Azure File Share

- **Overview**: Deploying via YAML is preferred for multi-container groups or when you need to mount volumes.
- **Example YAML for Single Container**:

```
apiVersion: '2019-12-01'
location: eastus
name: file-share-demo
properties:
```

```
  containers:
  - name: hellofiles
    properties:
      environmentVariables: []
      image: mcr.microsoft.com/azuredocs/aci-hellofiles
      ports:
      - port: 80
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
      volumeMounts:
      - mountPath: /aci/logs/
        name: filesharevolume
  osType: Linux
  restartPolicy: Always
  ipAddress:
    type: Public
    ports:
      - port: 80
    dnsNameLabel: aci-demo
  volumes:
  - name: filesharevolume
    azureFile:
      sharename: acishare
      storageAccountName: <Storage account name>
      storageAccountKey: <Storage account key>
tags: {}
type: Microsoft.ContainerInstance/containerGroups
```

Mounting Multiple Volumes

- **Scenario**: For mounting more than one volume, YAML or Azure Resource Manager templates are used.
- **Example for Multiple Volumes in YAML or JSON**:

```
"volumes": [{
  "name": "myvolume1",
  "azureFile": {
    "shareName": "share1",
    "storageAccountName": "myStorageAccount",
    "storageAccountKey": "<storage-account-key>"
  }
},
{
  "name": "myvolume2",
  "azureFile": {
    "shareName": "share2",
    "storageAccountName": "myStorageAccount",
    "storageAccountKey": "<storage-account-key>"
```

```
  }
}]
```

- **Mounting these Volumes in Containers**:

```
"volumeMounts": [{
  "name": "myvolume1",
  "mountPath": "/mnt/share1/"
},
{
  "name": "myvolume2",
  "mountPath": "/mnt/share2/"
}]
```

Note:

- Replace `<Storage account name>` and `<storage-account-key>` with actual values from your storage account.
- Ensure that you specify unique volume names and mount paths for each share to avoid conflicts within the container.

Using YAML or JSON templates for deployment provides a structured way to manage container configurations, especially when dealing with multiple resources or complex setups like mounting various volumes. It's like giving your containers a set of detailed instructions on where to find their belongings in their new digital home.

Introduction to Azure Container Apps

Overview

- **What is Azure Container Apps?**
  - A serverless container service designed for microservices, focusing on simplicity, robust autoscaling, and minimal infrastructure management.

Learning Objectives

After this module, you will understand:

- **Features and Benefits**:
  - **Serverless Architecture**: Focus on your code, not the infrastructure.
  - **Autoscaling**: Scale based on demand effortlessly.
  - **Microservice Support**: Ideal for applications broken down into smaller, independent components.
- **Deployment**:
  - **Azure CLI Deployment**:

```
# Example command to deploy a container app
az containerapp up --name mycontainerapp --resource-group myResourceGroup --image
mcr.microsoft.com/azuredocs/aca-helloworld:latest
```

- **Authentication & Authorization**:
    - Azure Container Apps includes built-in mechanisms for authentication which can be configured to use Azure Active Directory or other identity providers.
- **Revisions and Secrets**:
    - **Revisions**: Manage different versions of your application without downtime.
        - Create a new revision:

```
# Command to create a new revision of a container app
az containerapp revision create --name mycontainerapp --resource-group
myResourceGroup --image newimage:tag
```

- **Secrets Management**: Implement app secrets securely.
    - Add a secret:

```
# Example of adding a secret to a container app
az containerapp secret set --name mycontainerapp --resource-group myResourceGroup
--secret-name MySecret --secret-value secretvalue
```

Key Points

- Azure Container Apps simplifies the deployment and management of containerized applications, focusing on developer productivity by abstracting away the underlying complexities of container orchestration and infrastructure management.
- The service's integration with Azure's ecosystem allows for easy use with other Azure services, enhancing security, scalability, and operational efficiency.

Remember, Azure Container Apps is like having a team of highly efficient, invisible assistants in the cloud, taking care of your containers so you can focus on what really matters - your application's functionality and features.

Exploring Azure Container Apps

Overview

- **Platform**: Azure Container Apps runs on Azure Kubernetes Service, offering serverless capabilities for containerized applications and microservices.

Common Uses

- Deploying API endpoints
- Hosting background processing applications
- Event-driven processing
- Microservices architecture

Dynamic Scaling

Azure Container Apps can scale based on:

- HTTP traffic
- Event-driven triggers
- CPU or memory load
- Any KEDA-supported scaler

Note: Applications scaling on CPU or memory usage cannot scale to zero.

Key Features

- **Revisions and Lifecycle Management**:
  - Manage multiple revisions of your application for easy updates and rollbacks.
- **Autoscaling**:
  - Automatically scale your application based on demand.

```
# Example to set up autoscaling rules for a container app
az containerapp update --name mycontainerapp --resource-group
myResourceGroup --set "template.scaleRules=
[{'name':'http','http':'/api/health','auth':[]}]"
```

- **HTTPS Ingress**:
  - Enable secure access without managing additional Azure infrastructure.
- **Traffic Management**:
  - Implement blue/green deployments or A/B testing by splitting traffic.
- **Service Discovery and Internal Ingress**:
  - Secure communication between services with DNS-based discovery.
- **Integration with Dapr**:
  - Build microservices with simplified service-to-service interactions using Dapr.
- **Container Image Sources**:

    - Support for containers from any registry, like Docker Hub or Azure Container Registry (ACR).

- **Deployment and Management**:
    - **Azure CLI**:

```
# Example to create a container app
az containerapp up --name mycontainerapp --resource-group
myResourceGroup --image mcr.microsoft.com/azuredocs/aca-
helloworld:latest
```

    - **Azure Portal** or **ARM Templates** can also be used.

- **Networking**:
    - Integrate with existing virtual networks for secure deployment.

- **Secrets Management**:
    - Securely store and manage application secrets.

- **Monitoring**:
    - Use Azure Log Analytics for logging and monitoring.

Conclusion: Azure Container Apps offers a robust, serverless environment for running microservices and containerized applications with features tailored for modern application development, focusing on scalability, security, and simplicity. Remember, with Azure Container Apps, you're essentially giving your applications the ability to grow or shrink like a magic beanstalk, except this one doesn't need water, just traffic and events.

Azure Container Apps Environments

Overview

- **Container Apps Environment**: A secure boundary where individual container apps are deployed.
    - **Virtual Network**: Apps within the same environment share a virtual network.
    - **Log Analytics**: They also share the same Log Analytics workspace for logging.

Reasons for Same Environment Deployment

- **Service Management**: When services need to be managed together.
    - Example: A web app and its backend services.

- **Network Consistency**: Deploying applications that need to communicate within the same network.
    - Example: For internal communication without public exposure.

- **Dapr Usage**: For Dapr-enabled applications that use the Dapr service invocation API for communication.

```
# Example deployment command with Dapr
az containerapp up --name mydaprapp --resource-group myResourceGroup --image
mydaprimage --dapr-app-id mydaprapp --dapr-app-port 3000 --dapr-components
mydaprcomponents.yaml
```

- **Shared Dapr Configuration**: When multiple apps need to share Dapr configuration settings.
- **Shared Logging**: When apps need to share the same logging mechanism for easier monitoring.

Reasons for Different Environment Deployment

- **Resource Isolation**: To ensure applications do not share compute resources.
    - Example: For applications that need dedicated resources or for security reasons.
- **Dapr Isolation**: When you want to prevent Dapr service-to-service communication between apps.

Microservices with Azure Container Apps

Microservices allow for:

- **Independent Lifecycle Management**: Develop, upgrade, version, and scale each service independently.
- **Service Discovery**: Automatically discover and communicate with other services.
- **Dapr Integration**: Enhances microservices with:
    - **Observability**
    - **Pub/Sub messaging**
    - **Service-to-Service Invocation** with features like mutual TLS, retries, etc.

Dapr Benefits for Microservices:

- **Standardizes Communication**: Provides a consistent way for services to interact, handling distributed system complexities like:
    - **Failures**
    - **Retries**
    - **Timeouts**
- **Rich Programming Model**: Simplifies the development of microservices by abstracting away many of the complexities associated with distributed systems.

Remember, deploying container apps in Azure Container Apps environments is like setting up neighborhoods in a city; each neighborhood (environment) has its own rules, utilities, and community vibe, allowing services to interact closely or remain isolated as needed for optimal operation.

Exploring Containers in Azure Container Apps

Overview

- **Managed Kubernetes**: Azure Container Apps abstracts Kubernetes management, allowing focus on application development rather than infrastructure.
- **Flexibility**: Supports any Linux-based x86-64 container image, with no mandatory base image requirement.
    - **Automatic Recovery**: Containers automatically restart if they crash.

Container Configuration

- **Using ARM Templates**: Configuration for containers within Azure Container Apps can be defined in the `containers` array of the `properties.template` section in an ARM template. Here's how you might configure a container:

```
"containers": [
  {
    "name": "main",
    "image": "[parameters('container_image')]",
    "env": [
      {
        "name": "HTTP_PORT",
        "value": "80"
      },
      {
        "name": "SECRET_VAL",
        "secretRef": "mysecret"
      }
    ],
    "resources": {
      "cpu": 0.5,
      "memory": "1Gi"
    },
    "volumeMounts": [
      {
        "mountPath": "/myfiles",
        "volumeName": "azure-files-volume"
      }
    ],
    "probes": [
      {
        "type": "liveness",
        "httpGet": {
          "path": "/health",
          "port": 8080,
          "httpHeaders": [
            {
              "name": "Custom-Header",
              "value": "liveness probe"
            }
          ]
        },
        "initialDelaySeconds": 7,
        "periodSeconds": 3
```

```
          // The file might continue with more configuration options
        }
      ]
    }
  ]
```

- **Key Configuration Elements**:
  - **name**: The name of the container within the app.
  - **image**: The container image to use, parameterized for flexibility.
  - **env**: Environment variables, including secrets referenced by `secretRef`.
  - **resources**: CPU and memory allocation for the container.
  - **volumeMounts**: Mounting external volumes like Azure Files inside the container.
  - **probes**: Health checks like liveness probes to ensure the container is running correctly.

Revision Snapshots

- **Dynamic Updates**: Changes in the ARM template configuration result in a new revision of the container app, allowing for seamless updates without downtime.

Diagram: (Imagine a diagram here illustrating how containers for an Azure Container App are grouped in pods inside revision snapshots.)

Azure Container Apps provides a robust environment for running containers with detailed control over resources, environment, and health monitoring, making it an excellent choice for applications requiring flexibility and managed orchestration.

Multiple Containers in Azure Container Apps

- **Sidecar Pattern**: Multiple containers can be defined in one container app for implementing patterns like:
  - **Logging Agents**: To read and forward logs from the primary app.
  - **Cache Refreshers**: Background processes that update caches on shared volumes.
- **Shared Resources**: Containers within the same app share:
  - **Disk**: For data sharing.
  - **Network**: For communication.
  - **Lifecycle**: All containers in the app go through lifecycle events together.

Note: While possible, this is an advanced scenario. For microservices, it's generally better to deploy each service as its own container app.

- **Example Configuration**:

```
"containers": [
  {
    "name": "main",
    "image": "mcr.microsoft.com/azuredocs/aca-helloworld:latest",
```

```
    // ... other configurations ...
  },
  {
    "name": "sidecar",
    "image": "myregistry.azurecr.io/sidecar:latest",
    // ... configurations for sidecar container ...
  }
]
```

Container Registries

- **Private Registries**: To pull images from private registries:

```
{
  ...
  "registries": [{
    "server": "docker.io",
    "username": "my-registry-user-name",
    "passwordSecretRef": "my-password-secret-name"
  }]
}
```

- **Fields**:
  - **server**: The registry server address.
  - **username**: The registry username.
  - **passwordSecretRef**: Refers to a secret where the password is stored.
- **Purpose**: Allows Azure Container Apps to authenticate and pull images from private registries during deployment.

Limitations

- **Privileged Containers**: Not supported, causing runtime errors if root access is attempted.
- **OS Support**: Only Linux-based (linux/amd64) images are supported.

Remember, using Azure Container Apps to manage multiple containers or private registries is like orchestrating a band; each instrument (container) plays its role, supported by the rhythm section (the infrastructure), but sometimes you need the whole ensemble (separate apps) for the full symphony of microservices.

Authentication and Authorization in Azure Container Apps

Overview

- **Purpose**: Azure Container Apps offers built-in features for authentication and authorization, simplifying security for your containerized applications.
- **Benefits**:

    - No need for extensive coding or deep security knowledge.
    - Supports federated identity providers for user authentication.

Security Considerations

- **HTTPS Requirement**: Ensure your container app uses HTTPS by disabling `allowInsecure` in the ingress configuration.

Configuration Options

- **Restrict Access**:
    - **Require Authentication**: Only authenticated users can access the app.
    - **Allow Unauthenticated Access**: Authentication is available but not mandatory for accessing content.

Identity Providers Supported

Azure Container Apps supports integration with various identity providers for authentication:

Provider	Sign-in Endpoint	Documentation
Microsoft Identity Platform	`/.auth/login/aad`	[Microsoft Identity Platform]
Facebook	`/.auth/login/facebook`	[Facebook]
GitHub	`/.auth/login/github`	[GitHub]
Google	`/.auth/login/google`	[Google]
X (formerly known as Twitter)	`/.auth/login/twitter`	[X]
Any OpenID Connect provider	`/.auth/login/<providerName>`	[OpenID Connect]

- **Functionality**:
    - Each provider's sign-in endpoint is used for authenticating users and validating tokens.
    - You can offer multiple sign-in options for users.

Usage

- **Setup**:
    - Configure your container app in the Azure portal or via Azure CLI to use one or more of these providers.
    - When setting up, you'll need to provide the necessary credentials or configuration details for each provider.

Note: This built-in feature streamlines the process of securing your application by offloading the authentication to trusted identity providers, allowing developers to concentrate on application logic rather than security implementation details. Remember, in the digital world, this is like having a trusted bouncer at the door of your app party, letting in only those with the right credentials.

Feature Architecture for Authentication in Azure Container Apps

Overview

- **Middleware**: Authentication and authorization in Azure Container Apps are handled by a middleware component, which operates as a sidecar container on each replica of your application.
- **Security Layer**: Every HTTP request to your app goes through this security layer first.

Diagram: (Imagine a diagram here showing the flow of requests through the middleware sidecar before reaching the app container.)

Middleware Responsibilities

1. **User and Client Authentication**: Manages authentication with specified identity providers.
2. **Session Management**: Handles authenticated sessions.
3. **Identity Information**: Injects user identity data into HTTP request headers for app consumption.

Isolation

- **Security Container**: Runs separately from the application container, ensuring no direct code integration is necessary. This isolation means no language-specific frameworks are required for security features.

Authentication Flows

- **Server-Directed Flow (Without Provider SDK)**:
  - **Process**: The application delegates the sign-in process to Azure Container Apps.
  - **Use Case**: Typically used in browser applications where the user is redirected to the identity provider's sign-in page.
  - **Example Flow**:
    1. User attempts to access a protected resource.
    2. Redirect to the provider's login page (`/.auth/login/<provider>`).
    3. After authentication, the provider redirects back with an authentication token.
    4. Container Apps validates the token and creates a session.
- **Client-Directed Flow (With Provider SDK)**:
  - **Process**: The app manages authentication directly with the provider's SDK and then validates the token with Container Apps.
  - **Use Case**: Common in non-browser applications like mobile apps where direct integration with the provider's SDK is possible.
  - **Example Flow**:
    1. The mobile app uses the provider’s SDK to authenticate the user.
    2. After successful login, the app obtains the authentication token.
    3. The token is sent to Container Apps for validation.
    4. Container Apps verifies the token and authorizes the request.

Note: These flows illustrate how Azure Container Apps can work seamlessly with external identity providers, providing flexibility in how applications handle user authentication and authorization.

Authentication and Authorization in Azure Container Apps

Overview

- **Built-in Features**: Azure Container Apps offers native authentication and authorization, simplifying security implementations.
- **Benefits**:
  - No need for coding or deep security knowledge.
  - Integrates with federated identity providers for seamless user authentication.

Security Recommendations

- **HTTPS**: Essential for secure communication. Always ensure `allowInsecure` is set to `false` in your ingress configuration.

Configuration Options

- **Restrict Access**:
  - **Require Authentication**: Access to the app or APIs is restricted to authenticated users only.
  - **Allow Unauthenticated Access**: Users can access content without authentication, but authentication information is still available for app logic.

Supported Identity Providers

Azure Container Apps supports integration with several identity providers:

Provider	Sign-in Endpoint	How-To Guidance
Microsoft Identity Platform	`/.auth/login/aad`	[Microsoft Identity Platform]
Facebook	`/.auth/login/facebook`	[Facebook]
GitHub	`/.auth/login/github`	[GitHub]
Google	`/.auth/login/google`	[Google]
X (formerly known as Twitter)	`/.auth/login/twitter`	[X]
Any OpenID Connect provider	`/.auth/login/<providerName>`	[OpenID Connect]

Usage

- **Setup**:
  - In the Azure portal or through Azure CLI, configure your container app to use these providers. You'll need to provide credentials or client IDs/secrets for each provider you want to integrate.
  - Each provider will have its own setup process, but they all follow a similar pattern where you define the provider in the app's authentication settings.

Note: This integration with built-in authentication providers allows developers to leverage established authentication services, enhancing security while reducing the development overhead. It's like having a security guard at your digital door, ensuring only verified guests (users) get in.

Feature Architecture for Authentication in Azure Container Apps

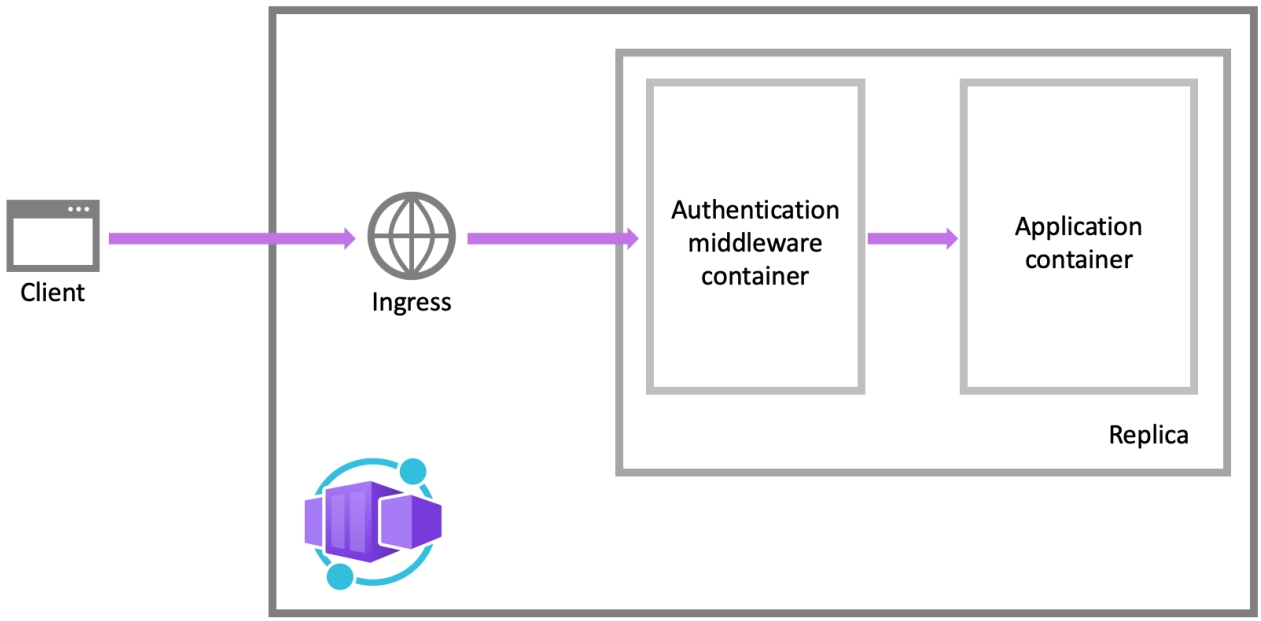
Middleware Component

- **Sidecar Container**:
  - The authentication and authorization middleware operates as a separate sidecar container alongside your application container on each replica.
  - **Responsibilities**:
    - **User Authentication**: Verifies user identity using specified identity providers.
    - **Session Management**: Handles session data for authenticated users.
    - **Header Injection**: Adds identity-related information to HTTP request headers for the application to use.

Security Layer

- **Request Interception**: All incoming HTTP requests are first processed by the sidecar container, ensuring security checks are performed before the request reaches the app.

Diagram: (Visualize a diagram here depicting how requests are intercepted and processed by the middleware before reaching the application container.)



Isolation

- **Separate Process**: The security container runs independently of your application, meaning:
  - No need for in-process integration with language-specific frameworks.
  - Security logic is handled outside your app's runtime environment.

Authentication Flows

- **Server-Directed Flow (Without Provider SDK)**:

- **Process**:
  1. User attempts to access a protected resource.
  2. Redirect to the provider's login page (e.g., `/.auth/login/<provider>`).
  3. After authentication, the user is redirected back to your app with an authentication token.
  4. The middleware validates the token and manages the session.
- **Use Case**: Ideal for web applications where the user interacts with the provider's login page directly.

- **Client-Directed Flow (With Provider SDK)**:
  - **Process**:
    1. The application (e.g., a mobile app) uses the provider's SDK to authenticate the user.
    2. After authentication, the app receives a token.
    3. This token is sent to Container Apps for validation.
    4. Upon validation, the middleware processes the request and passes it along with necessary headers to the app.
  - **Use Case**: Suitable for applications where the provider's UI isn't shown, like native mobile apps or backend services.

Note: These flows demonstrate how Azure Container Apps can adapt to different application scenarios, providing a versatile approach to managing user authentication and enhancing security without requiring extensive code modifications.

Revisions in Azure Container Apps

Overview

- **Revisions**: Represents an immutable snapshot of your container app's state. Useful for:
  - **Versioning**: Deploying new versions or reverting to previous ones.
  - **Traffic Control**: Managing which revisions are active and how traffic is directed.

Revision Naming

- **Default Naming**: Azure automatically appends a unique, semi-random suffix to the app name (e.g., `album-api--1st-revision`).
- **Customizing Suffix**: You can set a custom suffix for better organization or identification:
  - **ARM Templates**
  - **Azure CLI**:

```
az containerapp create --name album-api --resource-group
myResourceGroup --revision-suffix 1st-revision
```

or

```
az containerapp update --name album-api --resource-group
myResourceGroup --revision-suffix 2nd-revision
```

Updating Container Apps

- **Using Azure CLI**: To update and potentially create a new revision with changes:

```
az containerapp update \
  --name <APPLICATION_NAME> \
  --resource-group <RESOURCE_GROUP_NAME> \
  --image <IMAGE_NAME>
```

    - **What can be updated**: Environment variables, compute resources, scale settings, and the container image.

- **Revision-Scope Changes**: Changes that trigger a new revision include:
    - Modifying container configurations within the allowed parameters.

Listing Revisions

- To view all revisions:

```
az containerapp revision list \
  --name <APPLICATION_NAME> \
  --resource-group <RESOURCE_GROUP_NAME> \
  -o table
```

Note: Managing revisions in Azure Container Apps is akin to having a version control system for your deployed applications, allowing for controlled releases and rollbacks. Each revision acts like a save point in your application's journey, ensuring you can navigate through different stages of its evolution with ease.

Secrets Management in Azure Container Apps

Overview

- **Purpose**: Securely store and manage sensitive configuration data like connection strings or API keys.
- **Scope**: Secrets are application-level, not revision-specific, allowing:
    - **Consistency Across Revisions**: Secrets can be used by multiple revisions without changes triggering new revisions.
    - **Flexibility**: Secrets can be updated or removed without immediate impact on existing revisions.

Key Points

- **No New Revisions**: Adding, removing, or updating secrets does not automatically create new revisions.
- **Secret Updates**: Changes to secrets require either deploying a new revision or restarting an existing one to take effect.

- **Secret Removal**: Ensure no revisions reference a secret before deleting it to avoid application errors.

Note: Azure Container Apps does not integrate with Azure Key Vault directly; instead, use managed identity and the Key Vault SDK for secret management.

Defining and Using Secrets

- **Creating Secrets**: Use the `--secrets` parameter during app creation or update:

```
az containerapp create \
  --resource-group "my-resource-group" \
  --name queuereader \
  --environment "my-environment-name" \
  --image demos/queuereader:v1 \
  --secrets "queue-connection-string=$CONNECTION_STRING"
```

  - **Example**: Here, `$CONNECTION_STRING` is an environment variable containing the actual secret value.

- **Referencing Secrets in Environment Variables**:

```
az containerapp create \
  --resource-group "my-resource-group" \
  --name myQueueApp \
  --environment "my-environment-name" \
  --image demos/myQueueApp:v1 \
  --secrets "queue-connection-string=$CONNECTIONSTRING" \
  --env-vars "QueueName=myqueue" "ConnectionString=secretref:queue-
connection-string"
```

  - **Syntax**: Use `secretref:` followed by the secret name to reference it in an environment variable.

Note: Secrets in Azure Container Apps provide a secure way to pass sensitive information into your applications, ensuring that even in a cloud environment, your secrets remain, well, secret. Remember, managing secrets is like managing the keys to your digital kingdom; you want to ensure they're well-protected and distributed only to trusted entities.

Dapr Integration with Azure Container Apps

Overview

- **What is Dapr?**:
  - The Distributed Application Runtime (Dapr) is an open-source project under the CNCF, aimed at simplifying the development of distributed, microservice-based applications.
  - **Key Features**: It offers a set of APIs for handling various distributed system concerns like service-to-service communication, state management, and more.

Benefits of Managed Dapr in Azure Container Apps

- **Managed Service**: Azure Container Apps provides a managed version of Dapr:
  - **Version Management**: Automatic handling of Dapr version upgrades.
  - **Simplified Interaction**: Offers a streamlined way for developers to interact with Dapr capabilities.

Dapr APIs in Azure Container Apps

Dapr API	Description
Service-to-service invocation	Enables discovery and secure communication between services with features like automatic mTLS.
State management	Manages application state with transaction support, CRUD operations on data stores.
Pub/Sub	Facilitates event-driven communication between services via message brokers like Kafka or RabbitMQ.
Bindings	Allows applications to react to external events or resources (e.g., Azure Service Bus, HTTP).
Actors	Implements the actor model for scalable, single-threaded units of work, ideal for bursty workloads.
Observability	Integrates with Azure Application Insights for tracing and monitoring distributed systems.
Secrets	Provides secure access to secrets within your application or via Dapr components.
Configuration	Manages app configuration, allowing for dynamic updates without redeployment.

Note: The above APIs are stable. For experimental or alpha features, refer to Dapr's documentation on limitations.

Using Dapr with Azure Container Apps

- **Developer Productivity**: By integrating Dapr, developers can focus on business logic rather than the complexities of service orchestration.
- **Seamless Integration**: No need for developers to manage the Dapr runtime themselves; Azure Container Apps takes care of this, making it easier to adopt microservice patterns.

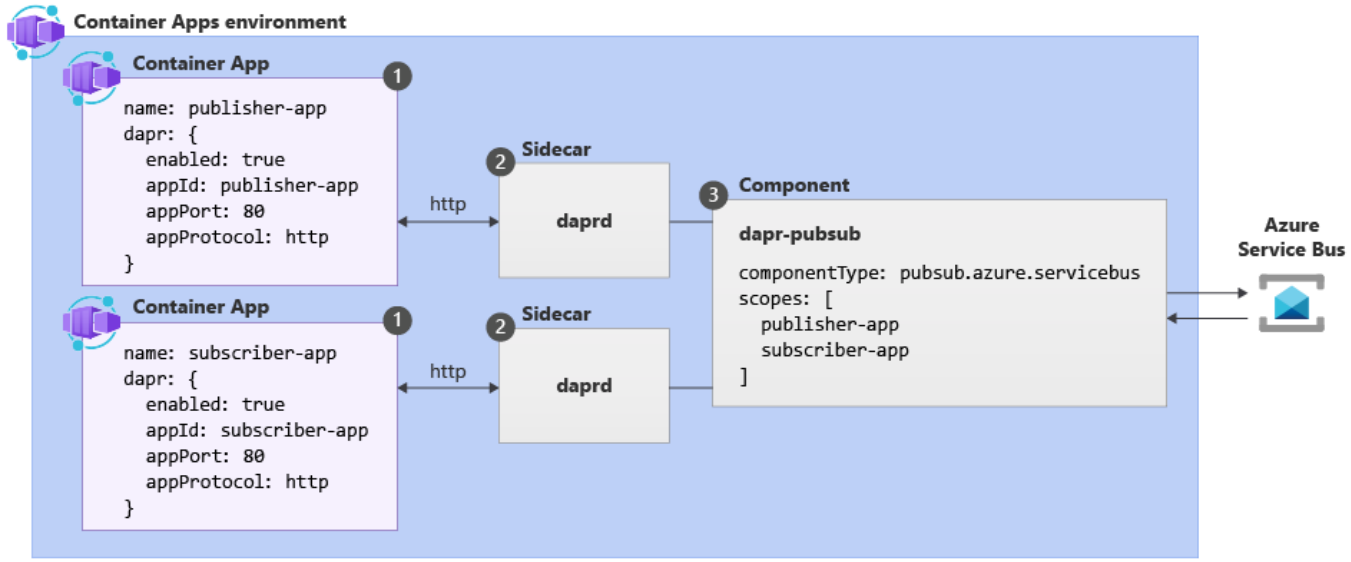
This integration of Dapr into Azure Container Apps provides developers with a powerful toolkit for building scalable, resilient, and observable distributed applications with minimal overhead.

Dapr Core Concepts in Azure Container Apps

Overview

- **Pub/Sub Example**: Used here to explain Dapr integration within the context of Azure Container Apps.

Diagram: (Visualize a diagram here illustrating the flow between Dapr-enabled container apps, the Dapr sidecar, and external services.)



Core Concepts

1. **Container Apps with Dapr Enabled**:
   - **Configuration**: Dapr is enabled by setting Dapr arguments at the container app level. These settings apply to all revisions of the app.
   - **Example Configuration**:

```
az containerapp update --name myapp --resource-group myResourceGroup --
set "template.properties.dapr.enabled=true" --set
"template.properties.dapr.appId=mydaprapp" --set
"template.properties.dapr.appPort=3000"
```

2. **Dapr Sidecar**:
   - **Functionality**: Provides access to Dapr APIs via HTTP or gRPC:
     - **HTTP Port**: 3500
     - **gRPC Port**: 50001

3. **Dapr Component Configuration**:
   - **Components**: Dapr uses a modular approach where each component adds specific functionality.
   - **Scopes**: Dapr components can be shared or specific to certain apps through scopes in the component configuration.

Dapr Enablement in Azure Container Apps

- **Methods to Configure Dapr**:
  - **CLI**: Using Azure CLI commands.

    - **IaC Templates**: Through Bicep or ARM templates.
    - **Azure Portal**: Via the management interface.

Dapr Components and Scopes

- **Components**:
    - **Purpose**: Offer abstractions for connecting to and managing external services.
    - **Sharing**: Components can be shared across multiple apps or scoped to specific ones.
    - **Security**: Use Dapr secrets for configuration metadata.
- **Scopes**:
    - **Default Behavior**: All Dapr-enabled apps load all deployed components unless scoped.
    - **Example Component Configuration with Scope**:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: pubsub
spec:
  type: pubsub.azure.servicebus
  version: v1
  metadata:
  - name: connectionString
    secretKeyRef:
      name: sb-conn-string
      key: value
  scopes:
    - myPublisherApp
    - mySubscriberApp
```

Here, the `pubsub` component is only loaded by `myPublisherApp` and `mySubscriberApp`.

Note: Configuring Dapr in Azure Container Apps simplifies the management of distributed services by abstracting away the complexities of service-to-service communication, state management, and more, allowing developers to focus on application logic while leveraging Dapr's robust feature set.

AZ-204: Implement User Authentication and Authorization

Course Duration: 1 hour 25 minutes

Learning Path: Intermediate Developer

Completion Status: 0 of 4 modules completed

Overview

- **Objective**: Learn to implement authentication and authorization for resources using:
  - Microsoft identity platform
  - Microsoft Authentication Library (MSAL)
  - Shared Access Signatures (SAS)
  - Microsoft Graph

Key Topics

- **Microsoft Identity Platform**: Integration for authentication and authorization services.
- **Microsoft Authentication Library**: SDK for handling authentication scenarios.
- **Shared Access Signatures**: For secure delegation of access to Azure Storage resources.
- **Microsoft Graph**: For accessing data in Microsoft 365, Windows 10, and Enterprise Mobility + Security.

Prerequisites

- **Experience**:
  - At least one year of experience in developing scalable solutions across all software development phases.
- **Knowledge**:
  - Basic understanding of Azure, its services, and the Azure portal.
- **Recommended**:
  - Complete AZ-900: Azure Fundamentals if you're new to Azure or cloud computing.

Additional Notes

- This module assumes some familiarity with development practices and Azure basics. If you're rusty on Azure fundamentals, consider brushing up on those before diving in deep here.

This format provides a quick reference for what you'll learn in the course, the prerequisites, and how to prepare if you're not quite up to speed with Azure concepts. Remember, if you're feeling lost in the Azure wilderness, the AZ-900 course might just be the map you need!

Introduction to Microsoft Identity Platform

Completion: Completed

Experience Points: 100 XP

Duration: 3 minutes

Overview

The Microsoft identity platform offers developers a suite of tools:

- **Authentication Service**: Securely authenticates users.
- **Open-Source Libraries**: Facilitates integration into applications.
- **Application Management Tools**: Helps in managing applications' access and permissions.

Learning Outcomes

Upon completion of this module, you will:

- **Identify Components**:
    - Understand what makes up the Microsoft identity platform.
- **Service Principals**:
    - Recognize the three types of service principals:
        1. Application Objects: Represent applications in Azure AD.
        2. Service Principals: Instances of applications in a tenant.
        3. Managed Identities: Automatically managed identities for Azure resources.
- **Permissions and Consent**:
    - Learn how permissions in Azure AD work.
    - Understand the concept of user consent for granting permissions.
    - Grasp how conditional access policies can affect application access.

Key Concepts

- **Service Principals**: These are like user accounts for your apps, allowing them to authenticate and access resources securely.
- **User Consent**: Users grant apps permission to access their data at runtime, based on the permissions your app requests.
- **Conditional Access**: This adds an extra layer of security, where access can be granted or denied based on various conditions like location, device health, or user identity.

Remember, understanding these foundational elements is like learning the secret handshake to enter the Azure club of secure application development!

Explore the Microsoft Identity Platform

Introduction

- The Microsoft identity platform allows developers to build applications where users can sign in using Microsoft or social accounts and access various APIs.

Key Components

- **Authentication Service**:
    - Compliant with OAuth 2.0 and OpenID Connect standards.
    - Supports authentication for:
        - **Work or School Accounts** through Microsoft Entra ID.
        - **Personal Microsoft Accounts** (Skype, Xbox, Outlook.com).
        - **Social or Local Accounts** with Azure AD B2C.
        - **Social or Local Customer Accounts** via Microsoft Entra External ID.
- **Open-Source Libraries**:

    - **Microsoft Authentication Libraries (MSAL)** for seamless authentication integration.
    - Support for other standards-compliant libraries.

- **Microsoft Identity Platform Endpoint**:

    - Compatible with MSAL or other standards-compliant libraries.
    - Implements user-friendly scopes.

- **Application Management Portal**:

    - Provides an interface in the Azure Portal for app registration and configuration.

- **Application Configuration**:

    - **API and PowerShell**: Allows for programmatic configuration via Microsoft Graph API and PowerShell, enhancing DevOps capabilities.

Modern Security Features

- The platform integrates:
    - **Passwordless Authentication**: Eliminates the need for traditional password inputs.
    - **Step-Up Authentication**: Adds layers of authentication when needed for increased security.
    - **Conditional Access**: Applies policies based on various conditions like device, location, or user risk.

Developer Benefits

- Developers can focus on building applications rather than implementing complex security features, as these are natively supported by the Microsoft identity platform.

Remember, with Microsoft's identity platform, you're not just adding security; you're adopting a whole ecosystem of identity and access management, making your app not just secure, but also user-friendly and future-proof!

Explore Service Principals

Registration with Microsoft Entra ID

- **Purpose**: To delegate identity and access management to Microsoft Entra ID, applications must be registered.
- **Registration Outcomes**:
    - An **application identity** is created for your app.
    - **Single Tenant** vs. **Multi-Tenant**:
        - **Single Tenant**: App is only accessible within your tenant.
        - **Multi-Tenant**: App can be accessed across different tenants.

Application Object & Service Principals

- **Creation**:

    - When registering an app, both an application object and a service principal object are created in the home tenant.
    - **App or Client ID**: A globally unique identifier for your app.

- **Customization**:

    - Add secrets, certificates, scopes in the Azure portal.
    - Customize app branding in the sign-in dialog.

- **Creation Methods**:

    - Besides Azure portal, use Azure PowerShell, Azure CLI, Microsoft Graph, etc.

Application Object

- **Scope**: Unique to each application.
- **Location**: Resides in the home tenant where registered.
- **Function**:

    - Serves as a template for creating service principal objects.
    - Defines:
        - Token issuance policies.
        - Resources the app might need.
        - Actions the app is permitted to perform.

- **Schema**: Defined by the Microsoft Graph Application entity.

Key Takeaways

- **Application Object**: Think of it as the blueprint or class in OOP.
- **Service Principal**: Each tenant where the app is used gets its own service principal, akin to an instance of the application.

Remember, in the world of Azure, registering an application is like giving your app its own identity card, while service principals are like the app's avatars roaming in each tenant's territory.

Service Principal Object

Security Principal Concept:

- When accessing resources secured by Microsoft Entra, entities must have a security principal:
    - **User Principal**: For users.
    - **Service Principal**: For applications.

Role of Service Principals:

- Defines access policies and permissions within a tenant.
- Enables authentication and authorization.

Types of Service Principals

1. **Application Service Principal**

- **Purpose**: Local representation of a global application object.
  - Created in each tenant where the app is used.
  - Specifies what the app can do, who can access it, and what resources it can use.

2. **Managed Identity Service Principal**

- **Use**: Represents a managed identity for applications to connect securely to resources.
  - Automatically created when managed identity is enabled.
  - Permissions are granted but cannot be directly modified.

3. **Legacy Service Principal**

- **Characteristics**:
  - Represents legacy applications or those created before modern app registrations.
  - Can have credentials, service principal names, reply URLs, etc.
  - Properties can be edited by authorized users but has no associated app registration.

Application Objects vs. Service Principals

- **Application Object**:
  - Global representation of the application.
  - One-to-one relationship with the software application.
  - One-to-many relationship with service principals.
- **Service Principal**:
  - Local representation in a specific tenant.
  - Must be created in each tenant where the app is used.
  - **Single-Tenant**: One service principal in the home tenant.
  - **Multi-Tenant**: Service principals created in each tenant where consent is given.

Key Dynamics:

- The application object acts as a template for service principals, defining common and default properties.
- Service principals provide the actual identity for sign-in and resource access within each tenant.

Remember, in the grand theater of Azure, service principals are like the actors, taking on roles (permissions) in each tenant's production, while the application object is the script, outlining what those roles should be across all performances.

Discover Permissions and Consent

Authorization Model

- **Microsoft Identity Platform**: Uses OAuth 2.0 for third-party apps to access resources on behalf of users.
- **Resource Identifiers**: Each resource has an application ID URI, e.g.,
  - Microsoft Graph: `https://graph.microsoft.com`
  - Microsoft 365 Mail API: `https://outlook.office.com`
  - Azure Key Vault: `https://vault.azure.net`

Scopes (Permissions)

- **Definition**: Functional chunks of a resource that apps can request.
  - Example: `https://graph.microsoft.com/Calendars.Read`
- **Requesting Permissions**: Apps specify scopes in requests to the authorize endpoint.
  - **Admin Consent**: Required for high-privilege permissions.

Note: If the resource identifier is omitted in the scope parameter, it defaults to Microsoft Graph.

Permission Types

- **Delegated Access**:
  - Apps act on behalf of a signed-in user.
  - Consent can be given by user or administrator.
- **App-Only Access**:
  - For apps running without user interaction (e.g., background services).
  - Only administrators can consent to these permissions.

Consent Types

- **Static User Consent**:
  - All permissions are predefined in the app's Azure portal configuration.
  - User or admin consent is prompted if not already granted.
  - Allows admin consent for all users in the organization.
  - **Issues**:
    - Requires all permissions at first sign-in, potentially overwhelming users.
    - Difficult to adapt to dynamically changing resource access needs.
- **Incremental and Dynamic User Consent**:
  - Allows apps to request permissions incrementally or dynamically as needed.
- **Admin Consent**:
  - Administrators can consent to permissions for all users.

Key Points

- **User Control**: Users and admins have control over data access.
- **Scope Definition**: Helps in fine-tuning app permissions to exactly what's needed.

- **Consent Strategy**: Important for maintaining user trust while ensuring app functionality.

Remember, in the realm of permissions and consents, it's like being at a dinner party where you're both the host and guest. You get to decide what dishes (permissions) are on the menu, but you need your guests' (users) consent to serve them.

Incremental and Dynamic User Consent & Admin Consent

Incremental Consent

- **Advantage**: Request permissions as needed rather than all at once.
  - **Implementation**: Include new scopes in the `scope` parameter when requesting an access token.
- **Limitation**: Only applies to delegated permissions, not app-only access.

Important:

- Dynamic consent requires all permissions to be registered in Azure if admin consent is needed for any of them.

Admin Consent

- **Requirement**: For high-privilege permissions.
- **Process**: Admins can consent for the entire organization.
  - **Static Permissions**: Must be set in the app registration portal for admin consent.

Requesting Individual User Consent

- **Method**:
  - Via the `scope` parameter in the OpenID Connect or OAuth 2.0 authorization request.

Example HTTP Request:

```
GET https://login.microsoftonline.com/common/oauth2/v2.0/authorize?
client_id=00001111-aaaa-2222-bbbb-3333cccc4444
&response_type=code
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&response_mode=query
&scope=
https%3A%2F%2Fgraph.microsoft.com%2Fcalendars.read%20
https%3A%2F%2Fgraph.microsoft.com%2Fmail.send
&state=12345
```

- **Scope Parameter**: Space-separated list of permissions.
  - Example: `https://graph.microsoft.com/calendars.read` and `https://graph.microsoft.com/mail.send`
- **User Interaction**:
  - If permissions are not previously consented, the user is prompted to grant them.

Key Points

- **Incremental Permissions**: Enhances user experience by not overwhelming with permissions requests initially.
- **Admin Consent**: Critical for managing high-risk permissions, ensuring organizational security.
- **Dynamic vs. Static**: Dynamic consent offers flexibility but needs careful management for admin scenarios.

Remember, with consent in Microsoft's identity platform, it's like asking for permission to borrow a cup of sugar. You might start with just a teaspoon, but if your recipe calls for more, you'll ask incrementally. And for the whole flour sack? That's when you might need to talk to the head chef (admin consent).

Discover Conditional Access

Introduction to Conditional Access

- **Function**: Provides multiple ways to secure apps and services within Microsoft Entra ID.
- **Examples**:
  - Enforcing multifactor authentication (MFA)
  - Allowing only Intune-enrolled devices
  - Restricting access based on user location or IP range

Impact on Apps

- **General**: Usually doesn't require changes in app behavior or code.
- **Exceptions**: Apps might need modifications if they:
  - Use the on-behalf-of flow
  - Access multiple services/resources
  - Are single-page apps using MSAL.js
  - Are web apps calling a resource

Code Changes:

- Necessary for handling Conditional Access challenges, often involving an interactive sign-in request.

Handling Conditional Access Challenges

- **Application**: Conditional Access policies can be set for:
  - The app itself
  - Web APIs the app calls

Conditional Access Scenarios

- **Single-Tenant iOS App Example**:
  - **Scenario**: App signs in user without API access request.
  - **Outcome**: Conditional Access policy automatically enforced upon sign-in, requiring MFA.
- **Multi-Tier Service Example**:

    - **Scenario**: App uses a middle tier to access API. Policy applies to the downstream API.
    - **Outcome**:
        - App requests token for middle tier, which then uses on-behalf-of flow for API access.
        - Middle tier receives a claims challenge from Conditional Access.
        - Middle tier must handle and forward this challenge back to the app for compliance.

Key Points

- **Flexibility**: Enterprise customers can apply or modify policies at any time.
- **Challenge Handling**: Apps need to be prepared to manage Conditional Access challenges to remain functional under new or changed policies.

Remember, Conditional Access is like the bouncer at the club of your app's data. It decides who gets in, when, and how, ensuring only the right users with the right credentials can access your app's VIP areas.

Implement Authentication with MSAL

Time Remaining: 25 minutes

Module Progress: 0 of 6 units completed

Overview

- **Objective**: To learn how to integrate authentication into applications using the Microsoft Authentication Library (MSAL).

Learning Goals

- Understand the basics of MSAL.
- Implement user authentication in applications.
- Handle authentication flows, token acquisition, and management.

Key Topics to Cover

1. **MSAL Basics**:
    - Overview of MSAL and its components.
2. **Setting Up Authentication**:
    - Configuring MSAL in your application environment.
    - Registering an application in Azure AD to use with MSAL.
3. **Authentication Flows**:
    - Interactive authentication (login prompts).
    - Silent token acquisition for improved user experience.
    - On-behalf-of flow for service-to-service communication.
4. **Token Management**:

    - Handling access tokens and refresh tokens.
    - Token caching strategies.

5. **Error Handling and Security Best Practices**:

    - Dealing with authentication errors.
    - Implementing security measures like token validation.

6. **Advanced Topics**:

    - Conditional Access handling.
    - Multi-tenant applications.

Next Steps

- Start with the first unit to get hands-on experience with MSAL setup.
- Progress through each unit, focusing on practical implementation of authentication concepts.

Remember, implementing authentication with MSAL is like giving your app its own set of keys to the kingdom of secure user identity management. Let's get those keys turned and doors opened securely!

Introduction to Microsoft Authentication Library (MSAL)

Overview

- **Purpose**: MSAL allows developers to obtain tokens from Microsoft identity platform for user authentication and access to secure APIs.

Learning Outcomes

Upon completion, you will:

- **Understand MSAL Benefits**:

    - Know how MSAL simplifies authentication and supports various application types and scenarios.

- **Client Application Instantiation**:

    - Learn to create both public client applications (like desktop or mobile apps) and confidential client applications (like web apps or APIs) programmatically.

- **App Registration**:

    - Be able to register an application with the Microsoft identity platform, setting up necessary configurations.

- **Token Retrieval**:

    - Implement token acquisition in an application using MSAL.NET, enabling secure API access.

Key Concepts

- **Token Acquisition**: MSAL handles the complexities of token management, making your application more secure and easier to maintain.
- **Application Types**:
  - **Public Clients**: Apps where the client secret cannot be trusted (e.g., mobile or desktop apps).
  - **Confidential Clients**: Apps where the client secret can be secured (e.g., web apps, web APIs).

Remember, with MSAL, you're not just adding security; you're getting a Swiss Army knife of authentication tools, perfect for any developer camping trip through the wilds of identity management.

Explore the Microsoft Authentication Library (MSAL)

Overview of MSAL

- **Purpose**: Enables token acquisition for user authentication and API access.
- **Platforms**: Supports .NET, JavaScript, Java, Python, Android, iOS, among others.
- **Benefits**:
  - Simplifies OAuth protocol handling.
  - Manages tokens for users or on behalf of applications.
  - Handles token caching and refreshing.
  - Facilitates audience specification for sign-ins.
  - Configures applications via configuration files.
  - Provides troubleshooting tools like exceptions, logging, and telemetry.

Application Types and Supported Platforms

- **Application Types**:
  - Web applications
  - Web APIs
  - Single-page applications (SPA)
  - Mobile and native applications
  - Daemons and server-side applications

Supported Platforms by MSAL Library:

Library	Supported Platforms and Frameworks
MSAL for Android	Android
MSAL Angular	Single-page apps with Angular and Angular.js
MSAL for iOS and macOS	iOS and macOS
MSAL Go (Preview)	Windows, macOS, Linux
MSAL Java	Windows, macOS, Linux
MSAL.js	JavaScript/TypeScript frameworks like Vue.js, Ember.js

Library	Supported Platforms and Frameworks
MSAL.NET	.NET Framework, .NET, .NET MAUI, WINUI, Xamarin
MSAL Node	Web apps (Express), desktop apps (Electron), console apps
MSAL Python	Windows, macOS, Linux
MSAL React	Single-page apps with React and similar libraries

Key Advantages of MSAL

- **Uniform API**: Consistent token acquisition methods across platforms.
- **Security**: Reduces direct protocol handling, enhancing security.
- **Developer Experience**: Simplifies development by managing token lifecycle and providing error insights.

Remember, MSAL is like your personal authentication butler; it handles all the intricate details of security tokens so you can focus on making your app do amazing things without worrying about who's knocking at your door.

Authentication Flows in MSAL

Types of Authentication Flows

Authentication Flow	Purpose	Supported Application Types
Authorization Code	User sign-in and API access on behalf of the user. Uses PKCE for SPAs.	Desktop, Mobile, SPA, Web
Client Credentials	API access using the application's identity. Ideal for server-to-server or automated processes.	Daemon
Device Code	User sign-in on devices with limited input capabilities or for CLI.	Desktop, Mobile
Implicit Grant	User sign-in and API access. **Note**: No longer recommended; use Authorization Code with PKCE.	SPA, Web (deprecated)
On-behalf-of (OBO)	Web API to web API access, passing user identity.	Web API
Username/Password (ROPC)	Direct user sign-in with password. **Note**: Not recommended due to security issues.	Desktop, Mobile
Integrated Windows Authentication (IWA)	Silent token acquisition on domain or Microsoft Entra joined devices.	Desktop, Mobile

Public vs. Confidential Clients

- **Public Client Applications**:

    - **Characteristics**:
        - Run on user-end devices (desktop, mobile, browser apps).
        - Cannot safely keep secrets due to potential exposure.
        - Use public client flows like Authorization Code with PKCE.
    - **Examples**: Mobile apps, Desktop apps, Browser-based SPAs.

- **Confidential Client Applications**:

    - **Characteristics**:
        - Operate on servers (web apps, web APIs, services).
        - Can securely hold secrets, proving identity to the authorization server.
        - Secrets are passed securely in the back channel.
    - **Examples**: Web applications, Daemon services, Web APIs.

Key Concepts

- **Client ID**: Exposed in all client types through the browser or app.
- **Client Secret**: Only confidential clients can safely use these for authentication, passed securely.

Remember, choosing the right authentication flow with MSAL is like picking the right key for the right door. Public clients are like house keys that anyone might find, while confidential clients are like those to your secret underground vault, trusted to keep the valuables safe.

Initialize Client Applications with MSAL.NET

Prerequisites

- **App Registration**: Before initialization, your application must be registered with Microsoft identity platform.
    - **Registration Details**:
        - **Application (client) ID**: A GUID string.
        - **Directory (tenant) ID**: For organizational IAM, identifies your tenant.
        - **Authority**: Comprises the identity provider's URL and sign-in audience.
        - **Client Credentials**:
            - For confidential clients: Client secret or certificate (X509Certificate2).
        - **Redirect URI**: Required for web apps and some public client apps.

Initialization Methods

- **Using Application Builders**:
    - **PublicClientApplicationBuilder**
    - **ConfidentialClientApplicationBuilder**
    - Allows configuration from code or configuration files or both.

Code Examples

Public Client Application

```
IPublicClientApplication app =
PublicClientApplicationBuilder.Create(clientId).Build();
```

- **Purpose**: To create an application that signs in users in the Microsoft Azure public cloud with work and school or personal Microsoft accounts.

Confidential Client Application

```
string redirectUri = "https://myapp.azurewebsites.net";
IConfidentialClientApplication app =
ConfidentialClientApplicationBuilder.Create(clientId)
    .WithClientSecret(clientSecret)
    .WithRedirectUri(redirectUri)
    .Build();
```

- **Purpose**: For a web app at `myapp.azurewebsites.net`, handling tokens for users in the Microsoft Azure public cloud with work, school, or personal Microsoft accounts.
- **Note**: Uses a client secret for identification with the identity provider.

Key Points

- **Flexibility**: MSAL.NET 3.x allows for flexible configuration of applications.
- **Security**: Confidential clients can securely use secrets for authentication.
- **Ease of Use**: Builders simplify the instantiation process, making it straightforward for developers to set up authentication.

Remember, initializing with MSAL is like setting up a profile in a social network but with much higher stakes - you're not just making friends, you're securing identities!

Builder Modifiers in MSAL.NET

Common Modifiers for Public and Confidential Clients

Modifier	Description
`.WithAuthority()`	Sets the default authority. Allows selection of Azure Cloud, audience, and tenant.
`.WithTenantId(string tenantId)`	Overrides the tenant ID, useful for multi-tenant applications or specifying a single tenant.
`.WithClientId(string)`	Overrides the client ID when needed.
`.WithRedirectUri(string redirectUri)`	Overrides the default redirect URI, especially useful for broker scenarios or testing.
`.WithComponent(string)`	Specifies the library name for telemetry purposes.

Modifier	Description
`.WithDebugLoggingCallback()`	Enables logging to `Debug.Write` for debugging purposes.
`.WithLogging()`	Sets up a callback for logging debug traces.
`.WithTelemetry(TelemetryCallback telemetryCallback)`	Configures telemetry sending via a delegate.

Examples

Setting Authority

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(clientId)
    .WithAuthority(AzureCloudInstance.AzurePublic, tenantId)
    .Build();
```

Setting Redirect URI

```
IPublicClientApplication app;
app = PublicClientApplicationBuilder.Create(client_id)
    .WithAuthority(AzureCloudInstance.AzurePublic, tenant_id)
    .WithRedirectUri("http://localhost")
    .Build();
```

Modifiers Specific to Confidential Clients

- Found in `ConfidentialClientApplicationBuilder`:
  - `.WithCertificate(X509Certificate2 certificate)`: Uses a certificate for authentication.
  - `.WithClientSecret(string clientSecret)`: Uses a client secret for authentication.

Important: These methods (`WithCertificate` and `WithClientSecret`) are mutually exclusive; using both results in an exception.

Key Points

- **Authority Customization**: `.WithAuthority()` allows for detailed control over where authentication occurs.
- **Security**: For confidential clients, choosing between certificate and secret provides flexibility in security strategy.
- **Logging & Telemetry**: These features help in debugging and monitoring application behavior.

Remember, using these modifiers in MSAL is like customizing your car's interior - you can adjust the seat (authority), steering wheel (client ID), and even the dashboard's display (telemetry) to fit your journey's needs. Just make sure you don't try to start the car with both the key and a push button at the same time!

Implement Shared Access Signatures (SAS)

Time Remaining: 18 minutes

Module Progress: 0 of 6 units completed

Overview

- **Objective**: To understand and implement shared access signatures (SAS) for secure, controlled access to Azure Storage resources.

Learning Goals

- Understand the concept of Shared Access Signatures.
- Learn how to generate SAS tokens for different scopes.
- Implement SAS in various Azure Storage scenarios.

Key Topics to Cover

1. **Introduction to SAS**:
    - What SAS is and why it's used for securing access.
2. **Types of SAS**:
    - **Service SAS**: Access to specific resources in a single storage service.
    - **Account SAS**: Access to multiple services within an Azure Storage account.
    - **User Delegation SAS**: Access granted via Azure AD credentials, more secure for blob storage.
3. **Generating SAS Tokens**:
    - **Service-level SAS**:

```
var sasToken = account.Sas.GetAccountSasToken(new AccountSasBuilder
{
    Services = AccountSasBuilder.Services.Blobs,
    ResourceTypes = AccountSasBuilder.ResourceTypes.Object |
AccountSasBuilder.ResourceTypes.Service,
    Protocol = SasProtocol.Https,
    StartsOn = DateTimeOffset.UtcNow,
    ExpiresOn = DateTimeOffset.UtcNow.AddHours(1),
    SignatureExpiry = DateTimeOffset.UtcNow.AddHours(1)
});
```

    - **Blob-specific SAS example**:

```
var blobClient = new BlobClient(new
Uri("https://myaccount.blob.core.windows.net/mycontainer/myblob"));
var sasBuilder = new BlobSasBuilder
```

```
{
    BlobContainerName = "mycontainer",
    BlobName = "myblob",
    Resource = "b",
    StartsOn = DateTimeOffset.UtcNow,
    ExpiresOn = DateTimeOffset.UtcNow.AddHours(1)
};
sasBuilder.SetPermissions(BlobSasPermissions.Read);
var sasToken = blobClient.GenerateSasUri(sasBuilder).Query;
```

4. **Security Considerations**:
    - Limiting permissions and time windows.
    - Using SAS in conjunction with other security measures like Azure AD.
5. **Revoking Access**:
    - Rotating keys, and understanding the implications of SAS token revocation.
6. **Practical Implementation**:
    - How to apply SAS in real-world scenarios, like allowing temporary access to a blob for downloading or uploading.

Next Steps

- Begin with the first unit to explore the basics of SAS.
- Work through each unit to gain practical experience in generating and using SAS tokens securely.

Remember, SAS tokens are like giving someone a key to a specific door in your storage house - they can only enter when and where you've allowed, and you can change the locks anytime!

Introduction to Shared Access Signatures (SAS)

Overview

- **What is SAS?**: A URI that provides restricted access to Azure Storage resources, allowing delegation of access without sharing account keys.

Learning Outcomes

Upon completion, you will:

1. **Recognize SAS Types**:
    - **Service SAS**: Grants access to a resource in a single storage service (e.g., Blob, File, Queue, Table).
    - **Account SAS**: Provides access to multiple storage services within the same account.
    - **User Delegation SAS**: More secure form of SAS that leverages Azure AD credentials for blob access.

2. **Understand Use Cases**:
    - When to use SAS for granting temporary or limited access to resources without exposing account keys.
    - Scenarios where direct access control through Azure AD might not be feasible or desired.
3. **Create a Stored Access Policy**:
    - Learn to set up a policy that can manage permissions centrally, making SAS revocation and modification easier.

Key Concepts

- **Security**: SAS allows for granular control over what actions can be performed and for how long, enhancing security by limiting exposure.
- **Flexibility**: Useful for scenarios where you need to grant specific rights for a limited time, like allowing a third-party to upload logs or download files.

Remember, SAS is like giving someone a temporary pass to your storage locker. You control what they can do inside, how long they can stay, and you can always take back the pass if needed.

Discover Shared Access Signatures (SAS)

Overview

- **Definition**: SAS is a signed URI providing access to Azure Storage resources with specific permissions for a defined time period.

How SAS Works

- **Components**:
    - **URI**: Points to the storage resource.
    - **Token**: Includes query parameters like the signature, which is generated from SAS parameters and signed with the account key or Azure AD credentials.
- **Security**: The signature is verified by Azure Storage to authorize the access.

Types of Shared Access Signatures

1. **User Delegation SAS**:
    - **Security**: Uses Microsoft Entra (Azure AD) credentials.
    - **Scope**: Limited to Blob storage.
    - **Recommendation**: Preferred for its security over key-based access.
2. **Service SAS**:
    - **Security**: Uses the storage account key.

    - **Scope**: Can access resources in any of the Azure Storage services (Blob, Queue, Table, Azure Files).

3. **Account SAS**:

    - **Security**: Also uses the storage account key.
    - **Scope**: Can delegate access across multiple storage services within the account.

Security Note:

- Microsoft advises using Microsoft Entra credentials for creating SAS when possible due to better security practices, especially for Blob storage where user delegation SAS is applicable.

Remember, choosing the right type of SAS is like deciding what kind of key you're giving to your guests:

- **User Delegation SAS**: Like a guest pass with a photo ID.
- **Service SAS**: Like a room key that only works for specific areas.
- **Account SAS**: Like a master key to the entire storage facility, but with restrictions on what can be done.

How Shared Access Signatures (SAS) Work

Components of a SAS URI

- **URI**: Specifies the resource location, e.g., `https://medicalrecords.blob.core.windows.net/patient-images/patient-116139-nq8z7f.jpg?`
- **SAS Token**: Contains the permissions and time constraints, e.g., `sp=r&st=2020-01-20T11:42:32Z&se=2020-01-20T19:42:32Z&spr=https&sv=2019-02-02&sr=b&sig=SrW1HZ5Nb6MbRzTbXCaPm%2BJiSEn15tC91Y4umMPwVZs%3D`

SAS Token Components

Component	Description
sp=r	Access rights: Here, 'r' for read. Possible values include 'a' (add), 'c' (create), 'd' (delete), 'l' (list), 'w' (write).
st=...	Start time for access.
se=...	End time for access. Example grants 8 hours of access.
sv=...	Version of the storage API used.
sr=b	Specifies the type of resource ('b' for blob).
sig=...	The cryptographic signature ensuring security of the SAS token.

Best Practices for SAS

- **Use HTTPS**: Always distribute SAS tokens over HTTPS to prevent man-in-the-middle attacks.
- **Prefer User Delegation SAS**:

    - Provides superior security by not requiring the storage account key, using Azure AD credentials instead.

- **Limit SAS Lifetime**:
    - Set expiration times as short as practical to minimize exposure if a token is compromised.
- **Principle of Least Privilege**:
    - Grant only the necessary permissions. For instance, use `sp=r` for read-only access when that's all that's needed.
- **Consider Middle-Tier Services**:
    - When SAS might pose too much risk, implement a middle-tier service to act as a gatekeeper for storage access.

Remember, using SAS is like giving someone a time-limited, purpose-specific key to your storage. You control what they can do, when they can do it, and for how long. Just make sure you don't leave the key under the doormat for too long!

Choosing When to Use Shared Access Signatures

When to Use SAS

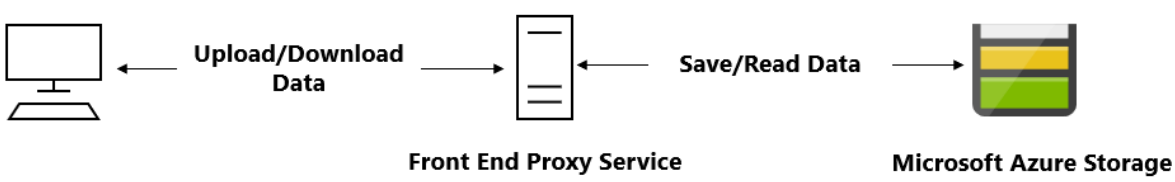
- **Use Case**: To grant secure access to storage account resources for clients without direct permissions.

Common Scenarios

1. User Data Access via a Service

- **Pattern 1: Front-end Proxy Service**
    - Clients interact through a service that handles authentication and business logic.
    - **Pros**: Control and validation.
    - **Cons**: Scalability issues with high data volume or transactions.

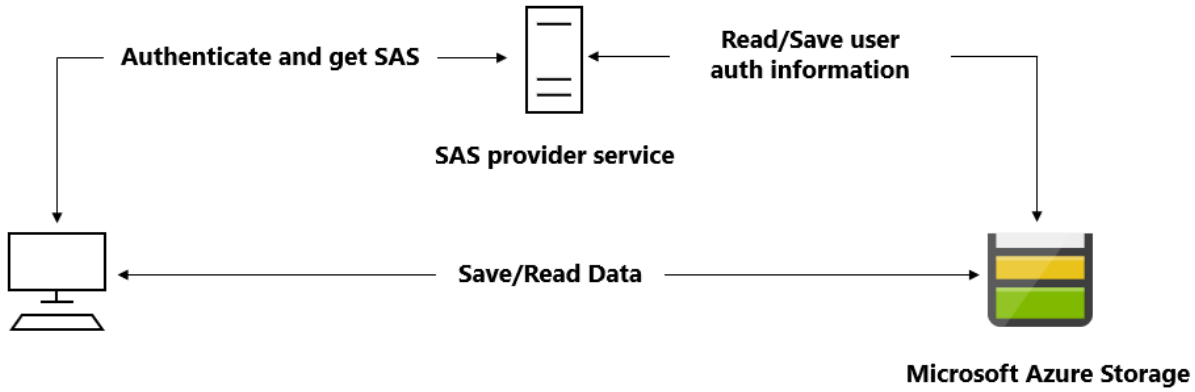
```
Scenario diagram:
[Client] -> [Front-end Proxy Service] -> [Azure Storage]
```



- **Pattern 2: SAS Provider Service**

- A lightweight service generates SAS for direct client access.
- **Pros**: Reduces load on the front-end, clients can directly access storage.
- **Cons**: Less control over data operations.

```
Scenario diagram:
[Client] -> [SAS Provider Service] -> [Client (with SAS)] -> [Azure Storage]
```



Hybrid Approach:

- Some data might be managed via the proxy, others directly with SAS for efficiency.

2. Copy Operations Requiring SAS

- **Cross-Account Blob Copy**:
  - **Source Blob**: SAS must be used for authorization.
  - **Destination Blob**: Optionally use SAS.
- **Cross-Account File Copy**:
  - **Source File**: SAS required for access.
  - **Destination File**: Optional SAS use.
- **Blob to File or File to Blob Copy**:
  - **Source Object**: Must use SAS, even within the same account.

Key Points

- **Flexibility**: SAS allows for direct client interaction with storage, reducing service load.
- **Security**: Ensures that access is time-bound and permission-limited.
- **Efficiency**: Ideal for scenarios where direct access is beneficial without compromising security.

Remember, using SAS is like giving a guest a temporary key card to your hotel room. They can enter and use what's needed but only for a limited time, and you can revoke or change the locks anytime.

Explore Stored Access Policies

Completion: Completed

Experience Points: 100 XP

Duration: 3 minutes

Overview of Stored Access Policies

- **Function**: Provides server-side control over service-level Shared Access Signatures (SAS).
- **Benefits**:
  - Groups SAS for easier management.
  - Allows modifications to SAS constraints (start/end times, permissions).
  - Enables revocation of SAS after issuance.

Supported Storage Resources

- **Blob Containers**
- **File Shares**
- **Queues**
- **Tables**

How Stored Access Policies Work

- **Policy Creation**: You define a policy for a storage resource which includes:
  - **Permissions**: What can be done with the SAS.
  - **Start Time & Expiry Time**: When the SAS is active.
- **SAS Association**: SAS tokens can be linked to this policy, inheriting its constraints.
- **Policy Management**:
  - **Modify**: Change times or permissions for all associated SAS.
  - **Revoke**: Remove or modify the policy to invalidate all associated SAS.

Remember, a stored access policy is like setting house rules for all the guest keys (SAS). If you decide to change the rules or throw out a guest, you can do it from one place without having to collect all the keys back.

Creating a Stored Access Policy

Policy Components

- **Start Time**, **Expiry Time**, **Permissions**: These can be set on either the SAS URI or the stored policy, but not redundantly in both.

Creating or Modifying a Stored Access Policy

1. **Use the Set ACL Operation**:
    - For different Azure Storage resources: Blob Container, Queue, Table, or File Share.
2. **Policy Parameters**:
    - **Signed Identifier**: A unique string up to 64 characters.
    - **Access Policy**: Includes optional parameters like expiry time, start time, and permissions.

Note:

- It might take up to 30 seconds for a new policy to take effect.
- Table entity range restrictions cannot be part of a stored access policy.

Example with C# .NET

```
BlobSignedIdentifier identifier = new BlobSignedIdentifier
{
    Id = "stored access policy identifier",
    AccessPolicy = new BlobAccessPolicy
    {
        ExpiresOn = DateTimeOffset.UtcNow.AddHours(1),
        Permissions = "rw"
    }
};

blobContainer.SetAccessPolicy(permissions: new BlobSignedIdentifier[] { identifier
});
```

Example with Azure CLI

```
az storage container policy create \
    --name <stored access policy identifier> \
    --container-name <container name> \
    --start <start time UTC datetime> \
    --expiry <expiry time UTC datetime> \
    --permissions <(a)dd, (c)reate, (d)elete, (l)ist, (r)ead, or (w)rite> \
    --account-key <storage account key> \
    --account-name <storage account name> \
```

Modifying or Revoking a Policy

- **Modify**:
    - Replace the existing policy via the Set ACL operation with updated parameters.
- **Revoke**:
    - **Delete**: Remove the policy to invalidate all associated SAS.

    - **Rename Identifier**: Break the link between SAS and the policy by changing the identifier.
    - **Change Expiry Time**: Set it to a past date to make associated SAS expire.

To remove or modify:

- Use the Set ACL operation again, specifying only the identifiers to retain or providing an empty body to clear all policies.

Remember, a stored access policy is like setting the house alarm code. If you need to change it or cancel access, you can do so, and all keys (SAS) linked to that old code will no longer work.

Explore Microsoft Graph

Overview

- **Objective**: To understand how Microsoft Graph enables data access and manipulation, and how to construct queries using RESTful APIs and code.

Learning Goals

- Gain an understanding of Microsoft Graph's role in Microsoft 365.
- Learn how to query Microsoft Graph via REST APIs.
- Implement Microsoft Graph queries in code for various platforms.

Key Topics to Cover

1. **Introduction to Microsoft Graph**:
    - What it is and its significance in Microsoft's ecosystem.
2. **Microsoft Graph Architecture**:
    - The structure of Microsoft Graph, including endpoints, APIs, and SDKs.
3. **Data Access with Microsoft Graph**:
    - How to access different types of data (user profiles, emails, files, etc.).
4. **REST API Fundamentals**:
    - Basics of REST, including HTTP methods, authentication, and JSON responses.
5. **Forming Queries**:
    - Constructing queries to fetch, update, or delete data.
    - Using OData query parameters for filtering, sorting, and pagination.
6. **Code Implementation**:
    - Writing code to interact with Microsoft Graph using various SDKs.
    - Example in *C#*:

```
using Microsoft.Graph;
using System;

var graphClient = new GraphServiceClient(
    new DelegateAuthenticationProvider(async (requestMessage) => {
        // Authenticate request
        var token = await GetAccessTokenAsync();
        requestMessage.Headers.Authorization =
            new AuthenticationHeaderValue("Bearer", token);
    }));

// Example request to get user's profile
var user = await graphClient.Me.Request().GetAsync();
Console.WriteLine(user.DisplayName);
```

7. **Permissions and Security**:
    - Understanding the permission model in Microsoft Graph.
    - Implementing secure access to Graph resources.
8. **Practical Scenarios**:
    - Applying Microsoft Graph in real-world applications like productivity tools, communication apps, etc.

Next Steps

- Begin with the first unit to grasp the basics of Microsoft Graph.
- Progress through each unit to build up your skills in using Microsoft Graph effectively.

Remember, Microsoft Graph is like a universal translator for Microsoft 365 services; it helps your apps speak the language of Microsoft's data services fluently.

Introduction to Microsoft Graph

Overview

- **Purpose**: Microsoft Graph allows developers to build applications that leverage Microsoft 365 data, enhancing user interaction across millions of users.

Learning Outcomes

Upon completion, you will be able to:

1. **Understand Microsoft Graph Benefits**:
    - **Centralized Data Access**: Microsoft Graph provides a single endpoint for accessing data across Microsoft services.
    - **Permission Model**: Granular control over data access with Microsoft Entra ID (Azure AD).

- **Cross-Platform Support**: Apps can be built for various platforms using consistent APIs.

2. **Perform Microsoft Graph Operations**:
   - **Using REST**:
     - Construct HTTP requests to interact with Microsoft Graph resources.
     - Example: Fetching user profile information:

```
GET https://graph.microsoft.com/v1.0/me
```

   - **Using SDKs**:
     - Utilize SDKs for easier integration in your preferred programming language.
     - Example in C#:

```
var graphClient = new GraphServiceClient(
    new DelegateAuthenticationProvider(async (requestMessage) => {
        // Authentication logic
    }));

var user = await graphClient.Me.Request().GetAsync();
```

3. **Apply Best Practices**:
   - **Efficiency**: Use batching to reduce API calls.
   - **Security**: Implement proper authentication and handle permissions effectively.
   - **Scalability**: Understand throttling and implement error handling to manage service limits.

Remember, Microsoft Graph is like a Swiss Army knife for developers; it's got a tool (or API) for nearly every scenario you can think of within the Microsoft ecosystem.

Discover Microsoft Graph

Overview

- **Function**: Microsoft Graph serves as the primary interface for accessing and manipulating data in Microsoft 365, Windows 10, and Enterprise Mobility + Security.

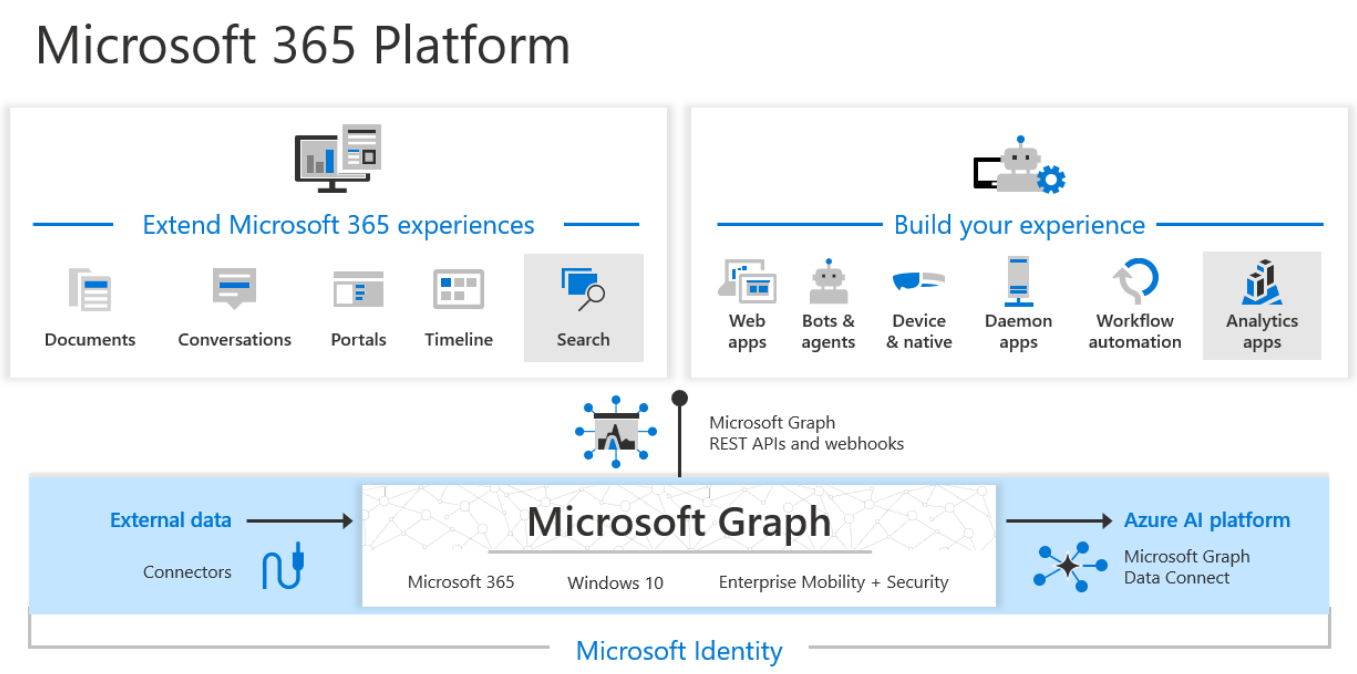
Key Components

1. **Microsoft Graph API**:
   - **Single Endpoint**: `https://graph.microsoft.com`
   - **Access**: Available via REST APIs or SDKs.
   - **Capabilities**: Manages identity, access, compliance, security, and protects against data leaks.

2. **Microsoft Graph Connectors**:
    - **Data Integration**: Brings external data into Microsoft Graph for enhanced Microsoft 365 experiences.
    - **Examples**: Connectors for Box, Google Drive, Jira, Salesforce, etc., enhancing Microsoft Search.
3. **Microsoft Graph Data Connect**:
    - **Data Delivery**: Facilitates moving Microsoft Graph data to Azure data stores for scalable, secure use.
    - **Purpose**: Enables building intelligent apps using cached Microsoft 365 data.

Benefits and Applications

- **Unified Data Access**: One API to access data across various Microsoft services.
- **Extensibility**: Allows developers to extend Microsoft 365 functionalities or create new intelligent applications.
- **Security and Compliance**: Built-in features to manage user and device identity, ensuring data security.



Remember, Microsoft Graph is like a universal adapter for Microsoft services; it lets your applications plug into the vast sea of Microsoft data with ease.

Query Microsoft Graph by Using REST

Completion: Completed

Experience Points: 100 XP

Duration: 3 minutes

Overview

- **Definition**: Microsoft Graph is a RESTful web API for accessing Microsoft Cloud service resources.
- **Prerequisites**: You must register your app and acquire authentication tokens.

API Structure

- **Namespaces**: Most resources and methods are in `microsoft.graph`. Some resources like `callRecord` are in sub-namespaces.

Making Requests

Request Structure

```
{HTTP method} https://graph.microsoft.com/{version}/{resource}?{query-parameters}
```

- **Components**:
  - **{HTTP method}**: GET, POST, PUT, PATCH, DELETE, etc.
  - **{version}**: API version, e.g., `v1.0` or `beta`.
  - **{resource}**: The Microsoft Graph resource you're interacting with, e.g., `users`, `me/messages`.
  - **{query-parameters}**: OData query options or custom parameters (`$select`, `$filter`, etc.).

Response Structure

- **Status Code**: HTTP status indicating the result of the request.
- **Response Message**: Data returned, could be JSON for data operations.
- **nextLink**: For large datasets, this field provides a URL for the next set of results (`@odata.nextLink`).

Example

Fetching User Details

```
GET https://graph.microsoft.com/v1.0/me?$select=displayName,mail
```

- **Response**:

```
{
  "@odata.context":
"https://graph.microsoft.com/v1.0/$metadata#users(displayName,mail)/$entity"
,
  "displayName": "John Doe",
  "mail": "john.doe@example.com"
}
```

Key Points

- **Authentication**: Always required before making API calls.

- **Namespaces**: Be aware of the namespace when dealing with specific resources.
- **Paging**: Use `@odata.nextLink` for accessing more results in large data sets.

Remember, interacting with Microsoft Graph via REST is like having a conversation with Microsoft 365's data in a language it understands perfectly – just make sure you have your credentials ready, akin to having the right passport for international travel.

HTTP Methods in Microsoft Graph

Overview

- **Function**: HTTP methods define actions on resources within Microsoft Graph.

Supported HTTP Methods

Method	Description
GET	Retrieve data from a resource.
POST	Create new resources or execute actions.
PATCH	Update a resource with specific changes.
PUT	Replace an entire resource with new data.
DELETE	Remove a resource.

Note:

- **GET** and **DELETE** do not require a request body.
- **POST**, **PATCH**, and **PUT** methods necessitate a JSON-formatted body to specify property values or action parameters.

API Versions

- **v1.0**: For stable, generally available APIs. Use for production apps.
- **beta**: For preview features which might change. Use only for development and testing.

Resources

- **Entities vs. Complex Types**:
  - **Entities** always have an `id` property.
  - Examples of resources: `me`, `user`, `group`, `drive`, `site`.
- **Relationships**: Access related resources via URLs like `me/messages`, `me/drive`.
- **Permissions**: Different resources require different permission levels, especially for create/update vs. read operations.

Query Parameters

- **Purpose**: Customize API responses.
- **Types**:
  - **OData System Query Options**: Modify the data returned, like `$select`, `$filter`.

Example Query: Fetching messages with a specific email address filter:

```
GET https://graph.microsoft.com/v1.0/me/messages?filter=emailAddress eq
'jon@contoso.com'
```

Remember, when interacting with Microsoft Graph, it's like being a diplomat with HTTP methods as your tools of negotiation, ensuring you're asking for or providing exactly what's needed.

Query Microsoft Graph by Using SDKs

Overview of Microsoft Graph SDKs

- **Components**:
  - **Service Library**: Generated from Microsoft Graph metadata, includes models and request builders.
  - **Core Library**: Enhances interactions with features like retry handling, authentication, etc.

Key Features of the Core Library

- **Retry Handling**: Automatically manages retries for failed requests.
- **Secure Redirects**: Handles HTTP redirects in a secure manner.
- **Transparent Authentication**: Manages token acquisition and refresh.
- **Payload Compression**: Reduces data transfer size for efficiency.
- **Paging through Collections**: Simplifies handling of large data sets.
- **Batch Requests**: Allows grouping multiple operations into a single HTTP request.

Using the .NET SDK

Installation

- **NuGet Packages**:
  - `Microsoft.Graph`: For v1.0 endpoint, includes fluent API support.
  - `Microsoft.Graph.Beta`: For beta endpoint access.
  - `Microsoft.Graph.Core`: Core functionality for Graph calls.

Creating a Microsoft Graph Client

- **Authentication**: Uses an authentication provider for token management.

Example Code: Creating a Graph client with device code flow:

```
using Microsoft.Graph;
using Azure.Identity;
```

```
var scopes = new[] { "User.Read" };
var tenantId = "common"; // For multi-tenant apps, single-tenant apps use tenant
ID
var clientId = "YOUR_CLIENT_ID";

var options = new TokenCredentialOptions
{
    AuthorityHost = AzureAuthorityHosts.AzurePublicCloud
};

Func<DeviceCodeInfo, CancellationToken, Task> callback = (code, cancellation) => {
    Console.WriteLine(code.Message);
    return Task.FromResult(0);
};

var deviceCodeCredential = new DeviceCodeCredential(
    callback, tenantId, clientId, options);

var graphClient = new GraphServiceClient(deviceCodeCredential, scopes);
```

Note:

- This example uses `DeviceCodeCredential` which is suitable for scenarios where user interaction is required to authenticate.
- **Authentication Providers**: Choose the right provider based on your application's scenario. Different providers support different authentication flows.

Remember, using SDKs is like having a well-equipped kitchen; they provide the tools to cook up a storm of API interactions, making your development process smoother and more efficient.

Operations with Microsoft Graph

Read Information

- **To retrieve a single entity**:

```
// GET https://graph.microsoft.com/v1.0/me
var user = await graphClient.Me.GetAsync();
```

Retrieve a List of Entities

- **Fetching with additional options like filtering and sorting**:

```
// GET /me/messages?$select=subject,sender&$filter=<some
condition>&$orderBy=receivedDateTime
var messages = await graphClient.Me.Messages
```

```
    .Request()
    .Select(m => new {
        m.Subject,
        m.Sender
    })
    .Filter("<filter condition>")
    .OrderBy("receivedDateTime")
    .GetAsync();
```

Delete an Entity

- **Deleting an entity from Microsoft Graph**:

```
// DELETE /me/messages/{message-id}
string messageId = "AQMkAGUy...";
var message = await graphClient.Me.Messages[messageId]
    .Request()
    .DeleteAsync();
```

Create a New Entity

- **Adding a new item to a collection**:

```
// POST /me/calendars
var calendar = new Calendar
{
    Name = "Volunteer"
};

var newCalendar = await graphClient.Me.Calendars
    .Request()
    .AddAsync(calendar);
```

Key Points:

- **GET**: Used for reading or retrieving information.
- **DELETE**: For removing entities, requires the entity's identifier.
- **POST**: To create a new entity, often used in collections like calendars or messages.
- **SDK Features**:
    - Fluent API in .NET SDK allows chaining methods for request configuration.
    - Query parameters like `$filter`, `$select`, and `$orderBy` can customize the data retrieved from Graph.

Remember, interacting with Microsoft Graph is like gardening with an API; you plant requests, nurture them with proper methods and parameters, and soon enough, you'll harvest the data you need.

Best Practices for Microsoft Graph

Authentication

- **Token Acquisition**: Use **OAuth 2.0** tokens.
    - **Presentation**:
        - As a Bearer token in the HTTP Authorization header.
        - Or via the graph client constructor in SDKs.
- **Tool**: Use **Microsoft Authentication Library (MSAL)** for token acquisition.

Consent and Authorization

Best Practices

1. **Least Privilege**:
    - **Permission Request**: Only ask for permissions needed for your app's functionality.
    - **Example**: For user creation, use minimal permissions as described in API documentation.
2. **Permission Types**:
    - **Delegated**: For user interactive apps where a user is signed in.
    - **Application**: For services/daemon applications running without user context.

   Caution:
    - Avoid using application permissions in interactive scenarios to prevent security and compliance risks.
3. **User and Admin Experience**:
    - **Consent**: Ensure clarity on who consents (users or admins) and adjust permission requests accordingly.
    - **Consent Types**:
        - **Static**: Permissions requested at app registration.
        - **Dynamic**: Permissions can change based on user interaction.
        - **Incremental**: Request permissions as needed during runtime.
4. **Multi-Tenant Considerations**:
    - **Admin Controls**:
        - Some tenants might restrict user consent, requiring admin consent.
        - Admins might set policies restricting certain operations, so prepare for `403 Forbidden` errors.

Note:

- Always verify user's permissions to ensure compliance with tenant policies.

- Design your application with flexibility to handle various consent and authorization scenarios across different tenants.

Remember, when dealing with Microsoft Graph, think of it like borrowing tools from a neighbor; you only take what you need, return them in good condition, and respect the rules of their house (or in this case, their tenant).

Handling Responses Effectively with Microsoft Graph

Key Practices for Response Handling

Pagination

- **Expect Paging**: When querying collections, results are often returned in pages due to server limits.
- **Handling Paging**:
  - Use the `@odata.nextLink` property to fetch subsequent pages.
  - **Example**:

```
var usersPage = await graphClient.Users
    .Request()
    .GetAsync();

while (usersPage.NextPageRequest != null)
{
    usersPage = await usersPage.NextPageRequest.GetAsync();
    // Process users from the current page
}
```

  - **End of Paging**: The final page will not have an `@odata.nextLink`.

Evolvable Enumerations

- **Impact**: New enum members can be added without breaking existing applications.
- **Handling**:
  - By default, GET returns only known enum members.
  - **Opt-in for Unknown Members**:

```
var requestHeaders = new Dictionary<string, string>
{
    { "Prefer", "odata.include-
annotations=\"OData.Community.Display.V1.FormattedValue\"" }
};
var response = await graphClient.Users
    .Request()
    .Header("Prefer", requestHeaders["Prefer"])
    .GetAsync();
```

- **Best Practice**: Design your app to handle both known and unknown enum values for future-proofing.

Storing Data Locally

- **Real-Time Data**: Prefer real-time retrieval from Microsoft Graph.
- **Caching Considerations**:
  - Only cache data when necessary and ensure it aligns with your terms of use and privacy policies.
  - **Retention and Deletion**: Implement policies for data management to comply with Microsoft APIs Terms of Use and privacy regulations.

Note:

- Always ensure your application respects Microsoft Graph's terms regarding data storage and use.
- Being prepared for pagination and evolvable enums helps in creating robust applications that adapt to Microsoft Graph's evolving API structure.

AZ-204: Implement Secure Azure Solutions

Overview

- **Objective**: Learn techniques to deploy applications securely in Azure using various Azure services for security management.

Key Topics Covered

1. **Azure Key Vault**:
   - Securely store and access secrets, keys, and certificates.
   - Manage sensitive data like API keys, passwords, and database strings.
2. **Managed Identities**:
   - Provide Azure services with an automatically managed identity in Azure AD.
   - Simplify security management by removing the need for credentials in your code.
3. **Azure App Configuration**:
   - Centralize configuration settings for your applications.
   - Manage application settings and feature flags at scale.
4. **Azure Role-Based Access Control (RBAC)**:
   - Control access to Azure resources based on roles.
   - Implement fine-grained access management.

Prerequisites

- **Experience**:

    - At least one year of experience developing scalable solutions across all phases of software development.

- **Knowledge**:
    - Basic understanding of Azure, cloud concepts, services, and the Azure portal.
- **Recommended**:
    - Complete AZ-900: Azure Fundamentals if you're new to Azure or cloud computing.

Additional Notes

- This module focuses on security practices essential for deploying applications in the cloud.
- Proper understanding and implementation of these services can significantly enhance the security posture of your applications in Azure.

Remember, securing your Azure solutions is like setting up a high-tech security system for your digital assets; you want to make sure only the right entities have access at the right times with the right permissions.

Introduction to Azure Key Vault

Overview

- **Azure Key Vault**: A service for secure storage and access management of secrets like API keys, passwords, certificates, and cryptographic keys.

Learning Outcomes

Upon completion, you will be able to:

1. **Understand the Benefits**:
    - **Centralized Secret Management**: One place to manage all secrets, reducing the risk of accidental exposure.
    - **Secure Access**: Control access with Azure's Identity and Access Management (IAM) capabilities.
    - **Compliance**: Helps in meeting regulatory compliance requirements.
2. **Authentication to Azure Key Vault**:
    - **Methods**:
        - **Azure AD Authentication**: Users, applications, and services can authenticate using Azure AD.
        - **Service Principal**: Used for applications and services to access Key Vault.
3. **Secret Management with Azure CLI**:
    - **Set a Secret**:

```
az keyvault secret set --vault-name "<your-unique-keyvault-name>" --
name "ExamplePassword" --value "hVFkk96"
```

- **Retrieve a Secret**:

```
az keyvault secret show --name "ExamplePassword" --vault-name "<your-
unique-keyvault-name>" --query value -o tsv
```

Note: Always ensure you replace `<your-unique-keyvault-name>` with your actual Key Vault name.

Remember, Azure Key Vault is like a digital safe for your most sensitive data. You wouldn't leave your house keys out in the open, so why would you with your API keys or passwords?

Explore Azure Key Vault

Containers in Azure Key Vault

- **Vaults**: Store software and HSM-backed keys, secrets, and certificates.
- **Managed HSM Pools**: Only for HSM-backed keys.

Key Vault Services

1. **Secrets Management**:
   - Secure storage for tokens, passwords, API keys, etc.
2. **Key Management**:
   - Centralized creation and control of encryption keys.
3. **Certificate Management**:
   - Manage SSL/TLS certificates for Azure and internal resources.

Service Tiers

- **Standard**: Uses software key encryption.
- **Premium**: Includes HSM-protected keys. For pricing details, refer to Azure Key Vault pricing page.

Benefits of Azure Key Vault

- **Centralized Secrets**:
  - Keeps application secrets in one place, reduces distribution risks.
- **Security**:
  - **Authentication**: Uses Microsoft Entra ID.

    - **Authorization**:
        - **Azure RBAC**: For vault management and data access.
        - **Key Vault Access Policy**: For data access only.

- **Monitoring and Logging**:
    - Logs can be stored, streamed, or sent to Azure Monitor.
    - Control over log access and lifecycle.

- **Simplified Administration**:
    - No need for deep HSM knowledge.
    - Automatic scaling and high availability through regional replication.
    - Integration with Azure's administrative tools.
    - Automation of certificate management tasks.

Additional Features

- **Scalability**: Easily scales with organizational needs.
- **Automation**: Handles certificate lifecycle tasks like enrollment and renewal.

Remember, Azure Key Vault is like your digital vault in the cloud where you can lock away your digital treasures, ensuring they're only accessible with the right keys and permissions.

Discover Azure Key Vault Best Practices

Authentication Methods

1. **Managed Identities for Azure Resources**:
    - **Recommended**: Assign identity to resources like VMs, enabling automatic secret rotation.
    - **Benefit**: No need for manual secret management.

2. **Service Principal with Certificate**:
    - **Use**: Certificate-based authentication.
    - **Drawback**: Certificate rotation responsibility on the application owner/developer.

3. **Service Principal with Secret**:
    - **Not Recommended**: Due to challenges with automatic secret rotation.

Data Security in Transit

- **TLS Enforcement**: Key Vault uses TLS for data protection during transit.
- **Perfect Forward Secrecy (PFS)**: Ensures each session's key is unique, enhancing security against interception.
- **Encryption**: RSA-based 2,048-bit key length for strong encryption.

Best Practices

- **Separate Vaults**:
    - Use different vaults for each application and environment (Dev, Pre-Prod, Prod) to avoid cross-environment secret sharing and reduce risk in case of a breach.
- **Access Control**:
    - **Principle**: Only allow authorized applications and users to access Key Vault.
    - **Tools**: Use Azure RBAC or Key Vault access policies for fine-grained access control.
- **Backup**:
    - Regularly backup your vault, especially upon significant changes like updates, deletions, or creations.
- **Logging**:
    - **Enable Logging**: For monitoring activities within Key Vault.
    - **Alerts**: Set up alerts for unusual or critical events.
- **Recovery Options**:
    - **Soft-Delete**: Enable to recover deleted secrets.
    - **Purge Protection**: Protect against immediate, forced deletions.

Remember, treating your Azure Key Vault like a high-security vault in a bank ensures your digital assets remain under lock and key, accessible only to those with the right credentials and permissions.

Authenticate to Azure Key Vault

Authentication Overview

- **Method**: Uses Microsoft Entra ID for identity verification.

Obtaining a Service Principal

1. **Managed Identity**:
    - **System-Assigned**: Automatically manages service principal for your application. **Recommended for use**.
2. **Manual Registration**:
    - Register the application in Microsoft Entra tenant if managed identity isn't feasible.

Authentication in Application Code

- **Azure Identity Client Library**: Simplifies authentication across different environments.
    - **SDKs Available For**:
        - **.NET**: Azure Identity SDK .NET
        - **Python**: Azure Identity SDK Python

- **Java**: Azure Identity SDK Java
- **JavaScript**: Azure Identity SDK JavaScript

Authentication Using REST

- **HTTP Authorization Header**:
  - Include the access token as a Bearer token.

```
PUT /keys/MYKEY?api-version=<api_version>  HTTP/1.1
Authorization: Bearer <access_token>
```

Error Handling

- **401 Not Authorized**: Returned if token is missing or invalid.
- **WWW-Authenticate Header**: Provided in error response to guide token acquisition:

```
401 Not Authorized
WWW-Authenticate: Bearer authorization="…", resource="…"
```

- **Parameters**:
  - **authorization**: URL for obtaining an access token.
  - **resource**: The Key Vault endpoint (`https://vault.azure.net`).

Note:

- Using managed identities is the best practice for effortless authentication management.
- For direct REST API calls, ensure the access token is included in the `Authorization` header correctly to avoid authentication errors.

Remember, authentication to Key Vault is like showing your ID at the bank; you need the right credentials (or tokens) to access your secure deposit box safely.

Implement Managed Identities

Overview

- **Objective**: Understand and implement managed identities in Azure for enhancing security by eliminating credential management.

Key Topics to Cover

1. **Introduction to Managed Identities**:
   - **Definition**: An Azure AD Identity for Azure resources, allowing them to authenticate to services without credentials in your code.

2. **Types of Managed Identities**:
    - **System-Assigned**:
        - Tied to the lifecycle of an Azure resource.
        - Automatically managed by Azure.
    - **User-Assigned**:
        - Created as a standalone resource.
        - Can be assigned to multiple Azure resources.
3. **Benefits**:
    - **No Credential Management**: Azure manages the identities, reducing security risks.
    - **Centralized Control**: Manage identity permissions through Azure RBAC.
    - **Secure Communication**: Between Azure services without exposing secrets.
4. **Implementing Managed Identities**:
    - **Configuration**:
        - Enable on resources like VMs, App Services, or Azure Functions.
    - **Authentication**:
        - Use managed identities to access other Azure resources or services like Key Vault.
5. **Practical Examples**:
    - **Accessing Azure Key Vault**:
        - Use managed identities to fetch secrets or keys securely.
    - **Accessing Azure Storage**:
        - Authenticate storage operations without managing access keys.
6. **Security Considerations**:
    - **Least Privilege**: Assign only necessary permissions.
    - **Monitoring and Logging**: Track usage of managed identities.
7. **Troubleshooting and Best Practices**:
    - **Common Issues**: Identity not found, permission errors.
    - **Best Practices**: Use system-assigned when possible, understand lifecycle implications.

Next Steps

- **Begin with Unit 1**: Get a foundational knowledge of managed identities.
- **Progress through Each Unit**: Learn how to set up, use, and manage managed identities effectively.

Remember, managed identities are like having an Azure butler; they take care of your authentication needs so you can focus on developing your application without worrying about key management.

Introduction to Managed Identities

Completion: Completed

Experience Points: 100 XP

Duration: 3 minutes

Overview

- **Challenge**: Managing secrets and credentials for secure communication within solutions.
- **Solution**: Azure Managed Identities reduce this burden by eliminating credential management.

Learning Outcomes

After completing this module, you'll be able to:

1. **Differentiate Managed Identity Types**:
    - **System-Assigned**:
        - Directly linked to an Azure resource.
        - Lifecycle tied to the resource.
    - **User-Assigned**:
        - Created independently and can be assigned to multiple resources.
        - Lifecycle managed separately from resources.
2. **Understand Identity Flows**:
    - **System-Assigned**: Identity is created when the resource is created and deleted with it.
    - **User-Assigned**: Identity can exist before or after the resources it's assigned to.
3. **Configure Managed Identities**:
    - Enabling and assigning identities to resources like Azure VMs, App Services, or Functions.
4. **Use Managed Identities**:
    - **REST API**:
        - Retrieve tokens for authentication without credentials.
        - **Example**:

```
GET /metadata/identity/oauth2/token?api-version=2018-02-
01&resource=https%3A%2F%2Fmanagement.azure.com%2F HTTP/1.1
Metadata: true
```

    - **Code**:
        - Using Azure SDKs to get tokens programmatically.
        - **Example in C#**:

```
var azureServiceTokenProvider = new AzureServiceTokenProvider();
string accessToken = await
azureServiceTokenProvider.GetAccessTokenAsync("https://management.
azure.com/");
```

Remember, managed identities in Azure are like digital identity cards for your resources, allowing them to prove who they are to other services without you having to manage any passwords or keys.

Explore Managed Identities

Overview

- **Challenge**: Handling secrets, credentials, and keys for service-to-service communication.
- **Solution**: Managed identities remove the need for credential management by providing automatic identities.

Role of Managed Identities

- **Purpose**: To allow Azure services to access other Azure services like Azure Key Vault securely without managing credentials directly.

Types of Managed Identities

1. **System-Assigned Managed Identity**:
    - **Creation**: Automatically created when enabled on an Azure service instance.
    - **Lifecycle**: Closely tied to the lifecycle of the Azure resource.
        - **Creation**: Identity is provisioned when the resource is created.
        - **Deletion**: Identity and credentials are automatically cleaned up when the resource is deleted.
2. **User-Assigned Managed Identity**:
    - **Creation**: Created as an independent Azure resource.
    - **Assignment**: Can be assigned to multiple Azure service instances.
    - **Lifecycle**: Managed separately from the resources it's assigned to:
        - **Can be created before or after the resources it will be used with.**
        - **Deletion** of the managed identity does not affect the resources it's assigned to, but it will remove the ability to authenticate with that identity.

Important Note: Managed identities are a special type of service principal in Microsoft Entra ID, designed to be used only with Azure resources. Upon deletion of the managed identity, the corresponding service principal is also removed.

Remember, using managed identities is like having a digital butler for your Azure services; they take care of the authentication so you can focus on the functionality of your application.

Characteristics of Managed Identities

Comparison Table

Property	System-assigned managed identity	User-assigned managed identity
Creation	Created with an Azure resource (e.g., VM, App Service)	Created independently as an Azure resource
Lifecycle	Tied to the lifecycle of the Azure resource it's created with.	Independent lifecycle; needs explicit deletion.
Sharing across Azure resources	Cannot be shared; linked to a single resource	Can be shared; assigned to multiple resources.

Common Use Cases

- **System-assigned Managed Identity**:
  - **Single Resource Workloads**: Ideal for applications running on a single resource like a virtual machine.
  - **Independent Identities**: When each component needs its own identity.
- **User-assigned Managed Identity**:
  - **Multiple Resources**: For workloads that span across multiple resources sharing the same identity.
  - **Preauthorization**: Useful in scenarios where resources need pre-authorized access during provisioning.
  - **Consistent Permissions**: When resources are frequently recycled but permissions need to remain.

Support in Azure Services

- **Compatibility**: Managed identities can authenticate to any service supporting Microsoft Entra (Azure AD) authentication.
- **Supported Services**: Check the list of Azure services supporting this feature at Services that support managed identities for Azure resources.

Note:

- While this module uses Azure virtual machines as examples, the principles apply broadly to any Azure resource that supports Microsoft Entra authentication.

Remember, managed identities are like specialized keys in Azure; they unlock secure access for your resources without you needing to manage the keyring.

Discover the Managed Identities Authentication Flow

System-Assigned Managed Identity Flow with Azure VM

1. **Enable Identity**:
    - **Request**: Azure Resource Manager (ARM) receives a request to enable system-assigned managed identity on a VM.
2. **Service Principal Creation**:
    - ARM creates a service principal for the VM in Microsoft Entra ID within the trusted tenant of the subscription.
3. **Identity Configuration**:
    - ARM updates the Azure Instance Metadata Service (IMDS) identity endpoint with the service principal's client ID and certificate.
4. **Access Granting**:
    - **Azure Resource Manager**: Use RBAC in Microsoft Entra ID to assign roles to the VM's service principal.
    - **Key Vault**: Grant access to specific secrets or keys in Key Vault for the VM's identity.
5. **Token Request**:
    - **From VM**: Code running on the VM requests a token from the IMDS endpoint:

      ```
      http://169.254.169.254/metadata/identity/oauth2/token
      ```
6. **Token Acquisition**:
    - The VM makes a call to Microsoft Entra ID to request an access token using the client ID and certificate. Microsoft Entra ID responds with a JSON Web Token (JWT).
7. **Token Usage**:
    - The code then uses this access token to authenticate calls to Azure services that support Microsoft Entra authentication.

Note:

- This flow ensures that the virtual machine can securely access other Azure resources or services like Azure Key Vault without exposing any credentials in your code or configuration.

Remember, this process is like a VM showing its ID card to get entry into other Azure services' exclusive clubs without needing to carry around the actual membership card.

User-Assigned Managed Identity Flow with Azure VM

1. **Create Identity**:

- **Request**: Azure Resource Manager (ARM) receives a request to create a user-assigned managed identity.

2. **Service Principal Creation**:
   - ARM creates a service principal for this identity in Microsoft Entra ID within the trusted tenant of the subscription.
3. **Identity Assignment**:
   - ARM receives a request to assign this user-assigned managed identity to a VM. The VM's Azure Instance Metadata Service (IMDS) identity endpoint is updated with the identity's client ID and certificate.
4. **Access Granting** (Can be done before or after step 3):
   - **Azure Resource Manager**: Assign roles to the service principal using RBAC in Microsoft Entra ID.
   - **Key Vault**: Grant the identity access to specific secrets or keys.
5. **Token Request**:
   - **From VM**: Code running on the VM requests an access token from the IMDS endpoint:

     ```
     http://169.254.169.254/metadata/identity/oauth2/token
     ```
6. **Token Acquisition**:
   - The VM uses the client ID and certificate from step 3 to request an access token from Microsoft Entra ID. Entra ID issues a JSON Web Token (JWT) in response.
7. **Token Usage**:
   - The application code on the VM uses this JWT to authenticate when calling Azure services that support Microsoft Entra authentication.

Note:

- The flexibility of user-assigned managed identities allows for sharing the same identity across multiple resources, promoting consistency in permissions even when resources are frequently created or deleted.

Remember, user-assigned identities are like freelance security guards; they can work for multiple VMs, providing a consistent security policy across your Azure environment.

Configure Managed Identities for Azure VMs

System-Assigned Managed Identity

Prerequisites

- Your account must have the **Virtual Machine Contributor** role assignment.

Enable During VM Creation

- **Command**: Use Azure CLI to create a VM with a system-assigned managed identity.

```
az vm create --resource-group myResourceGroup \
    --name myVM --image win2016datacenter \
    --generate-ssh-keys \
    --assign-identity \
    --role contributor \
    --scope mySubscription \
    --admin-username azureuser \
    --admin-password myPassword12
```

  - **Parameters**:
    - `--assign-identity`: Enables system-assigned identity.
    - `--role contributor`: Assigns the VM the contributor role.
    - `--scope`: Specifies the resource scope for the role assignment.

Enable on Existing VM

- **Command**: Assign the system-assigned identity to a VM that's already created.

```
az vm identity assign -g myResourceGroup -n myVm
```

Note:

- Enabling managed identity during VM creation or on an existing VM involves similar steps but requires different commands depending on whether the VM is new or already exists.

Remember, configuring managed identities is like setting up automatic identification for your VM, making it easier for it to access other Azure services securely without you manually managing the credentials.

User-Assigned Managed Identity Configuration

Prerequisites

- **Roles**: Your account needs **Virtual Machine Contributor** and **Managed Identity Operator** role assignments.

Steps to Enable User-Assigned Managed Identity

1. Create the User-Assigned Identity

- **Command**: Use Azure CLI to create a new user-assigned managed identity.

```
az identity create -g myResourceGroup -n myUserAssignedIdentity
```

- **-g**: Resource group name
- **-n**: Name of the identity

2. Assign the Identity

During VM Creation

- **Command**: Create a VM with a user-assigned identity.

```
az vm create \
--resource-group <RESOURCE GROUP> \
--name <VM NAME> \
--image Ubuntu2204 \
--admin-username <USER NAME> \
--admin-password <PASSWORD> \
--assign-identity <USER ASSIGNED IDENTITY NAME> \
--role <ROLE> \
--scope <SUBSCRIPTION>
```

To an Existing VM

- **Command**: Assign a user-assigned identity to an existing VM.

```
az vm identity assign \
    -g <RESOURCE GROUP> \
    -n <VM NAME> \
    --identities <USER ASSIGNED IDENTITY>
```

Azure SDKs Support for Managed Identities

- **.NET**: Manage resources from a VM with managed identities.
- **Java**: Manage storage from a VM with managed identities.
- **Node.js**: Create VM with system-assigned managed identity.
- **Python**: Create VM with system-assigned managed identity.
- **Ruby**: Create VM with system-assigned managed identity.

Note:

- User-assigned identities provide flexibility to be shared across multiple resources, allowing for centralized management of identities and permissions.

Remember, assigning a user-assigned identity to a VM is like giving it a custom ID badge that it can carry to different services, allowing it to access resources without the need for shared secrets or credentials.

Acquire an Access Token with Managed Identities

Overview

- **Purpose**: Allow client applications to request an app-only access token for Azure resources.

Using DefaultAzureCredential

- **Library**: Azure Identity library includes `DefaultAzureCredential`.
- **Functionality**: Attempts multiple authentication mechanisms in order, stopping at the first successful one.

Authentication Mechanisms Order

1. **Environment Variables**: Reads credentials from environment variables.
2. **Managed Identity**: Uses if the app is on an Azure host with managed identity.
3. **Visual Studio**: Utilizes credentials from Visual Studio if authenticated.
4. **Azure CLI**: Checks if a user has logged in via `az login`.
5. **Azure PowerShell**: Checks for `Connect-AzAccount` authentication.
6. **Interactive Browser**: Uses system browser for interactive authentication (disabled by default).

Note:

- `DefaultAzureCredential` is designed for ease in common scenarios, but for more control, other credential types might be used.

Adding Azure.Identity to Your Project

- **Command**: To add the Azure Identity SDK to a .NET project:

```
dotnet add package Azure.Identity
```

Benefits

- **Development to Production**: The same code works in both environments without changes:
  - Development: Authenticates with your personal credentials.
  - Production: Authenticates via managed identity.

Remember, `DefaultAzureCredential` is like an all-in-one Swiss Army knife for authentication; it tries various tools until it finds the one that works, making your life easier during development and deployment.

Managing Authentication with Azure SDK

Using DefaultAzureCredential

- **For Key Vault Secrets**:

```
// Create a secret client using the DefaultAzureCredential
var client = new SecretClient(new Uri("https://myvault.vault.azure.net/"),
new DefaultAzureCredential());
```

- **With User-Assigned Managed Identity**:

```
string userAssignedClientId = "<your managed identity client Id>";
var credential = new DefaultAzureCredential(new
DefaultAzureCredentialOptions { ManagedIdentityClientId =
userAssignedClientId });

var blobClient = new BlobClient(new
Uri("https://myaccount.blob.core.windows.net/mycontainer/myblob"),
credential);
```

Custom Authentication with ChainedTokenCredential

- **For Customized Authentication Flow**:
  - Here, `ChainedTokenCredential` allows you to define a custom sequence of credential providers.

```
// Authenticate using managed identity if it is available; otherwise use the
Azure CLI to authenticate.

var credential = new ChainedTokenCredential(new ManagedIdentityCredential(),
new AzureCliCredential());

var eventHubProducerClient = new
EventHubProducerClient("myeventhub.eventhubs.windows.net", "myhubpath",
credential);
```

Note:

- **DefaultAzureCredential** simplifies development by trying multiple authentication methods, ideal for standard use cases.
- **ChainedTokenCredential** offers more control, allowing you to specify the order of authentication attempts, which is beneficial in scenarios requiring specific fallback behaviors.

Remember, using these credentials is like having an authentication Swiss Army knife; you can pull out the right tool for the job, whether it's the simplest approach or a custom sequence tailored to your environment's needs.

Implement Azure App Configuration

Overview

- **Objective**: Understand how to use Azure App Configuration to manage application settings and features centrally.

Key Topics to Cover

1. **Introduction to Azure App Configuration**:
    - **Concept**: A managed service for centralizing app configurations.
    - **Benefits**: Central management, security, and dynamic updates of settings.
2. **Creating an App Configuration Store**:
    - **Process**: Steps to set up a new configuration store in Azure.
3. **Managing Configuration Data**:
    - **Key-Value Pairs**: How to add, modify, and retrieve settings.
    - **Feature Flags**: Use for feature toggles in applications.
4. **Security and Access Control**:
    - **Authentication**: Using Azure AD for access.
    - **Authorization**: Role-based access control for data in App Configuration.
5. **Integration with Applications**:
    - **Using SDKs**: How to integrate with different programming languages (e.g., .NET, Java).
    - **Dynamic Configuration**: Techniques for apps to detect and respond to config changes.
6. **Advanced Features**:
    - **Labels and Tags**: Organize and version configurations.
    - **Event Grid Integration**: React to configuration changes in real-time.
7. **Best Practices**:
    - **Environment Separation**: Use different configurations for different environments.
    - **Secret Management**: How to handle sensitive information in configuration.

Next Steps

- **Begin with Unit 1**: Gain foundational knowledge of Azure App Configuration.
- **Progress through Each Unit**: Learn about setting up, securing, and using App Configuration effectively in your applications.

Remember, Azure App Configuration is like having a centralized control panel for your application's settings and features, allowing you to adjust the behavior of your apps without redeployment.

Introduction to Azure App Configuration

Completion: Completed

Experience Points: 100 XP

Duration: 3 minutes

Overview

- **Service**: Azure App Configuration is designed to manage application settings and feature flags centrally.

Learning Outcomes

After completing this module, you'll be able to:

1. **Understand Benefits**:
    - **Centralized Management**: Easily manage settings across various environments and applications.
    - **Security**: Securely store configuration data with Azure's security features.
    - **Dynamic Updates**: Push configuration changes without redeploying the application.
    - **Feature Flags**: Enable or disable features quickly across services.
2. **Storage in Azure App Configuration**:
    - **Key-Value Pairs**: Store configuration as simple key-value data.
    - **Labels**: Tag configurations for different purposes (e.g., different environments).
    - **Snapshots**: Capture the state of configuration at a point in time for consistency.
3. **Implement Feature Management**:
    - **Feature Flags**: Use to toggle features on/off in production with minimal impact.
    - **A/B Testing**: Control which users see new features.
4. **Security and Access**:
    - **Authentication**: Integrate with Azure Active Directory for secure access.
    - **Authorization**: Use Azure role-based access control (RBAC) to manage who can access or modify configurations.
    - **Encryption**: Data at rest and in transit is encrypted.

Note:

- Azure App Configuration helps in keeping your application's configuration out of your codebase, reducing the risk of exposing sensitive information and simplifying configuration management.

Remember, think of Azure App Configuration as your app's external brain; it remembers all the settings and features you might need to tweak or change on the fly, without any need to rewire your application.

Explore the Azure App Configuration Service

Overview

- **Service Description**: Azure App Configuration centralizes the management of application settings and feature flags, simplifying configuration management for distributed applications.

Key Benefits

- **Quick Setup**: A fully managed service with minimal setup time.
- **Flexible Key Structure**: Allows for hierarchical and flexible key representations.
- **Labeling**: Tag configurations for easy organization and versioning.
- **Historical Data**: Point-in-time replay of settings for auditing or rollback.
- **Feature Flags**: Dedicated UI for managing feature toggles.
- **Configuration Comparison**: Compare configurations based on custom dimensions.
- **Security**: Uses Azure-managed identities, ensures data encryption.
- **Framework Integration**: Seamlessly integrates with popular development frameworks.

Azure App Configuration vs. Azure Key Vault

- **App Configuration**: For non-sensitive configuration data, feature flags.
- **Key Vault**: For secrets like connection strings, API keys.

Scenarios Enabled by App Configuration

- **Central Management**: Manage configuration data across different environments and regions.
- **Dynamic Updates**: Modify app settings on-the-fly without redeployment.
- **Feature Control**: Toggle features in real-time for quick rollouts or rollbacks.

Using App Configuration

Client Libraries for Integration

- **.NET**:
    - **ASP.NET Core**: Use the App Configuration provider for .NET.
    - **.NET Framework and ASP.NET**: Utilize the App Configuration builder for .NET.
- **Java Spring**: Integrate with the App Configuration provider for Spring Cloud.
- **JavaScript/Node.js**: Use the App Configuration provider for JavaScript.
- **Python**: Implement with the App Configuration provider for Python.
- **Other Languages**: Use the App Configuration REST API for broader compatibility.

Note:

- Choosing the right method to connect to App Configuration depends on your application's programming language and framework, ensuring seamless integration and management of configuration settings.

Remember, Azure App Configuration is like your app's personal assistant; it keeps all your settings organized, accessible, and secure, letting you focus on development rather than configuration management.

Azure App Configuration Key-Value Pairs

Overview:

- Azure App Configuration stores data in key-value pairs.

Keys:

- **Purpose:** Names for key-value pairs, used for storage and retrieval.
- **Structure:** Can be organized hierarchically with delimiters like `/` or `:`.
    - **Example:**

```
AppName:Service1:ApiEndpoint
AppName:Service2:ApiEndpoint
```

- **Case Sensitivity:** Keys are case-sensitive (e.g., `app1` ≠ `App1`).
- **Character Restrictions:**
    - Avoid using `*`, `,`, `\`. Escape with `\{Reserved Character}` if needed.
    - Combined size limit of 10,000 characters for key, value, and attributes.

Designing Key Namespaces:

- **Flat vs. Hierarchical:**
    - **Flat:** Simple but less organized.
    - **Hierarchical:**
        - Easier to read, manage, and use.
        - Allows for logical grouping with delimiters.

Labels:

- **Function:** Differentiate key-values with the same key but different contexts.
- **Default:** No label (`\0` or `%00` for no label).
- **Use Cases:**
    - Environment differentiation:

```
Key = AppName:DbEndpoint & Label = Test
Key = AppName:DbEndpoint & Label = Staging
Key = AppName:DbEndpoint & Label = Production
```

This structure helps in understanding and utilizing Azure App Configuration effectively for managing application settings. Remember, while hierarchical naming makes things more manageable, the simplicity of flat naming can be beneficial for smaller applications or when the complexity of hierarchy isn't warranted.

Azure App Configuration Key-Value Management

Versioning Key Values:

- Azure App Configuration does not automatically version key-values upon modification.

- **Versioning Strategy:** Use labels for different versions:
  - **Example:** Implement version control or commit IDs as labels.

```
Key = AppSetting & Label = v1.0.0
Key = AppSetting & Label = commit-abc123
```

Querying Key Values:

- **Uniqueness:** Each key-value is identified by its key and a label (`\0` for no label).
- **Query Mechanism:** Use pattern matching to retrieve key-values:
  - The store returns key-values, their values, and attributes that match the pattern.

Value Storage:

- **Character Set:** Values are unicode strings, allowing all unicode characters.
- **Content Type:**
  - An optional content type can be set for each value to indicate how the value should be processed:

```
Key = "MimeTypeExample"
Value = "This is an example value."
Content-Type = "text/plain"
```

Security:

- **Encryption:** All data (keys and values) is encrypted both at rest and in transit.
- **Usage Note:**
  - App Configuration should not be used for storing application secrets. Instead, use Azure Key Vault for secret management.

Summary:

- Use labels for versioning or environment differentiation.
- Query with patterns to manage and retrieve configurations.
- Values are flexible with unicode support, include metadata like content type for proper application handling.
- Ensure security through Azure's encryption but remember, it's not a secret management tool.

Feature Management in Azure

Overview: Feature management allows for separating feature release from code deployment, enabling dynamic control over feature availability.

Key Concepts:

- **Feature Flag:**
  - A binary state (`on`/`off`) variable controlling whether a code block should execute.

- **Feature Manager:**
  - Manages all feature flags within an application, offering functionalities like caching and state updates.
- **Filter:**
  - Defines criteria for when a feature flag should be active (e.g., user group, device type, geographic location, time).

Implementation Components:

1. **Application:** Uses feature flags within its codebase.
2. **Feature Flag Repository:** External storage for flag states, allowing changes without code redeployment.

Feature Flag Usage in Code:

- **Basic Pattern:**

```
if (featureFlag) {
    // Run this code block when flag is true
}
```

- **Static Flag Setting:**

```
bool featureFlag = true;
```

- **Dynamic Flag Evaluation:**

```
bool featureFlag = isBetaUser(); // Evaluates based on some condition
```

- **Conditional Branching:**

```
if (featureFlag) {
    // Code for when feature is enabled
} else {
    // Code for when feature is disabled
}
```

Summary:

- Feature flags facilitate agile feature control post-deployment.
- They allow for safe testing and gradual rollouts of new features or changes.
- Use a feature manager for comprehensive management and a repository for external state control.

Azure Feature Flag Management

Feature Flag Declaration:

- **Parts of a Feature Flag:**
  1. **Name:** Identifies the feature flag.
  2. **Filters:** A list determining when the feature should be active.
- **Filter Logic:**
  - Filters are evaluated in order.
  - The feature flag is enabled if any filter returns true.
  - If no filter enables the feature, the flag remains off.

Configuring Feature Flags in JSON:

- **Example Configuration:**

```
"FeatureManagement": {
    "FeatureA": true, // Feature always on
    "FeatureB": false, // Feature always off
    "FeatureC": {
        "EnabledFor": [{
            "Name": "Percentage",
            "Parameters": {
                "Value": 50
            }
        }]
    }
}
```

  - **Explanation:**
    - `FeatureA` is directly set to true, meaning it's always on.
    - `FeatureB` is set to false, meaning it's always off unless toggled in the repository.
    - `FeatureC` uses a percentage filter, enabling the feature for 50% of users.

Feature Flag Repository:

- **Externalization:**
  - Feature flags are stored outside the application for dynamic control.
  - Allows changes without redeployment.
- **Azure App Configuration:**
  - Acts as a centralized repository for managing feature flags.
  - Supports defining and manipulating feature flag states.
- **Integration with Applications:**

- Use App Configuration libraries for your programming framework to interact with these flags.

Summary:

- Feature flags in Azure enable dynamic feature control through JSON configuration.
- They can be managed via Azure App Configuration, which provides a robust system for flag management without code changes.

Securing Azure App Configuration Data

Encryption with Customer-Managed Keys:

- **Default Encryption:** Azure App Configuration uses Microsoft-provided 256-bit AES encryption for data at rest.
- **Customer-Managed Keys:**
  - When enabled:
    - An App Configuration instance uses a managed identity to authenticate with Microsoft Entra ID.
    - The identity accesses Azure Key Vault to wrap the instance's encryption key.
    - The wrapped key is stored, and the unwrapped key is cached for an hour, refreshing hourly to ensure availability.

Steps for Enabling Customer-Managed Keys:

1. **Components Needed:**
   - **Azure App Configuration instance (Standard tier)**
   - **Azure Key Vault** with:
     - Soft-delete enabled
     - Purge protection enabled
   - **RSA or RSA-HSM key** in Key Vault with:
     - Not expired
     - Enabled
     - Capabilities: Wrap Key and Unwrap Key
2. **Configuration Steps:**
   - Assign a managed identity to the App Configuration instance.
   - Grant this identity `GET`, `WRAP`, and `UNWRAP` permissions in Key Vault's access policy.

Private Endpoints:

- **Purpose:** To enable secure access over a private link from within a virtual network.
- **Benefits:**
  - **Security:** Firewall can block public endpoint access.
  - **Network Isolation:** Ensures data stays within the virtual network.
  - **On-Premises Access:** Connects securely from on-premises via VPN or ExpressRoute with private peering.

Summary:

- Use customer-managed keys for greater control over data encryption.
- Implement private endpoints to enhance security and network isolation for your configuration data.

Managed Identities in Azure

Overview:

- Managed identities from Microsoft Entra ID enable Azure App Configuration to access other Azure resources without managing secrets.

Types of Managed Identities:

- **System-Assigned Identity:**
  - Directly associated with the resource (like App Configuration store).
  - Deleted when the resource is deleted.
  - Only one per resource.
- **User-Assigned Identity:**
  - Standalone Azure resource.
  - Can be assigned to multiple resources.
  - Persists independently of resource lifecycle.

Setting Up System-Assigned Identity:

- Use the Azure CLI command `az appconfig identity assign` to create a system-assigned identity for an App Configuration store:

```
az appconfig identity assign \
    --name myTestAppConfigStore \
    --resource-group myResourceGroup
```

Setting Up User-Assigned Identity:

1. **Create the Identity:**
   - First, create the user-assigned identity in the same resource group:

```
az identity create --resource-group myResourceGroup --name
myUserAssignedIdentity
```

2. **Assign Identity to App Configuration:**
   - Then, assign this identity to your App Configuration store:

```
az appconfig identity assign --name myTestAppConfigStore \
    --resource-group myResourceGroup \
```

```
    --identities /subscriptions/[subscription
id]/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/
userAssignedIdentities/myUserAssignedIdentity
```

- Replace `[subscription id]` with your actual subscription ID.

Summary:

- Managed identities simplify security by removing the need for secret management.
- System-assigned identities are lifecycle-bound to the resource, while user-assigned identities offer more flexibility for reuse across multiple resources.
- Use Azure CLI commands to manage these identities with ease.

AZ-204: Implement API Management

Course Duration: 42 minutes

Learning Path: Intermediate Developer

Completion Status: 0 of 1 modules completed

Overview

- **Objective**: Learn to use Azure API Management for enhancing, securing, and managing APIs effectively.

Key Topics to Cover

1. **Introduction to API Management**:
   - Understand what API Management is and why it's crucial for developers and businesses.
2. **Functionality of API Management**:
   - **API Gateway**: Acts as the front door for all API calls.
   - **Transformations**: Modify API requests and responses to meet specific needs.
   - **Security**: Implement policies for authentication, rate limiting, and more.
3. **API Transformation**:
   - Learn how to alter API structures, protocols, or contents to fit different client needs or standards.
4. **Securing APIs**:
   - **Authentication**: OAuth 2.0, Azure AD, certificates.
   - **Authorization**: Control access with subscription keys and Azure RBAC.
   - **Rate Limiting**: Manage API usage to prevent abuse.
5. **Creating and Managing Backend APIs**:
   - **Mocking**: Use mock responses for development or testing before the actual backend is ready.
   - **API Versioning**: Manage different versions of your API without breaking existing clients.

6. **Developer Portal**:
    - How to use the API Management developer portal for API discovery, testing, and documentation.

Prerequisites

- **Experience**:
    - At least one year of experience in developing scalable solutions through all phases of software development.
- **Knowledge**:
    - Basic understanding of Azure, cloud concepts, Azure services, and familiarity with the Azure portal.
- **Recommended**:
    - Complete AZ-900: Azure Fundamentals if you're new to Azure or cloud computing.

Note:

- API Management in Azure is like a sophisticated concierge service for your APIs, ensuring they are presented, secured, and managed in the best way possible to both developers and consumers.

Remember, mastering API Management is key to making your APIs not only functional but also scalable, secure, and user-friendly for external developers and internal teams alike.

Introduction to Azure API Management

Overview

- **Function**: Azure API Management (APIM) facilitates the publication, management, and analysis of APIs, enabling organizations to expose their data and services to developers.

Learning Outcomes

After completing this module, you'll be able to:

1. **Understand Components of API Management**:
    - **Gateway**: Receives API calls and routes them to the backend.
    - **Publisher Portal**: For managing your APIs, products, and developers.
    - **Developer Portal**: A place for developers to discover, learn about, and test APIs.
    - **Administration**: Handles configuration and policy management.
2. **API Gateway Functions**:
    - **Routing**: Directs requests to the correct backend service.
    - **Caching**: Improves performance by caching responses.
    - **Transformation**: Modifies request/response content or format.

    - **Analytics**: Monitors API usage for insights and billing.

3. **Securing APIs**:

    - **Subscriptions**: Developers subscribe to products, which groups APIs, ensuring controlled access.
    - **Certificates**: Use for mutual TLS authentication to secure communication.

4. **Creating a Backend API**:

    - **Mocking**: Set up mock responses during development or testing phases.
    - **API Definition**: Use OpenAPI (Swagger) or other specifications to define your API structure.
    - **Implementation**: Link to or proxy actual backend services.

Note:

- API Management acts like a sophisticated gatekeeper for your APIs, ensuring they're accessible, secure, and performant while providing insights into their usage.

Remember, API Management is like being the host of an exclusive club where your APIs are the VIPs; you control who gets in, how they interact, and ensure everyone has a good time while keeping the party secure and efficient.

Discover the API Management Service

Overview

- **Functionality**: Azure API Management (APIM) offers tools for developer engagement, analytics, security, and API protection.

Key Components of Azure API Management

1. API Gateway

- **Role**: Acts as the entry point for all API calls.
    - **Functions**:
        - **Routing**: Directs requests to backend services.
        - **Authentication**: Verifies keys and credentials.
        - **Policy Enforcement**: Applies usage quotas, rate limits.
        - **Transformation**: Modifies requests/responses based on policies.
        - **Caching**: Reduces backend load by caching responses.
        - **Monitoring**: Generates logs, metrics, and traces.

2. Management Plane

- **Purpose**: Administrative interface for setting up and managing API programs.
    - **Tasks**:
        - **Service Configuration**: Provision and configure APIM settings.
        - **API Definition**: Define or import API schemas.
        - **Product Packaging**: Group APIs into products.
        - **Policy Management**: Define policies for APIs.

        - **Analytics**: Gain insights from usage data.
        - **User Management**: Control access and manage user accounts.

3. Developer Portal

- **Description**: A customizable website for API documentation and interaction.
    - **Features**:
        - **Documentation**: Provides detailed information about APIs.
        - **Testing**: Interactive console for API calls.
        - **Account Management**: Developer registration and subscription.
        - **Usage Analytics**: Developers can view their own usage stats.
        - **API Definitions**: Ability to download or view API specs.
        - **Key Management**: Manage subscription keys.

Note:

- Different tiers of Azure API Management offer varying levels of capacity and features, allowing scalability from small projects to enterprise-level API programs.

Remember, Azure API Management is like your API's personal butler, concierge, and security detail all rolled into one, ensuring your APIs are presented in the best light, protected, and utilized effectively by developers.

Key Concepts in Azure API Management

Products

- **Role**: Products group one or more APIs for presentation to developers.
    - **Attributes**:
        - **Title**: Name of the product.
        - **Description**: Detailed explanation for developers.
        - **Terms of Use**: Usage conditions.
    - **Visibility**:
        - **Open**: No subscription needed.
        - **Protected**: Requires subscription, which might need approval.

Groups

- **Purpose**: Control product visibility to different user categories.
    - **System Groups**:
        - **Administrators**: Manage API Management, create APIs, operations, and products.
        - **Developers**: Users who develop applications using the APIs.
        - **Guests**: Unauthenticated users with limited access.
    - **Custom Groups**: Admins can create additional groups.
    - **External Groups**: Can leverage Microsoft Entra (Azure AD) groups.

Developers

- **Definition**: User accounts within the API Management service.

    - **Access**: Developers can:
        - Be created or invited by admins.
        - Sign up via the Developer Portal.
        - Belong to multiple groups, gaining access to products associated with those groups.

Policies

- **Function**: Define how requests and responses are handled.
    - **Statements**: Examples include:
        - **Format Conversion**: XML to JSON or vice versa.
        - **Rate Limiting**: Throttle calls to prevent abuse.
    - **Policy Expressions**: Used for dynamic values or control flow within policies.
    - **Scopes**: Policies can be applied at:
        - **Global**: Affects all APIs.
        - **Product**: Specific to a product.
        - **API**: Applies to a single API.
        - **Operation**: Impacts a specific operation within an API.

Note:

- Policies are crucial for managing API behavior, ensuring security, performance, and compliance with business rules.

Remember, in Azure API Management:

- **Products** are like your API's storefront, where you decide how and to whom they are sold.
- **Groups** are like different levels of membership to your store, controlling who gets to browse or buy.
- **Developers** are your shoppers, navigating through your product offerings.
- **Policies** are the rules of engagement for how transactions (API calls) are handled in your store.

Explore API Gateways

Role of an API Gateway

- **Position**: Acts as an intermediary between clients and backend services.
- **Functions**:
    - **Reverse Proxy**: Routes client requests to the correct service.
    - **Cross-Cutting Concerns**: Handles authentication, SSL termination, rate limiting, etc.

Problems Without an API Gateway

1. **Complexity in Client Code**: Clients need to manage multiple endpoints and handle service failures.
2. **Coupling**: Clients become tightly coupled to backend service structure, complicating maintenance and refactoring.
3. **Multi-Service Operations**: Operations often require multiple service calls.
4. **Security and Protocol Management**: Each service must manage security and protocol exposure, increasing attack surface and limiting protocol flexibility.

Benefits of Using an API Gateway

- **Decoupling**: Separates clients from directly interacting with backend services.
- **Simplified Client Interaction**: Clients deal with one endpoint, making updates and maintenance easier.
- **Centralized Policy Management**: Apply policies like authentication or rate limiting once at the gateway level.

Types of API Gateways in Azure API Management

Managed Gateway

- **Deployment**: Default component for every API Management instance in Azure.
- **Traffic**: All API traffic goes through Azure, regardless of backend location.

Self-Hosted Gateway

- **Deployment**: Containerized version for deployment outside Azure.
- **Use Case**: Ideal for:
    - **Hybrid Environments**: Where APIs are hosted both on-premises and in the cloud.
    - **Multicloud Scenarios**: Manages APIs across different cloud platforms from a single Azure API Management service.

Note:

- The API Gateway in Azure API Management can be viewed as a sophisticated traffic controller for your APIs, ensuring smooth, secure, and managed access to your services.

Remember, an API gateway is like the front desk of a hotel for your APIs; it directs clients to the right room (service), checks their credentials, and manages the flow of guests (requests) to ensure a harmonious stay (API interaction).

Explore API Management Policies

Completion: Completed

Experience Points: 100 XP

Duration: 3 minutes

Role of Policies

- **Function**: Policies are used to modify the behavior of your API through configuration, not code changes.
- **Execution**: Policies run in the gateway between the API consumer and the backend.

Policy Configuration

- **Format**: XML document with sections for different stages of request handling.
- **Sections**:
    - **inbound**: Policies applied to incoming requests.

- **backend**: Policies executed before forwarding the request to the backend.
- **outbound**: Policies applied to the response before sending it back.
- **on-error**: Handles error conditions, allowing for custom error responses.

Sample Policy Configuration

```
<policies>
  <inbound>
    <!-- statements to be applied to the request go here -->
  </inbound>
  <backend>
    <!-- statements before forwarding to backend service -->
  </backend>
  <outbound>
    <!-- statements applied to the response -->
  </outbound>
  <on-error>
    <!-- error handling statements -->
  </on-error>
</policies>
```

- **Error Handling**: If an error occurs, processing skips to the `on-error` section, where you can manage errors using `context.LastError`.

Policy Expressions

- **Usage**: Policy expressions can be used as values in policies for dynamic behavior.
- **Types**:
  - **Single Statement**: `@{ }`
  - **Multi-Statement**: `@{ }`
- **Context**: Expressions can use `context` for accessing request/response data and various .NET types.

Example of Policy with Expressions

```
<policies>
    <inbound>
        <base />
        <set-header name="x-request-context-data" exists-action="override">
            <value>@(context.User.Id)</value>
            <value>@(context.Deployment.Region)</value>
        </set-header>
    </inbound>
</policies>
```

- **Explanation**: This policy adds a custom header to the request with dynamic values like the user's ID and the region where the API Gateway is deployed.

Note:

- Policies and expressions offer a powerful way to manage and manipulate API traffic, allowing for sophisticated control without backend code changes.

Remember, policies in API Management are like the rules of engagement for your APIs; they govern how requests are processed, how responses are shaped, and how errors are handled, all without touching the backend services.

Apply Policies at Different Scopes and Filter Response Content

Scope of Policies

- **Multiple Policy Application**: Policies can be defined at various scopes like global, product, API, or operation level. When an API is called, all applicable policies are executed.

Deterministic Policy Ordering

- **Using `<base />`**: This element ensures that broader scope policies are executed before more specific ones.

```
<policies>
    <inbound>
        <cross-domain />
        <base />
        <find-and-replace from="xyz" to="abc" />
    </inbound>
</policies>
```

- **Execution Order**:
    1. `<cross-domain />` executes first.
    2. `<base />` would run any global or parent policies.
    3. `<find-and-replace>` runs last.

Filtering Response Content

- **Policy Application Based on Product**: This example shows how to modify the response based on the product being used.

```
<policies>
  <inbound>
    <base />
  </inbound>
  <backend>
    <base />
  </backend>
  <outbound>
    <base />
```

```
    <choose>
      <when condition="@(context.Response.StatusCode == 200 &&
context.Product.Name.Equals("Starter"))">
        <set-body>
          @{
            var response = context.Response.Body.As<JObject>();
            foreach (var key in new [] {"minutely", "hourly", "daily", "flags"}) {
              response.Property(key).Remove();
            }
            return response.ToString();
          }
        </set-body>
      </when>
    </choose>
  </outbound>
  <on-error>
    <base />
  </on-error>
</policies>
```

- **Explanation**:
  - **Condition**: Checks if the response status is 200 and the product name is "Starter".
  - **Action**: Removes specified properties (`minutely`, `hourly`, `daily`, `flags`) from the JSON response for users subscribed to the "Starter" product.

Note:

- **Policy Design**: Using `<base />` allows for a clean way to combine policies at different scopes, ensuring they are applied in a predictable order.
- **Response Filtering**: Customizes the response content dynamically based on the consumer's subscription, enhancing API usage control without backend changes.

Remember, policies act like configurable middleware in your API pipeline, where you can decide not only how requests are processed but also how responses are shaped, tailored to different users or conditions.

Create Advanced Policies in Azure API Management

Overview of Advanced Policies

Control Flow Policy - `<choose>`

- **Purpose**: Allows conditional execution of policy statements.
- **Structure**: Similar to if-then-else or switch in programming.

```
<choose>
    <when condition="Boolean expression | Boolean constant">
        <!-- policy statements if condition is true -->
```

```
    </when>
    <when condition="Boolean expression | Boolean constant">
        <!-- policy statements if the above condition is true -->
    </when>
    <otherwise>
        <!-- policy statements if no conditions are true -->
    </otherwise>
</choose>
```

- **Execution**: Evaluates conditions sequentially. Only the first true `<when>` executes. The `<otherwise>` runs if no conditions are met.

Forward Request Policy

- **Function**: Forwards the API request to the backend service.

```
<forward-request timeout="time in seconds" follow-redirects="true | false"/>
```

- **Behavior**:
  - If removed, the request won't be forwarded to the backend.
  - Outbound policies are evaluated upon successful inbound policy execution.

Limit Concurrency Policy

- **Purpose**: Throttles the number of concurrent executions of enclosed policies.

```
<limit-concurrency key="expression" max-count="number">
    <!-- nested policy statements -->
</limit-concurrency>
```

- **Effect**: When concurrency exceeds `max-count`, new requests receive a 429 status code.

Note:

- **Policy Expressions**: Can be used within conditions or as part of the policy configuration for dynamic behavior.
- **Advanced Policies**: These policies give you fine control over how requests are processed, allowing for sophisticated API management scenarios like traffic shaping, conditional routing, and performance optimization.

Remember, these advanced policies are the fine-tuning knobs of your API Management service, enabling you to tailor the behavior of your APIs with precision based on conditions, concurrency, and backend interactions.

Additional Advanced Policies in Azure API Management

Log to Event Hub

- **Function**: Logs messages to an Azure Event Hub for analysis.

```
<log-to-eventhub logger-id="id of the logger entity" partition-id="index of the
partition where messages are sent" partition-key="value used for partition
assignment">
  Expression returning a string to be logged
</log-to-eventhub>
```

- **Use Case**: Useful for logging request/response context for later analysis.

Mock Response

- **Purpose**: Used to simulate API responses, useful during development or testing.

```
<mock-response status-code="code" content-type="media type"/>
```

- **Behavior**:
  - Prefers response examples if available.
  - Falls back to schema-generated responses.
  - Returns empty responses if no examples or schemas exist.

Retry

- **Function**: Repeats enclosed policy execution based on conditions.

```
<retry
    condition="boolean expression or literal"
    count="number of retry attempts"
    interval="retry interval in seconds"
    max-interval="maximum retry interval in seconds"
    delta="retry interval delta in seconds"
    first-fast-retry="boolean expression or literal">
    <!-- Child policies -->
</retry>
```

- **Parameters**:
  - **condition**: When true, retry attempts continue.
  - **count**: Maximum number of retries.
  - **interval**: Time between retries.
  - **max-interval**: Maximum time between retries.
  - **delta**: Increment for retry intervals.
  - **first-fast-retry**: Option for an immediate first retry.

Return Response

- **Purpose**: Aborts further policy execution and returns a response to the caller.

```
<return-response response-variable-name="existing context variable">
  <set-header/>
  <set-body/>
  <set-status/>
</return-response>
```

- **Behavior**:
  - Can return a default response or customize it with policy statements.
  - Allows setting of headers, body, and status before response.

Note:

- These policies enhance the robustness and flexibility of your API management, allowing for better error handling, logging, testing, and immediate response control.

Remember, using these policies is like having a set of control levers for your API's behavior, ensuring you can manage logs, simulate responses, handle retries, and directly control what gets back to your clients.

Secure APIs by Using Subscriptions

Subscription Keys for API Security

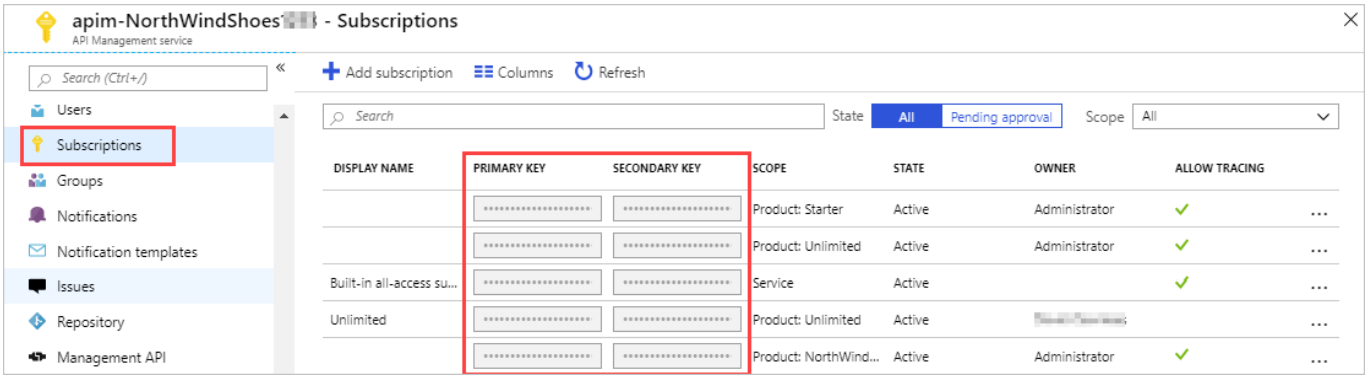
- **Purpose**: Subscription keys are used to secure access to APIs published through Azure API Management.
- **Functionality**:
  - **Mechanism**: Clients must include a valid subscription key in their API requests.
  - **Effect**: Requests without a key are rejected by the API Gateway.

Subscriptions

- **Definition**: A container for subscription keys, enabling developers to consume APIs.
- **Acquisition**:
  - Developers can request subscriptions, which might require approval.
  - API publishers can create subscriptions directly for consumers.

Subscription Scopes

- **All APIs**: Subscription applies to all APIs accessible via the gateway.
- **Single API**: Key is valid for one specific API and all its operations.
- **Product**:
  - A collection of APIs with shared access rules, quotas, and terms.
  - An API can belong to multiple products.



Key Management

- **Key Types**:
  - **Primary and Secondary Keys**: Each subscription comes with two keys.
  - **Regeneration**: Keys can be regenerated if security is compromised or for rotation purposes.
- **Key Usage**:
  - **Inclusion**: Keys must be included in API calls, either in headers or as query parameters.
  - **Redundancy**: Having two keys allows for seamless key rotation without service interruption.

Developer Interaction

- **Workflow**:
  1. Developer requests access to a product or API.
  2. If approved, the publisher provides the subscription key securely.

Note:

- While subscription keys are a common method, API Management also supports OAuth2.0, client certificates, and IP allow listing for added security.

Remember, subscription keys in API Management act like VIP passes for your APIs; they grant access to the show (API access) but can be revoked or changed if someone's caught sneaking in without an invitation.

Call an API with the Subscription Key

Including Subscription Keys in API Calls

- **Requirement**: A valid subscription key must be included with each API call to access protected endpoints.

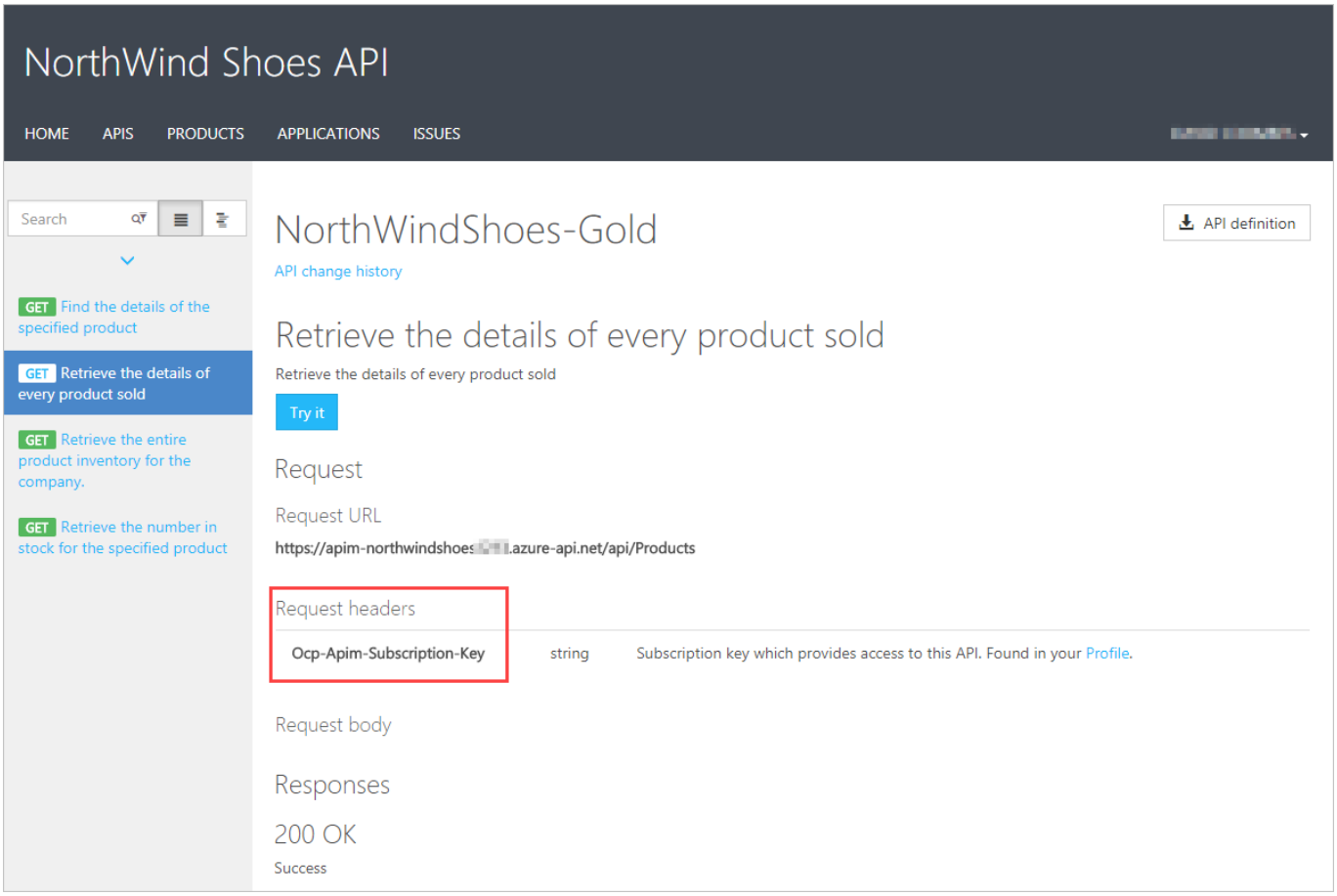
Ways to Include Keys

1. **HTTP Headers**:
   - **Default Header Name**: `Ocp-Apim-Subscription-Key`
   - **Using `curl`**:

```
curl --header "Ocp-Apim-Subscription-Key: <key string>" https://<apim
gateway>.azure-api.net/api/path
```

2. **Query String**:
    - **Default Query Parameter**: `subscription-key`
    - **Using `curl`**:

```
curl https://<apim gateway>.azure-api.net/api/path?subscription-key=
<key string>
```



Testing API Calls

- **Developer Portal**: Offers an interactive environment to test API calls with the subscription key included automatically.
- **Command Line Tools**: `curl` can be used for testing from the command line with the key provided either in the header or query string.

Error Handling

- **Missing Key**: Without the subscription key, API Management will return a `401 Access Denied` HTTP status code.

Note:

- The choice between headers and query strings for passing the subscription key can depend on security preferences or API design considerations. Headers are generally more secure as they aren't logged in plain text by many intermediate systems.

Remember, the subscription key is your API's bouncer; without it, there's no entry to the exclusive club of your API's resources. Make sure you've got your ticket (key) in hand before you try to get in.

Secure APIs by Using Certificates

TLS Mutual Authentication with Certificates

- **Purpose**: Use certificates to ensure mutual authentication between the client and the API Management Gateway.

Gateway Configuration for Certificate Validation

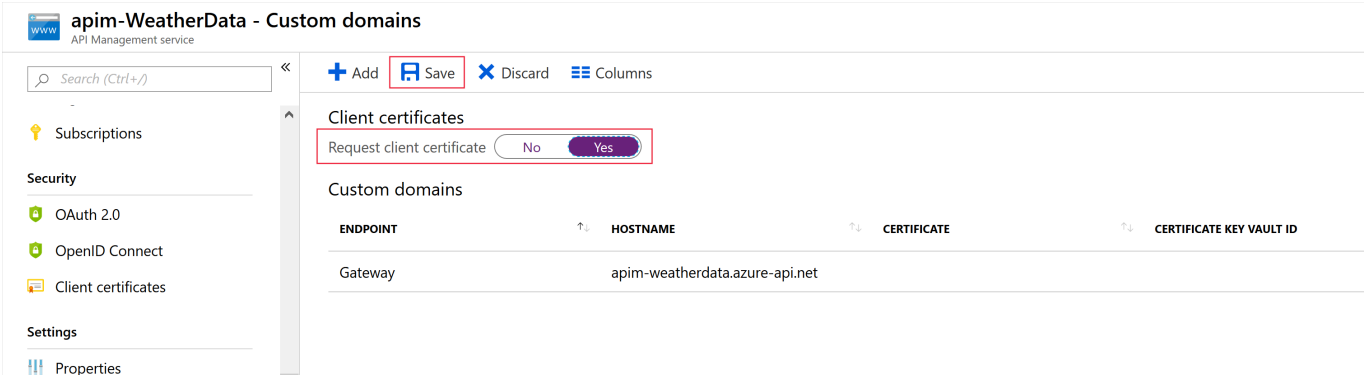
- **Properties for Certificate Inspection**:
    - **Certificate Authority (CA)**: Only accept certificates from specified CAs.
    - **Thumbprint**: Validate certificates by matching thumbprints.
    - **Subject**: Check for specific certificate subjects.
    - **Expiration Date**: Reject expired certificates.
- **Policy Configuration**: These checks can be combined in the gateway's inbound policy to enforce security requirements.

Certificate Verification

- **CA Trust**: Certificates are trusted if issued by a known CA.
    - **Automation**: Trusted CAs can be configured in the Azure portal.
- **Self-Signed Certificates**:
    - Require manual verification to ensure authenticity, often done through direct delivery.

Consumption Tier Specifics

- **Serverless Design**: Suited for serverless architectures like Azure Functions.
- **Client Certificates**:
    - **Enablement**: Must be explicitly enabled in the Consumption tier via the Custom domains page.
    - **Availability**: This step is not required for other service tiers.



Configuration Steps

1. **Enable Client Certificates**:
    - For Consumption tier, go to the Custom domains page in Azure portal.
2. **Set Up Certificate Policies**:
    - Use inbound policies to check for the desired certificate properties.

Note:

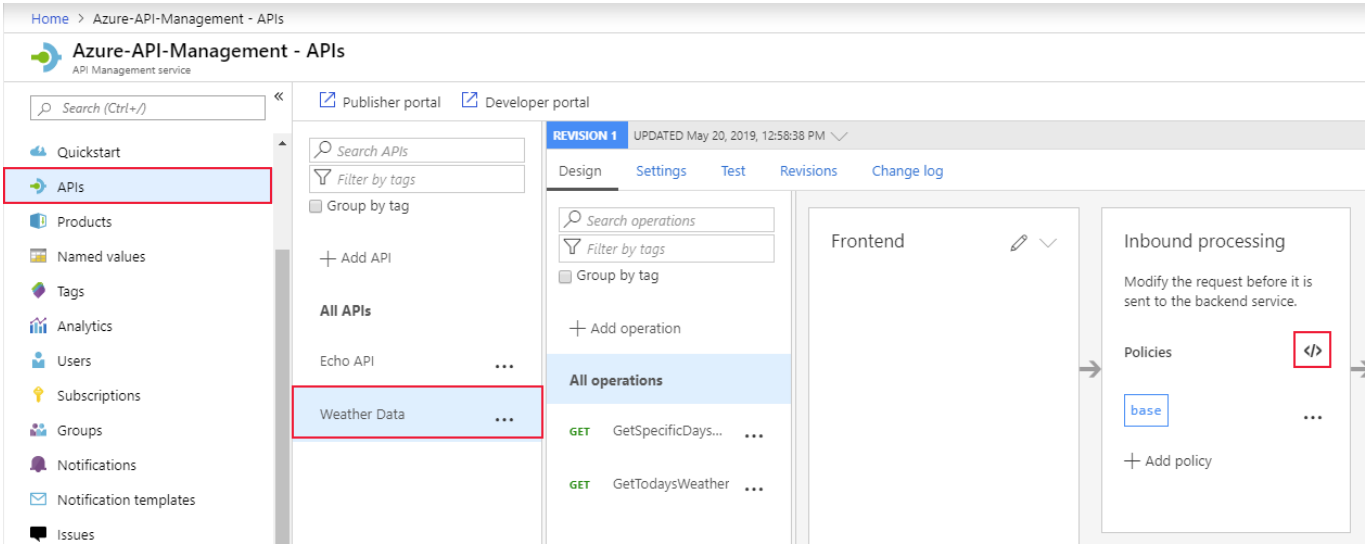
- **Security Benefit**: Certificate-based authentication adds an additional layer of security, ensuring that only clients with valid, trusted certificates can access your APIs.
- **Flexibility**: Policies can be tailored to accept certificates based on different criteria, enhancing your API's security posture.

Remember, certificates in API Management are like exclusive VIP passes; they ensure only the right clients with the correct credentials can enter the secure space of your API, keeping imposters out.

Certificate Authorization Policies

Overview

- **Purpose**: Define policies to validate client certificates in the inbound processing policy file.



Policy Statements

Check Thumbprint of a Client Certificate

- **Function**: Ensures the certificate has not been altered since issuance.

```
<choose>
    <when condition="@(context.Request.Certificate == null ||
context.Request.Certificate.Thumbprint != "desired-thumbprint")" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate" />
        </return-response>
    </when>
</choose>
```

Check Thumbprint Against Uploaded Certificates

- **Scenario**: Allows for multiple client certificates, each with a unique thumbprint.

```
<choose>
    <when condition="@(context.Request.Certificate == null ||
!context.Request.Certificate.Verify()  || !context.Deployment.Certificates.Any(c
=> c.Value.Thumbprint == context.Request.Certificate.Thumbprint))" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate" />
        </return-response>
    </when>
</choose>
```

- **Process**:
    1. Obtain and upload client certificates to the API Management resource.
    2. Policy checks if the client's certificate is among those uploaded.

Check Issuer and Subject of a Client Certificate

- **Validation**: Ensures the certificate is from a trusted issuer and matches the expected subject.

```
<choose>
    <when condition="@(context.Request.Certificate == null ||
context.Request.Certificate.Issuer != "trusted-issuer" ||
context.Request.Certificate.SubjectName.Name != "expected-subject-name")" >
        <return-response>
            <set-status code="403" reason="Invalid client certificate" />
        </return-response>
    </when>
</choose>
```

Note:

- These policies are crucial for enforcing security at the gateway level, ensuring only authenticated clients with proper certificates can access the API.
- The `<choose>` policy allows conditional logic to either grant or deny access based on certificate validation.

Remember, like checking a guest list at an exclusive event, these certificate policies ensure only guests with the right credentials (certificates) get access to your API's resources, keeping the party secure and exclusive.

AZ-204: Develop event-based solutions

Services Covered:

- **Azure Event Grid**
- **Azure Event Hubs**

Objective

- Learn to build applications using event-based architectures.
- Integrate Azure Event Grid and Azure Event Hubs into your solutions.

Prerequisites

- **Experience**: At least one year of experience in developing scalable solutions through all phases of software development.
- **Azure Knowledge**: Basic understanding of Azure, cloud concepts, services, and navigation within the Azure portal.
- **Recommendation**: If new to Azure or cloud computing, complete the AZ-900: Azure Fundamentals course first.

Explore Azure Event Grid

Topics Covered:

- **Integration**: Learn how to integrate Azure Event Grid into your solutions.
- **Access Control**: Implementation of access control for events.
- **Routing Custom Events**: Techniques to route custom events to web endpoints using Azure CLI.

Introduction to Azure Event Grid

Key Points:

- **Integration**: Azure Event Grid is highly integrated with Azure services and supports third-party integrations.
- **Efficiency**: Reduces costs by avoiding constant polling, facilitating event-driven architectures.
- **Reliability**: Efficiently routes events from Azure and non-Azure sources to subscribers.

Learning Outcomes:

- Understand the operational mechanics of Event Grid and its connectivity with various services and handlers.
- Comprehend the event delivery mechanism and error handling in Event Grid.
- Implement **authentication and authorization** for Event Grid.
- Use **Azure CLI** to route custom events to web endpoints.

Explore Azure Event Grid

Overview of Azure Event Grid:

- **Service Type**: Highly scalable, managed Pub/Sub messaging service.
- **Protocols**: Supports HTTP and MQTT (v3.1.1 & v5.0).

Capabilities:

- **Data Pipelines**: Facilitates building data pipelines with IoT device data.
- **Application Integration**: Helps in integrating various applications.
- **Serverless Architectures**: Enables building event-driven serverless architectures.

Features:

- **Event Publishing**: Publishers can announce system state changes through events.
- **Event Subscription**: Subscribers can receive events via push or pull delivery methods.
- **IoT Support**: Event Grid supports IoT solutions through MQTT protocols.
- **Interoperability**: Adopts CloudEvents 1.0 specification for cross-system event handling.

Concepts in Azure Event Grid

Publishers

- **Definition**: An application or service that sends events to Event Grid.
- **Types**:
    - **Direct Publishers**: Azure services or your own applications publish events to announce changes or occurrences.
    - **Partners**: External entities that publish events to Azure Event Grid. They can also subscribe to events through the Partner Events feature.

Events and CloudEvents

- **Event**: Describes an occurrence in a system, containing:
    - Common information (source, time, unique identifier).
    - Specific details relevant to the event type.
- **CloudEvents 1.0**:
    - Event Grid uses the CloudEvents standard for event formatting.

- **Format Example**:

```
{
    "specversion" : "1.0",
    "type" : "com.yourcompany.order.created",
    "source" : "https://yourcompany.com/orders/",
    "subject" : "O-28964",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "application/json",
    "data" : {
       "orderId" : "O-28964",
       "URL" : "https://com.yourcompany/orders/O-28964"
    }
}
```

- **Event Size Limitations**:
  - **Maximum Size**: 1 MB
  - **Charging**: Events over 64 KB are billed in 64-KB increments.

These concepts are fundamental to understanding how Azure Event Grid operates and how you can interact with it in your applications.

Event Sources

- **Definition**: Where the event originates.
- **Examples**:
  - Azure Storage for blob events.
  - IoT Hub for device events.
  - Custom application-defined events.

Topics

- **Purpose**: Holds events published to Event Grid.
- **Types**:
  - **System Topics**: Provided by Azure services, not visible in your subscription but subscribable if you have resource access.
  - **Custom Topics**: For applications or third-party use, visible in your subscription.
  - **Partner Topics**: For receiving events from a partner system through Partner Events feature.

Event Subscriptions

- **Function**: Specifies interest in particular events from a topic.
- **Components**:
  - **Endpoint**: Where events are sent.
  - **Filtering**: By event type or subject pattern.
  - **Expiration**: Can set for temporary subscriptions.

Event Handlers

- **Definition**: Receives and processes events.
- **Types**:
  - Azure services or custom webhooks.
- **Delivery Assurance**:
  - HTTP webhook: Retried until a 200 OK response.
  - Azure Storage Queue: Retried until successfully processed.

Security

- **Subscription**: Requires permissions on the topic.
- **Event Delivery**:
  - For Azure services, managed identities with appropriate RBAC roles are used (e.g., Event Hubs Data Sender for Event Hubs).

This breakdown covers the essential components and security measures associated with Azure Event Grid, providing a clear understanding of how to manage events, subscriptions, and secure delivery.

Discover Event Schemas

Completion Status: Completed
Experience Points: 100 XP
Duration: 3 minutes

Event Schemas in Azure Event Grid

- **Schema Types**:
  - Event Grid event schema
  - Cloud event schema

Common Event Properties

All events share four required string properties:

- **topic**: The source of the event.
- **subject**: The subject of the event within the topic.
- **id**: A unique identifier for the event.
- **eventType**: The type of event that occurred.

Publisher-Specific Data

- **data**: Contains properties unique to the event source (e.g., Azure Storage or Event Hubs specifics).

Event Delivery

- **Array**: Events are sent to Event Grid in an array, with each array up to 1 MB in total size.
- **Event Size Limitations**:
  - Each event within the array must not exceed 1 MB.

    - **Error**: 413 Payload Too Large if limits are exceeded.
- **Charging**: Operations are charged in 64 KB increments. Events larger than 64 KB are charged as multiple events.

JSON Schema Example

Here's an example of the structure for an Event Grid event:

```
[
  {
    "topic": "string",
    "subject": "string",
    "id": "string",
    "eventType": "string",
    "eventTime": "string",
    "data":{
      "object-unique-to-each-publisher"
    },
    "dataVersion": "string",
    "metadataVersion": "string"
  }
]
```

- **Finding Schemas**: JSON schemas for Event Grid events and data payloads from Azure publishers can be found in the Event Schema store.

This format should assist in understanding the structure and limitations of event schemas in Azure Event Grid.

Event Properties in Azure Event Grid

Overview: All events share common top-level data properties:

Property	Type	Required	Description
topic	string	No	Full resource path to the event source. Provided by Event Grid if not included.
subject	string	Yes	Publisher-defined path to the event subject.
eventType	string	Yes	The type of event from the source.
eventTime	string	Yes	Time when the event is generated (UTC).
id	string	Yes	Unique identifier for the event.
data	object	No	Contains event-specific data, defined by the resource provider.
dataVersion	string	No	Schema version of the data object, defined by the publisher.
metadataVersion	string	No	Schema version of the event metadata, defined by Event Grid.

Notes:

- For custom topics, the event publisher decides the structure of the `data` object.
- When creating event subjects, make them informative for easy filtering and routing by subscribers.

Subject Structuring for Custom Topics:

- Use a path structure like `/A/B/C` for subjects.
    - Allows for broad (`/A`) or narrow (`/A/B`) filtering.

Cloud Events Schema

- Azure Event Grid supports CloudEvents v1.0 in JSON format over HTTP.
- CloudEvents aims to standardize event data for better interoperability.

Example of an Azure Blob Storage event in CloudEvents format:

```
{
    "specversion": "1.0",
    "type": "Microsoft.Storage.BlobCreated",
    "source": "/subscriptions/{subscription-id}/resourceGroups/{resource-
group}/providers/Microsoft.Storage/storageAccounts/{storage-account}",
    "id": "9aeb0fdf-c01e-0131-0922-9eb54906e209",
    "time": "2019-11-18T15:13:39.4589254Z",
    "subject": "blobServices/default/containers/{storage-container}/blobs/{new-
file}",
    "dataschema": "#",
    "data": {
        "api": "PutBlockList",
        "clientRequestId": "4c5dd7fb-2c48-4a27-bb30-5361b5de920a",
        "requestId": "9aeb0fdf-c01e-0131-0922-9eb549000000",
        "eTag": "0x8D76C39E4407333",
        "contentType": "image/png",
        "contentLength": 30699,
        "blobType": "BlockBlob",
        "url": "https://gridtesting.blob.core.windows.net/testcontainer/{new-
file}",
        "sequencer": "00000000000000000000000000099240000000000c41c18",
        "storageDiagnostics": {
            "batchId": "681fe319-3006-00a8-0022-9e7cde000000"
        }
    }
}
```

Header Differences:

- **CloudEvents Schema**: `"content-type":"application/cloudevents+json; charset=utf-8"`
- **Event Grid Schema**: `"content-type":"application/json; charset=utf-8"`

Usage:

- Event Grid supports both input and output of events in CloudEvents schema.

- Can be used for system events (like Blob Storage, IoT Hub) and custom events, with transformation capabilities.

This format provides a concise summary of the event properties and the implementation of CloudEvents schema in Azure Event Grid.

Explore Event Delivery Durability

Event Grid Delivery Durability

- **Guarantee**: Event Grid ensures at least one delivery attempt per event for each subscription.
- **Delivery Failure**: If there's no acknowledgment or if there's a failure, Event Grid employs a retry mechanism.
- **Default Behavior**: Delivers one event at a time, encapsulated in an array.

Note:

- Event Grid does not guarantee the order of event delivery.

Retry Schedule

- **On Error**: Event Grid decides to:
  - Retry the delivery,
  - Send to dead-letter storage,
  - or Drop the event based on the error type.
- **Non-Retry Conditions**:
  - **Azure Resources**: Errors 400, 413
  - **Webhook**: Errors 400, 413, 401

Important:

- Without Dead-Letter configuration, events encountering these errors are dropped.
- **Retry Mechanism**:
  - Waits for 30 seconds for response, then queues for retry.
  - Uses an exponential back-off policy.
  - Adds randomness to retry attempts and might skip retries if the endpoint is consistently unresponsive or overwhelmed.

Retry Policy Customization

- **Configurable Settings** for retry policy:
  - **Maximum number of attempts**: 1 to 30 (default is 30)
  - **Event time-to-live (TTL)**: 1 to 1440 minutes (default is 1440 minutes)

Example (Azure CLI):

```
az eventgrid event-subscription create \
  -g gridResourceGroup \
  --topic-name <topic_name> \
  --name <event_subscription_name> \
  --endpoint <endpoint_URL> \
  --max-delivery-attempts 18
```

- This sets the maximum delivery attempts to 18 for the event subscription.

These notes cover the key aspects of how Azure Event Grid handles event delivery failures and how you can customize the retry policy for your event subscriptions.

Output Batching

- **Purpose**: Improves HTTP performance in high-throughput scenarios.
- **Default**: Batching is off by default; can be enabled.
- **Settings**:
  - **Max events per batch**: 1 - 5,000.
  - **Preferred batch size in kilobytes**: Target for batch size, might be smaller if fewer events are available or if an event is larger than the preferred size.

Delayed Delivery

- **Scenario**: When an endpoint faces repeated delivery failures.
- **Action**: Event Grid delays subsequent retries and new deliveries to protect both the endpoint and the Event Grid system.

Dead-Letter Events

- **When**:
  - Event isn't delivered within TTL.
  - Delivery attempts exceed the limit.
- **Behavior**:
  - Events are moved to a storage account if configured.
  - **Default**: Dead-lettering is not enabled.
  - **Configuration**: Requires a storage account specified at the time of event subscription creation.
- **Immediate Dead-Lettering**: For HTTP response codes 400 or 413.
- **Delay**: 5-minute delay before moving events to dead-letter location.
- **If Dead-Letter Unavailable**: Events are dropped after 4 hours.

Custom Delivery Properties

- **Capability**: Set up to 10 custom HTTP headers for events.

- **Limitations**:
  - Each header value ≤ 4,096 bytes.
- **Supported Destinations**:
  - Webhooks
  - Azure Service Bus topics and queues
  - Azure Event Hubs
  - Relay Hybrid Connections

Example Storage Account Configuration: Before setting up dead-lettering:

```
- Ensure you have a storage account with a container.
- Provide the endpoint of this container when creating the event subscription.
```

These notes summarize the key points about batching, delayed delivery, dead-lettering, and custom headers for event delivery in Azure Event Grid.

Control Access to Events

Completion Status: Completed
Experience Points: 100 XP
Duration: 3 minutes

Azure Event Grid Access Control

Azure Event Grid uses Azure role-based access control (Azure RBAC) for managing access to various operations like listing, creating, or managing event subscriptions and keys.

Built-in Roles

- **Event Grid Subscription Reader**:
  - Permissions: Read event subscriptions.
- **Event Grid Subscription Contributor**:
  - Permissions: Manage event subscriptions.
- **Event Grid Contributor**:
  - Permissions: Create and manage Event Grid resources.
- **Event Grid Data Sender**:
  - Permissions: Send events to Event Grid topics.

Notes on Event Subscription Roles:

- These roles are essential for managing event subscriptions, particularly in event domains.
- They do not grant permissions to create or manage topics.

Permissions for Event Subscriptions

- **Non-WebHook Event Handlers**:
  - Requires write access to the resource (e.g., event hub or queue storage) to prevent unauthorized event sending.
- **Permission Requirement**:
  - **Microsoft.EventGrid/EventSubscriptions/Write** on the event source resource.

Topic Types and Permissions:

- **System Topics**:
  - **Permission Needed**: Write subscription at the scope of the event-publishing resource.
  - **Resource Format**:

```
/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/{resource-provider}/{resource-type}/{resource-name}
```

- **Custom Topics**:
  - **Permission Needed**: Write subscription at the scope of the Event Grid topic.
  - **Resource Format**:

```
/subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/Microsoft.EventGrid/topics/{topic-name}
```

These notes provide an overview of how access control is implemented within Azure Event Grid using Azure RBAC, detailing the roles and the necessary permissions for managing event subscriptions.

Receive Events by Using Webhooks

Webhook Integration with Azure Event Grid

- **Delivery Method**: Event Grid sends events via HTTP POST requests to configured endpoints.

Endpoint Validation

- **Purpose**: To confirm ownership of the webhook endpoint and prevent spam.

Services with Automatic Validation:

- Azure Logic Apps with Event Grid Connector
- Azure Automation via webhook
- Azure Functions with Event Grid Trigger

For Other Endpoints:

- **Validation Handshake**:
  - **Synchronous**:
    - Event Grid sends a validation event during subscription creation.
    - Endpoint must return the `validationCode` from the event data in the HTTP response.
    - Supported in all Event Grid versions.
  - **Asynchronous** (for API version 2018-05-01-preview and later):
    - **Process**:
      - Event Grid sends a `validationUrl` in the event.
      - Manually perform a GET request to this URL within 5 minutes.
    - **Subscription State**:
      - **AwaitingManualAction**: While waiting for manual validation.
      - **Failed**: If manual validation isn't completed in time.

Important Notes:

- The webhook endpoint must return an HTTP 200 status to accept the validation event before entering manual validation mode.
- **Certificates**: Self-signed certificates are not supported for validation; use certificates from a commercial CA.

Filter Events

Options for Filtering Events in Event Subscriptions

1. Event Type Filtering

- **Default**: All event types are sent.
- **Custom**: Specify particular event types to receive.
  - **Example**: Filter for `Microsoft.Resources.ResourceWriteSuccess` to only receive resource update events.

JSON Syntax for Event Type Filtering:

```
"filter": {
  "includedEventTypes": [
    "Microsoft.Resources.ResourceWriteFailure",
    "Microsoft.Resources.ResourceWriteSuccess"
  ]
}
```

2. Subject Filtering

- **Methods**:

- **Begins with**: Filter events where the subject starts with a specific string.
- **Ends with**: Filter events where the subject ends with a specific string.
- **Usage**: Useful for focusing on events for specific directories or file types.

JSON Syntax for Subject Filtering:

```
"filter": {
  "subjectBeginsWith": "/blobServices/default/containers/mycontainer/log",
  "subjectEndsWith": ".jpg"
}
```

3. Advanced Filtering

- **Purpose**: Filter based on data fields within the event with various comparison operators.
- **Components**:
  - **operatorType**: Defines the comparison type (e.g., `NumberGreaterThanOrEquals`, `StringContains`).
  - **key**: The field in the event data to filter on.
  - **value** or **values**: The comparison value(s).

JSON Syntax for Advanced Filtering:

```
"filter": {
  "advancedFilters": [
    {
      "operatorType": "NumberGreaterThanOrEquals",
      "key": "Data.Key1",
      "value": 5
    },
    {
      "operatorType": "StringContains",
      "key": "Subject",
      "values": ["container1", "container2"]
    }
  ]
}
```

These filtering options allow you to customize which events are delivered to your endpoint, making your event-driven architecture more efficient by focusing on the events that matter most for your application.

Explore Azure Event Hubs

Objectives:

- Understand how Azure Event Hubs captures events.
- Learn to scale processing applications using Event Hubs.

Introduction to Azure Event Hubs

Completion Status: Completed
Experience Points: 100 XP
Duration: 3 minutes

Key Points about Azure Event Hubs:

- **Service Type**: Big data streaming platform and event ingestion service.
- **Capability**: Processes up to millions of events per second.

Benefits and Capabilities:

- **Ingests streaming data** into a hub for transformation and storage.
- **Integration**: Works with real-time analytics providers and batching/storage solutions.

Learning Outcomes:

- Understand **Event Hubs' benefits** and its role in capturing streaming data.
- Learn how to **process events** in real-time or in batches.
- Gain skills to **perform common operations** using the Event Hubs client library.

Discover Azure Event Hubs

Overview of Azure Event Hubs:

- **Service**: Native cloud data-streaming service, capable of handling millions of events per second with low latency.
- **Compatibility**: Fully compatible with Apache Kafka, allowing for easy migration of Kafka workloads.

Core Features:

- **Multi-Protocol Support**:
    - AMQP 1.0, Apache Kafka, and HTTPS.
- **Partitioned Consumer Model**:
    - Allows for concurrent processing by multiple applications with control over processing speed.
- **Serverless Architecture**:
    - Integrates seamlessly with Azure Functions.
- **SDK Availability**:
    - Supported in .NET, Java, Python, JavaScript for stream processing.

Key Capabilities:

Apache Kafka on Azure Event Hubs

- **Benefit**: Migrate Kafka workloads to Azure without code changes; no need for external Kafka management.

Schema Registry in Event Hubs

- **Function**: Manages schemas for event streaming apps, included free with each Event Hubs namespace.
- **Integration**: Works with both Kafka and Event Hubs SDK applications.

Real-time Stream Processing

- **Integration with Azure Stream Analytics**:
  - No-code editor for Stream Analytics jobs via drag-and-drop.
  - Or use SQL-based query language for real-time processing with extensive function support.

Key Concepts in Azure Event Hubs

Components:

Producer Applications

- **Function**: Ingest data into Event Hubs.
- **Tools**: Use Event Hubs SDKs or Kafka producer clients.

Namespace

- **Definition**: A management container for event hubs or Kafka topics.
- **Purpose**: Manages tasks like capacity allocation, network security, and geo-disaster recovery.

Event Hubs/Kafka Topic

- **Description**: An append-only, distributed log for organizing events.
- **Structure**: Can have multiple partitions.

Partitions

- **Purpose**: Allow scaling of event hubs by providing parallel data streams.
- **Analogy**: Similar to lanes on a highway for data flow.

Consumer Applications

- **Function**: Process data from the event log.
- **Offset Management**: Seek through the log, manage consumer offsets.
- **Clients**: Kafka consumer clients or Event Hubs SDK clients.

Consumer Group

- **Role**: Logical grouping of consumer instances for reading data.
- **Benefit**: Allows multiple consumers to independently read the same data at their own pace and offset.

Explore Event Hubs Capture

Completion Status: Completed
Experience Points: 100 XP
Duration: 3 minutes

Overview of Event Hubs Capture:

- **Purpose**: Automatically capture streaming data from Event Hubs.
- **Destinations**:
  - Azure Blob Storage
  - Azure Data Lake Storage
- **Flexibility**: Capture can be set based on time or size intervals.
- **Economics**: No administrative costs, scales with Event Hubs throughput units (standard tier) or processing units (premium tier).

Functionality:

- **Data Retention**: Event Hubs acts as a durable buffer for telemetry ingress, with data aging off based on retention settings.

- **Partitioned Consumer Model**: Allows for independent consumption of data segments, aiding scalability.

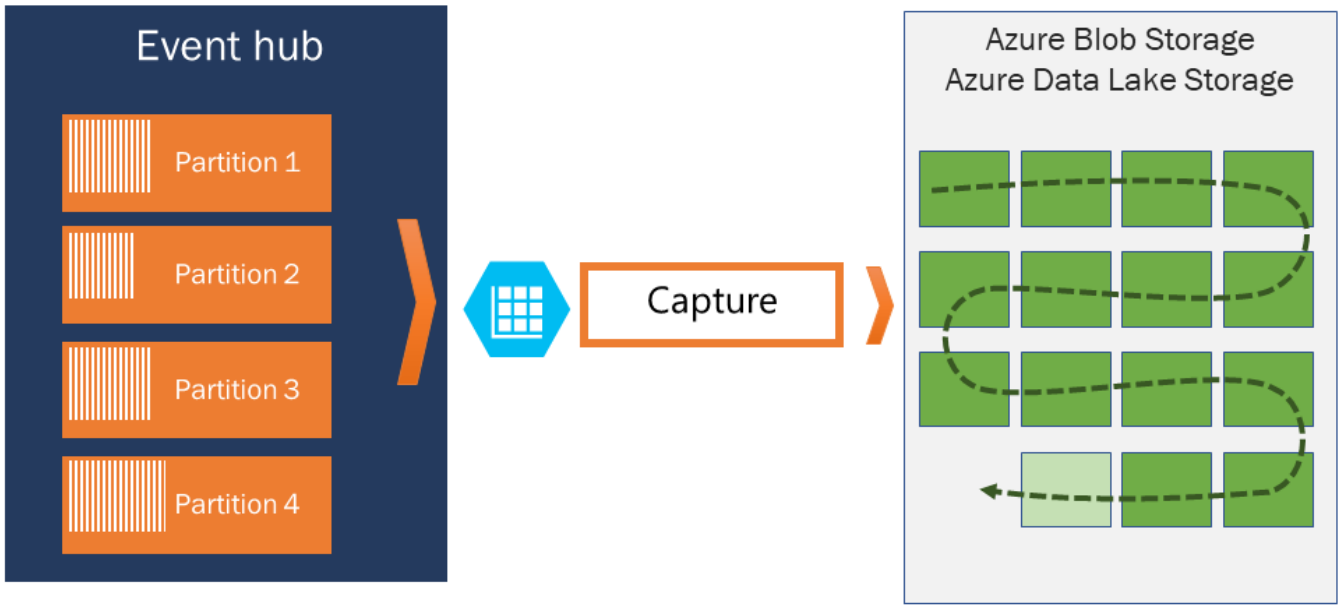
- **Storage Configuration**:

  - Users specify the storage account (Blob or Data Lake) for captured data.
  - Storage can be in the same or different region.

- **Data Format**:

  - **Captured data** is in **Apache Avro** format, which is:
    - Compact and fast binary format.
    - Rich in data structures with inline schema.
    - Compatible with Hadoop ecosystem, Stream Analytics, and Azure Data Factory.

Image Concept:



This feature allows for both real-time and batch-based data pipelines on the same data stream, providing scalability and flexibility for growing solutions.

Capture Windowing in Event Hubs

- **Window Configuration**: Event Hubs Capture allows setting up windows for capture based on:
  - **Size** and **Time**.
  - **Policy**: First trigger (size or time) initiates capture ("first wins policy").
- **Capture Process**:
  - Each partition captures data independently.
  - Data is written as a completed block blob when the interval is met.
- **Storage Path Convention**:

```
{Namespace}/{EventHub}/{PartitionId}/{Year}/{Month}/{Day}/{Hour}/{Minute}/{Second}
```

  - Example Path:

```
https://mystorageaccount.blob.core.windows.net/mycontainer/mynamespace/myeventhub/0/2017/12/08/03/03/17.avro
```

Scaling with Throughput Units

- **Throughput Unit (TU)**:

    - **Ingress**: 1 MB/s or 1,000 events/s per TU.
    - **Egress**: Twice the ingress rate.
    - **Standard Tier**: Configurable from 1-20 TUs, can be increased via support request.

- **Capture Mechanism**:

    - **Bypasses Egress Quotas**: Capture directly from internal storage, preserving egress for other consumers.

- **Empty File Creation**:

    - Event Hubs writes empty files when no data is present to ensure:
        - Predictable processing cadence.
        - Markers for batch processors.

This setup ensures that data capture from Event Hubs is both flexible and scalable, maintaining efficiency in data handling and storage.

Scale Your Processing Application with Azure Event Hubs

Scaling Strategies:

- **Multiple Instances**: Run multiple instances of your application to distribute the load.
- **Load Balancing**:
    - Use `EventProcessorHost` in older versions or `EventProcessorClient` (`EventHubConsumerClient` in Python/JS in newer versions) for load balancing and checkpointing.

Note:

- **Partitioned Consumers** are key for scaling in Event Hubs, promoting parallelism over the competing consumers pattern.

Example Scenario: Home Security Monitoring

- **Context**: Monitoring 100,000 homes with various sensors.
- **Event Hub Configuration**: 16 partitions for sensor data ingestion.

Requirements for Consuming End:

1. **Scale**:

    - Allow multiple consumers to read from different partitions, ensuring load distribution.

2. **Load Balance**:

    - Dynamically adjust the number of consumers based on event volume (e.g., adding new sensor types).

3. **Seamless Resume on Failures**:
    - Other consumers should take over partitions from a failed consumer.
    - Checkpoints ensure resuming from the last known point or slightly before.
4. **Consume and Process Events**:
    - Beyond management, the application must:
        - **Consume** events.
        - **Process** them (e.g., aggregation).
        - **Store** results (e.g., to Blob storage).

Event Processor or Consumer Client in Azure Event Hubs

Core Concepts:

Using SDKs:

- **.NET/Java**: Utilize `EventProcessorClient`.
- **Python/JavaScript**: Use `EventHubConsumerClient`.
- **Recommendation**: Use event processor clients for most production scenarios for reading and processing.

Partition Ownership Tracking:

- **Functionality**: Event processors manage ownership of one or more partitions.
    - **Process**: Ownership is evenly distributed among active instances for an event hub and consumer group.
    - **Mechanism**:
        - Each processor has a unique identifier.
        - Claims partitions via checkpoint store updates.

Receiving Messages:

- **Event Handling**:
    - Each event is delivered to the processing function one at a time per partition.
    - Users must implement retry logic for at-least-once processing.
    - **Caution**: Watch for poisoned messages that might cause infinite retries.
- **Performance**:
    - **Keep processing light**; minimal processing per event.
    - **For complex operations**: Use multiple consumer groups with separate processors.

Checkpointing:

- **Purpose**: Marks the last processed event position in a partition.

    - **Process**: Done per-partition within a consumer group.
    - **Usage**:
        - Resumability: If a processor fails, another can resume from the last checkpoint.
        - Data Retention: Allows returning to older data by specifying a lower offset.

Thread Safety and Concurrency:

- **Default Behavior**: Events from one partition are processed sequentially.
    - **Background Processing**: Event pump runs on separate threads.
    - **Concurrency**: Events from different partitions can be processed concurrently.
    - **Synchronization**: Necessary for shared state across partitions.

Control Access to Events in Azure Event Hubs

Authentication and Authorization Methods:

Microsoft Entra ID (Azure AD)

- **Built-in Roles**:
    - **Azure Event Hubs Data Owner**: Full access to Event Hubs resources.
    - **Azure Event Hubs Data Sender**: Send-only access.
    - **Azure Event Hubs Data Receiver**: Receive-only access.

Authorize with Managed Identities:

- **Setup**: Configure Azure RBAC for managed identities.
- **Roles**: Assign roles like `Azure Event Hubs Data Owner` to grant appropriate access.

Authorize with Microsoft Identity Platform:

- **Benefit**: No need to store credentials in code.
- **Process**:
    - Use OAuth 2.0 for token acquisition.
    - Microsoft Entra ID authenticates the security principal, returns an access token for authorization.

Shared Access Signatures (SAS)

For Publishers:

- **Publisher Concept**: Virtual endpoint for sending messages to Event Hubs.
- **Token Mechanism**:
    - Each client has a unique token for sending to one specific publisher.
    - Tokens are signed with shared keys, not known to clients, preventing token creation by clients.
    - Tokens expire, requiring renewal.

For Consumers:

- **Scope of SAS**:

- SAS policies define permissions at the namespace, event hub, or topic level.
- **Limitation**: SAS does not provide granular control at the consumer group level; permissions apply to all consumer groups within the entity.

Perform Common Operations with Azure Event Hubs Client Library

Completion Status: Completed
Experience Points: 100 XP
Duration: 3 minutes

Operations with Azure.Messaging.EventHubs

Inspect Event Hub Partitions

- **Purpose**: To determine available partitions.
- **Client**: `EventHubProducerClient` can be used for this (concept applies to all clients).

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString,
eventHubName))
{
    string[] partitionIds = await producer.GetPartitionIdsAsync();
}
```

Publish Events

- **Client Creation**: `EventHubProducerClient` for publishing.
- **Method**: Publish events in batches with automatic routing for even distribution and high availability.

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

await using (var producer = new EventHubProducerClient(connectionString,
eventHubName))
{
    using EventDataBatch eventBatch = await producer.CreateBatchAsync();
    eventBatch.TryAdd(new EventData(new BinaryData("First")));
    eventBatch.TryAdd(new EventData(new BinaryData("Second")));

    await producer.SendAsync(eventBatch);
}
```

Read Events

- **Client Creation**: `EventHubConsumerClient` for reading events.
- **Consumer Group**: Uses the default consumer group for simplicity.
- **Note**: This method is for exploration/prototyping, not production. Use `EventProcessorClient` for production.

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup,
connectionString, eventHubName))
{
    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

    await foreach (PartitionEvent receivedEvent in
consumer.ReadEventsAsync(cancellationSource.Token))
    {
        // Processing logic for each event
    }
}
```

These examples demonstrate basic operations for interacting with Azure Event Hubs using the C# client library. Remember to use production-ready methods like `EventProcessorClient` for actual deployments.

Reading and Processing Events in Azure Event Hubs

Read Events from a Specific Partition

- **Purpose**: To consume events from a particular partition.
- **Starting Point**: Define where to start reading from in the event stream.

```
var connectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE >>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";

string consumerGroup = EventHubConsumerClient.DefaultConsumerGroupName;

await using (var consumer = new EventHubConsumerClient(consumerGroup,
connectionString, eventHubName))
{
    // Set the starting position to the earliest event
    EventPosition startingPosition = EventPosition.Earliest;
    // Get the first partition's ID
    string partitionId = (await consumer.GetPartitionIdsAsync()).First();

    using var cancellationSource = new CancellationTokenSource();
    cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));
```

```
    await foreach (PartitionEvent receivedEvent in
consumer.ReadEventsFromPartitionAsync(partitionId, startingPosition,
cancellationSource.Token))
    {
        // Process each received event here
    }
}
```

Process Events Using EventProcessorClient

- **For Production**: Recommended for robust, scalable event processing.
- **Dependencies**: Requires Azure Storage for state persistence.

```
var cancellationSource = new CancellationTokenSource();
cancellationSource.CancelAfter(TimeSpan.FromSeconds(45));

var storageConnectionString = "<< CONNECTION STRING FOR THE STORAGE ACCOUNT >>";
var blobContainerName = "<< NAME OF THE BLOB CONTAINER >>";

var eventHubsConnectionString = "<< CONNECTION STRING FOR THE EVENT HUBS NAMESPACE
>>";
var eventHubName = "<< NAME OF THE EVENT HUB >>";
var consumerGroup = "<< NAME OF THE EVENT HUB CONSUMER GROUP >>";

// Define event handling methods
Task processEventHandler(ProcessEventArgs eventArgs) => Task.CompletedTask;
Task processErrorHandler(ProcessErrorEventArgs eventArgs) => Task.CompletedTask;

// Initialize BlobContainerClient for state storage
var storageClient = new BlobContainerClient(storageConnectionString,
blobContainerName);

// Create and configure the EventProcessorClient
var processor = new EventProcessorClient(storageClient, consumerGroup,
eventHubsConnectionString, eventHubName);

processor.ProcessEventAsync += processEventHandler;
processor.ProcessErrorAsync += processErrorHandler;

await processor.StartProcessingAsync();

try
{
    // Keep the application running to allow background processing
    await Task.Delay(Timeout.Infinite, cancellationSource.Token);
}
catch (TaskCanceledException)
{
    // This exception is expected when the delay is canceled
}
```

```
try
{
    // Stop the processor when done or if cancellation is requested
    await processor.StopProcessingAsync();
}
finally
{
    // Clean up event handlers to prevent memory leaks
    processor.ProcessEventAsync -= processEventHandler;
    processor.ProcessErrorAsync -= processErrorHandler;
}
```

These examples illustrate how to interact with specific partitions or use a more production-ready approach with `EventProcessorClient` for handling events in Azure Event Hubs.

AZ-204: Develop Message-Based Solutions

Topics Covered:

- **Azure Queue Storage**: Learn how to implement and manage queues for scalable and durable message storage.
- **Azure Service Bus**: Dive into creating applications that leverage Service Bus for reliable message handling, including topics and subscriptions.

Key Takeaways:

- **Building Message-Based Architectures**: Understand how to integrate Azure Service Bus and Azure Queue Storage into your applications for effective message handling.

Prerequisites:

- **Experience**: At least one year of experience in developing scalable solutions, covering all phases of software development.
- **Azure Knowledge**:
  - Basic understanding of Azure, cloud concepts, services, and familiarity with the Azure portal.
- **Recommended for Beginners**:
  - If new to Azure or cloud computing, consider taking the **AZ-900: Azure Fundamentals** course first.

This should give you a concise overview to get you started or to refresh your memory on the course content. Remember, hands-on practice with Azure services will solidify your understanding!

Discover Azure Message Queues

Overview:

- **Focus**: Integration of **Azure Service Bus** and **Azure Queue Storage** into your solutions.
- **Skills**:

- **Sending messages** using .NET.
- **Receiving messages** using .NET.

Key Topics:

- **Azure Service Bus**:
  - Learn about topics, queues, and subscriptions for message handling.
- **Azure Queue Storage**:
  - Understand how to manage and interact with queues for durable message storage.

Practical Application:

- **Sending Messages**:

```
using Azure.Messaging.ServiceBus;

// Connection string to your Service Bus namespace
string connectionString = "<YOUR_CONNECTION_STRING>";
string queueName = "yourQueueName";

// Create a ServiceBusClient that will be used to create Sender/Receiver
objects
await using (ServiceBusClient client = new
ServiceBusClient(connectionString))
{
    // create a sender for the queue
    ServiceBusSender sender = client.CreateSender(queueName);

    // create a message that we can send
    ServiceBusMessage message = new ServiceBusMessage("Hello, World!");

    // send the message
    await sender.SendMessageAsync(message);
}
```

- **Receiving Messages**:

```
using Azure.Messaging.ServiceBus;

string connectionString = "<YOUR_CONNECTION_STRING>";
string queueName = "yourQueueName";

await using (ServiceBusClient client = new
ServiceBusClient(connectionString))
{
    // create a receiver that we can use to receive and settle the message
    ServiceBusReceiver receiver = client.CreateReceiver(queueName);

    // receive the message
```

```
    ServiceBusReceivedMessage receivedMessage = await
receiver.ReceiveMessageAsync();

    if (receivedMessage != null)
    {
        Console.WriteLine($"Received: {receivedMessage.Body}");
        // complete the message, thereby deleting it from the queue
        await receiver.CompleteMessageAsync(receivedMessage);
    }
}
```

These code blocks illustrate basic operations for sending and receiving messages with Azure Service Bus using .NET. Remember to replace `<YOUR_CONNECTION_STRING>` with your actual connection string.

This module sets the foundation for understanding and implementing message-based solutions in Azure, focusing on practical .NET integration.

Introduction

Azure Queue Mechanisms:

- **Service Bus Queues**:
  - **Part of**: Broader Azure messaging infrastructure.
  - **Features**: Supports queuing, publish/subscribe, and advanced integration patterns.
  - **Purpose**: Designed for integrating applications or components across different communication protocols, data contracts, trust domains, or network environments.
- **Storage Queues**:
  - **Part of**: Azure Storage infrastructure.
  - **Capabilities**: Allows storage of large numbers of messages, accessible globally via HTTP/HTTPS with authenticated calls.
  - **Capacity**: Can handle millions of messages, limited by the storage account's total capacity.
  - **Usage**: Frequently used for creating backlogs for asynchronous work processing.

Learning Outcomes:

After completing this module, you will:

- **Choose** the right queue mechanism for your solution.
- **Explain** the operations of Service Bus messaging entities.
- **Send and Receive Messages** from a Service Bus queue using .NET:

```
using Azure.Messaging.ServiceBus;

// Example for sending a message
string connectionString = "<YOUR_SERVICE_BUS_CONNECTION_STRING>";
```

```
string queueName = "yourQueueName";

await using (ServiceBusClient client = new
ServiceBusClient(connectionString))
{
    ServiceBusSender sender = client.CreateSender(queueName);
    ServiceBusMessage message = new ServiceBusMessage("Hello from Service
Bus!");
    await sender.SendMessageAsync(message);
}

// Example for receiving a message
await using (ServiceBusClient client = new
ServiceBusClient(connectionString))
{
    ServiceBusReceiver receiver = client.CreateReceiver(queueName);
    ServiceBusReceivedMessage receivedMessage = await
receiver.ReceiveMessageAsync();
    if (receivedMessage != null)
    {
        Console.WriteLine($"Received: {receivedMessage.Body}");
        await receiver.CompleteMessageAsync(receivedMessage);
    }
}
```

- **Identify** key components of Azure Queue Storage.
- **Create Queues and Manage Messages** in Azure Queue Storage using .NET:

```
using Azure.Storage.Queues;

// Example for creating a queue and inserting a message
string connectionString = "<YOUR_STORAGE_CONNECTION_STRING>";
string queueName = "yourQueueName";

QueueClient queue = new QueueClient(connectionString, queueName);
await queue.CreateIfNotExistsAsync();

await queue.SendMessageAsync("Hello from Storage Queue!");

// Example for peeking at messages
var response = await queue.PeekMessagesAsync(maxMessages: 1);
foreach (PeekedMessage message in response.Value)
{
    Console.WriteLine($"Peeked message: {message.MessageText}");
}
```

This module provides a foundational understanding of Azure's queue services, equipping you with the knowledge to decide and implement the appropriate queue technology for your project needs. Remember to

replace placeholders like `<YOUR_SERVICE_BUS_CONNECTION_STRING>` and `<YOUR_STORAGE_CONNECTION_STRING>` with your actual connection strings.

Choose a Message Queue Solution

Overview:

- Both **Storage Queues** and **Service Bus Queues** are viable options, but they cater to different needs.

When to Use Service Bus Queues:

- **Non-Polling Message Reception**: Use when you want to receive messages without constant polling, thanks to long-polling via TCP-based protocols.
- **FIFO Delivery**: When your solution needs guaranteed first-in-first-out message order.
- **Duplicate Detection**: For automatic detection and handling of duplicate messages.
- **Parallel Stream Processing**: When messages are part of long-running streams identified by session IDs, allowing for parallel processing where nodes compete for streams.
- **Transactional Operations**: For atomic send/receive operations in transactions.
- **Message Size**: When dealing with messages larger than 64 KB but within the 256 KB or 1 MB limit of Service Bus tiers.
- **Role-Based Access**: When you need to apply different access rights for senders and receivers.

When to Use Storage Queues:

- **Large Message Storage**: If your application requires storing over 80 GB of messages in a queue.
- **Progress Tracking**: When you need to track processing progress, allowing other workers to pick up where others left off in case of failures.
- **Server-Side Logging**: For comprehensive logging of all queue transactions.

Code Example:

Service Bus Queue Example for FIFO Delivery & Session Handling:

```
using Azure.Messaging.ServiceBus;

// Example scenario where messages are part of a session for ordered processing
string connectionString = "<YOUR_SERVICE_BUS_CONNECTION_STRING>";
string queueName = "sessionQueue";

await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusSender sender = client.CreateSender(queueName);
    for (int i = 0; i < 10; i++)
    {
        ServiceBusMessage message = new ServiceBusMessage($"Message {i}")
        {
            SessionId = "Session1"
        };
        await sender.SendMessageAsync(message);
```

```
    }

    // Receiving messages in order due to session lock
    ServiceBusSessionReceiver sessionReceiver = await
client.AcceptNextSessionAsync(queueName);
    while (true)
    {
        ServiceBusReceivedMessage message = await
sessionReceiver.ReceiveMessageAsync();
        if (message == null) break;
        Console.WriteLine($"Received: {message.Body}");
        await sessionReceiver.CompleteMessageAsync(message);
    }
}
```

Storage Queue Example for Large Message Storage:

```
using Azure.Storage.Queues;

// Example for managing large queues
string connectionString = "<YOUR_STORAGE_CONNECTION_STRING>";
string queueName = "largeMessageQueue";

QueueClient queue = new QueueClient(connectionString, queueName);

// Enqueue messages
for (int i = 0; i < 100000; i++) // Assuming each message is small to not exceed
the storage limit quickly
{
    await queue.SendMessageAsync($"Message {i}");
}

// Dequeue and process messages
QueueMessage[] messages = await
queue.ReceiveMessagesAsync(32).Value.ToArrayAsync(); // Batch receive for
efficiency
foreach (QueueMessage message in messages)
{
    // Process message here
    Console.WriteLine($"Processing: {message.MessageText}");
    await queue.DeleteMessageAsync(message.MessageId, message.PopReceipt); //
Remove message after processing
}
```

These recommendations help in choosing the right queue technology based on your application requirements, ensuring scalability, efficiency, and reliability. Remember to replace placeholders with actual connection strings.

Explore Azure Service Bus

Overview:

- **Azure Service Bus**: A fully managed enterprise message broker offering queues and publish-subscribe topics.
- **Purpose**: Decouples applications and services, facilitating data transfer via messages.

Messaging Fundamentals:

- **Messages**: Containers with metadata, capable of holding various data formats like JSON, XML, Apache Avro, or Plain Text.

Common Messaging Scenarios:

- **Data Transfer**: For business data like sales orders, inventory movements, etc.
- **Application Decoupling**: Enhances reliability and scalability; allows for asynchronous operation where clients and services don't need to be online simultaneously.
- **Topics and Subscriptions**: Facilitates one-to-many communication models.
- **Message Sessions**: Supports workflows requiring message sequencing or deferral.

Service Bus Tiers:

- **Basic, Standard, and Premium** are available, each designed for different use cases:
    - **Premium Tier**:
        - **Use Case**: For mission-critical applications needing scale, performance, and availability.
        - **Advantages**:
            - High throughput
            - Predictable performance
            - Fixed pricing model
            - Scalability of workload
            - Supports messages up to 100 MB
    - **Standard Tier**:
        - **Use Case**: More varied, less critical applications.
        - **Characteristics**:
            - Variable throughput and latency
            - Pay-as-you-go pricing
            - Message size limit up to 256 KB

Table Summary:

Feature	Premium	Standard
Throughput	High	Variable
Performance	Predictable	Variable latency
Pricing	Fixed	Pay as you go
Scalability	Ability to scale workload	N/A
Message Size	Up to 100 MB	Up to 256 KB

Code Example:

Here's a basic example of sending and receiving a message using Azure Service Bus:

```
using Azure.Messaging.ServiceBus;

// Connection string to your Service Bus namespace
string connectionString = "<YOUR_CONNECTION_STRING>";
string queueName = "yourQueueName";

// Sending a message
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusSender sender = client.CreateSender(queueName);
    ServiceBusMessage message = new ServiceBusMessage("Hello from Service Bus!");
    await sender.SendMessageAsync(message);
}

// Receiving a message
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusReceiver receiver = client.CreateReceiver(queueName);
    ServiceBusReceivedMessage receivedMessage = await
receiver.ReceiveMessageAsync();

    if (receivedMessage != null)
    {
        Console.WriteLine($"Received: {receivedMessage.Body}");
        await receiver.CompleteMessageAsync(receivedMessage);
    }
}
```

Remember, when choosing a tier, consider the needs of your application for performance, scalability, and message handling capabilities. Always replace `<YOUR_CONNECTION_STRING>` with your actual Service Bus connection string.

Advanced Features - Azure Service Bus

Feature Overview:

Feature	Description
Message Sessions	Ensures FIFO processing for related messages. Sessions allow exclusive and ordered handling.
Autoforwarding	Chains a queue or subscription to another within the same namespace for automatic message forwarding.
Dead-letter Queue (DLQ)	Stores messages that couldn't be delivered. Allows inspection and removal of messages from the DLQ.

Feature	Description
Scheduled Delivery	Delays message processing by scheduling when a message becomes available.
Message Deferral	Allows clients to postpone message retrieval until later, keeping the message in the queue or subscription.
Transactions	Groups operations for atomic execution on a single messaging entity (queue, topic, or subscription).
Filtering and Actions	Defines which messages subscribers receive from topics via subscription rules.
Autodelete on Idle	Automatically deletes a queue if it remains idle for a specified duration (minimum 5 minutes).
Duplicate Detection	Prevents duplicate messages by allowing resend while discarding duplicates.
Security Protocols	Supports SAS, RBAC, and Managed Identities for securing resources.
Geo-Disaster Recovery	Ensures data processing can continue in another region or datacenter during outages.
Security	Utilizes AMQP 1.0 and HTTP/REST protocols for secure communication.

Compliance and Protocols:

- **Primary Protocol**: AMQP 1.0 (ISO/IEC standard), enabling interoperability with various brokers.
- **Service Bus Premium**: Compliant with JMS 2.0 API for Java/Jakarta EE.

Client Libraries:

- **.NET**: Azure Service Bus for .NET
- **Java**: Azure Service Bus libraries for Java, including JMS 2.0 provider
- **JavaScript/TypeScript**: Azure Service Bus Modules for JavaScript and TypeScript
- **Python**: Azure Service Bus libraries for Python

These client libraries are fully supported by the Azure SDK, offering robust interfaces for interacting with Service Bus from various programming languages.

Code Example:

Here's a quick example of using message sessions to ensure FIFO processing with .NET:

```
using Azure.Messaging.ServiceBus;

// Connection string to your Service Bus namespace
string connectionString = "<YOUR_CONNECTION_STRING>";
string queueName = "yourSessionQueue";
```

```
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    // Sending messages with session ID
    ServiceBusSender sender = client.CreateSender(queueName);
    for (int i = 0; i < 5; i++)
    {
        ServiceBusMessage message = new ServiceBusMessage($"Message {i}")
        {
            SessionId = "Session1" // All messages in the same session to ensure
order
        };
        await sender.SendMessageAsync(message);
    }

    // Receiving messages in order
    ServiceBusSessionReceiver sessionReceiver = await
client.AcceptNextSessionAsync(queueName);
    while (true)
    {
        ServiceBusReceivedMessage message = await
sessionReceiver.ReceiveMessageAsync();
        if (message == null) break;
        Console.WriteLine($"Received in order: {message.Body} from session
{message.SessionId}");
        await sessionReceiver.CompleteMessageAsync(message);
    }
}
```

Remember to replace `<YOUR_CONNECTION_STRING>` with your actual Service Bus connection string. This example demonstrates how sessions can be used to guarantee message order in Service Bus.

Discover Service Bus Queues, Topics, and Subscriptions

Service Bus Messaging Entities:

- **Queues**:
    - **FIFO Delivery**: Messages are delivered in the order they are added, to one or more competing consumers.
    - **Single Consumer**: Only one consumer processes each message.
    - **Durable Storage**: Messages are stored durably, allowing asynchronous processing.
    - **Benefits**:
        - **Load-Leveling**: Enables different rates for sending and receiving messages, handling average rather than peak load.
        - **Loose Coupling**: Producers and consumers are independent, allowing upgrades without impact.
    - **Creation**: Can be created via Azure portal, PowerShell, CLI, or Resource Manager templates.

- **Usage**: Messages can be sent and received using clients in C#, Java, Python, JavaScript.

Receive Modes:

- **Receive and Delete**:
  - **Process**: Marks the message as consumed upon receipt, suitable for scenarios where message loss is acceptable.
  - **Risk**: If the consumer crashes post-receipt but before processing, the message is lost.
- **Peek Lock**:
  - **Process**:
    - **Stage 1**: Locks the message, preventing others from receiving it, and returns it to the consumer.
    - **Stage 2**: After processing, the consumer confirms completion, and Service Bus marks the message as consumed.
  - **Advantages**:
    - **Error Handling**: If processing fails, the consumer can abandon the message, making it available again.
    - **Timeouts**: If processing exceeds the lock timeout, the message is unlocked for another attempt.

Code Example:

Here's a simple example of sending a message to a queue and receiving it with both receive modes in C#:

```
using Azure.Messaging.ServiceBus;

string connectionString = "<YOUR_CONNECTION_STRING>";
string queueName = "yourQueueName";

// Sending a message
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusSender sender = client.CreateSender(queueName);
    await sender.SendMessageAsync(new ServiceBusMessage("Hello from Service
Bus!"));
}

// Receiving with 'Receive and Delete' mode
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusReceiver receiver = client.CreateReceiver(queueName, new
ServiceBusReceiverOptions { ReceiveMode = ServiceBusReceiveMode.ReceiveAndDelete
});
    ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();
    if (message != null) Console.WriteLine($"Received (ReceiveAndDelete):
{message.Body}");
}
```

```
// Receiving with 'Peek Lock' mode
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusReceiver receiver = client.CreateReceiver(queueName, new
ServiceBusReceiverOptions { ReceiveMode = ServiceBusReceiveMode.PeekLock });
    ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();
    if (message != null)
    {
        Console.WriteLine($"Received (PeekLock): {message.Body}");
        await receiver.CompleteMessageAsync(message); // Mark message as processed
    }
}
```

Replace `<YOUR_CONNECTION_STRING>` with your actual Service Bus connection string. This example illustrates how different receive modes work, highlighting their implications for message handling in your applications.

Topics and Subscriptions

Overview:

- **Queues vs. Topics**:
  - **Queues** allow one consumer to process each message.
  - **Topics and Subscriptions** provide a one-to-many communication model (publish-subscribe pattern), scaling to multiple recipients.

Functionality:

- **Topics**:
  - Publishers send messages to a topic.
- **Subscriptions**:
  - Consumers receive messages from subscriptions to the topic, not directly from the topic itself.
  - Each subscription can act like a virtual queue, getting copies of messages sent to the topic.

Message Handling:

- **Message Sending**: Similar to queues, messages are sent to a topic.
- **Message Receiving**: Consumers retrieve messages from subscriptions, which can have multiple consumers in a competing consumer pattern similar to queues.

Benefits:

- **Scalability**: Supports multiple subscribers.
- **Decoupling**: Temporal decoupling, load leveling, and load balancing are supported, mirroring queue capabilities.

Rules and Actions:

- **Filtering**:

    - Subscriptions can filter messages based on properties (system or custom), using SQL-like expressions.
    - Without a filter, all messages go to the subscription.
- **Actions**:
    - Modify message properties based on filter criteria.

Code Example:

Here's how you might set up a topic, subscription, and apply rules using C#:

```
using Azure.Messaging.ServiceBus;
using Azure.Messaging.ServiceBus.Administration;

string connectionString = "<YOUR_CONNECTION_STRING>";
string topicName = "myTopic";
string subscriptionName = "mySubscription";

// Create Topic
ServiceBusAdministrationClient adminClient = new
ServiceBusAdministrationClient(connectionString);
await adminClient.CreateTopicAsync(topicName);

// Create Subscription with filter
await adminClient.CreateSubscriptionAsync(new CreateSubscriptionOptions(topicName,
subscriptionName)
{
    // SQL filter example to receive messages where Label equals 'Alert'
    // If no filter is specified, all messages are copied to this subscription
    SqlFilter = new SqlFilter("Label = @string('Alert')")
});

// Sending a message to the topic
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusSender sender = client.CreateSender(topicName);
    await sender.SendMessageAsync(new ServiceBusMessage("This is an alert
message")
    {
        Label = "Alert" // The message will match the subscription filter
    });
}

// Receiving from the subscription
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusReceiver receiver = client.CreateReceiver(topicName,
subscriptionName);
    ServiceBusReceivedMessage message = await receiver.ReceiveMessageAsync();
    if (message != null)
    {
        Console.WriteLine($"Received from subscription: {message.Body}");
        await receiver.CompleteMessageAsync(message);
```

```
    }
}
```

Replace `<YOUR_CONNECTION_STRING>` with your actual Service Bus connection string. This example demonstrates creating a topic, setting up a subscription with a filter, sending a message to the topic, and receiving it through the subscription. Remember, without a filter, all messages would be received by the subscription.

Explore Service Bus Message Payloads and Serialization

Message Composition:

- **Payload**: Binary data that Service Bus doesn't process; it's for the application to handle.
- **Metadata**: Key-value pair properties:
  - **Broker Properties**: System-defined, control message functionality or map to standardized metadata.
  - **User Properties**: Custom key-value pairs set by the application.

Message Routing and Correlation:

- **Key Properties for Routing**:
  - **To, ReplyTo, ReplyToSessionId, MessageId, CorrelationId, SessionId**

Routing Patterns:

- **Simple Request/Reply**:
  - Publisher sends to a queue, expects a reply to their queue specified in `ReplyTo`.
  - Consumer replies using `CorrelationId` set to the `MessageId` of the original message.
- **Multicast Request/Reply**:
  - Similar to simple request/reply but uses a topic for message distribution, allowing multiple subscribers to respond.
- **Multiplexing**:
  - Uses `SessionId` to group related messages for exclusive handling by a receiver under session lock.
- **Multiplexed Request/Reply**:
  - Enables multiple publishers to share a reply queue using `ReplyToSessionId` to direct responses to specific sessions.

Routing Techniques:

- **Within Namespace**: Uses autoforward chaining and topic subscription rules.
- **Across Namespaces**: Can be handled with Azure Logic Apps.
- **To Property**: Reserved for future use; applications should use user properties for custom routing.

Code Example:

Here's an example in C# demonstrating a simple request/reply scenario:

```
using Azure.Messaging.ServiceBus;

string connectionString = "<YOUR_CONNECTION_STRING>";
string requestQueueName = "requestQueue";
string replyQueueName = "replyQueue";

// Publisher sending a request
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusSender sender = client.CreateSender(requestQueueName);
    ServiceBusMessage requestMessage = new ServiceBusMessage("Request message")
    {
        ReplyTo = replyQueueName,
        MessageId = "unique-request-id"
    };
    await sender.SendMessageAsync(requestMessage);
}

// Consumer receiving and replying
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusReceiver receiver = client.CreateReceiver(requestQueueName);
    ServiceBusReceivedMessage receivedMessage = await
receiver.ReceiveMessageAsync();
    if (receivedMessage != null)
    {
        Console.WriteLine($"Received request: {receivedMessage.Body}");
        ServiceBusSender replySender =
client.CreateSender(receivedMessage.ReplyTo);
        ServiceBusMessage replyMessage = new ServiceBusMessage("Reply message")
        {
            CorrelationId = receivedMessage.MessageId
        };
        await replySender.SendMessageAsync(replyMessage);
        await receiver.CompleteMessageAsync(receivedMessage);
    }
}

// Publisher receiving the reply
await using (ServiceBusClient client = new ServiceBusClient(connectionString))
{
    ServiceBusReceiver replyReceiver = client.CreateReceiver(replyQueueName);
    ServiceBusReceivedMessage reply = await replyReceiver.ReceiveMessageAsync();
    if (reply != null)
    {
        Console.WriteLine($"Received reply: {reply.Body}");
        await replyReceiver.CompleteMessageAsync(reply);
```

```
        }
    }
```

Replace `<YOUR_CONNECTION_STRING>` with your actual Service Bus connection string. This example shows how to send a request, process it, and then receive a reply using the properties for correlation.

Payload Serialization

Payload Characteristics:

- **Binary Block**: When in transit or stored in Service Bus, the payload is an opaque binary block.

ContentType Property:

- **Purpose**: Describes the payload format.
- **Format**: Suggested to use MIME content-type descriptions per IETF RFC2045, e.g., `application/json;charset=utf-8`.

Serialization in Different .NET Versions:

- **.NET Framework**:
    - Supports creating `BrokeredMessage` instances directly from .NET objects.
    - **Legacy SBMP Protocol**:
        - Uses default binary serializer or externally supplied serializers.
- **.NET Standard and Java**:
    - No direct support for creating messages from arbitrary objects; requires explicit serialization.

AMQP Protocol:

- **Serialization**: Objects are serialized into an AMQP graph which includes `ArrayList` and `IDictionary<string, object>`.
- **Deserialization**: Use `GetBody<T>()` method on the receiver side to retrieve objects, specifying the expected type `T`.

Explicit Control Over Serialization:

- **Recommendation**:
    - Applications should serialize objects to streams before sending and deserialize on receipt for more control.
    - **Reason**: AMQP's binary encoding is ecosystem-specific, making it challenging for HTTP clients to decode.

Code Example:

Here's an example demonstrating explicit serialization and deserialization for both AMQP and HTTP compatibility in C#:

```
using System;
using System.IO;
using System.Text;
using Azure.Messaging.ServiceBus;
using Newtonsoft.Json;

// Example object to serialize
public class MyObject {
    public string Name { get; set; }
    public int Age { get; set; }
}

string connectionString = "<YOUR_CONNECTION_STRING>";
string queueName = "myQueue";

// Serialize object to JSON string
var objToSend = new MyObject { Name = "John Doe", Age = 30 };
var jsonString = JsonConvert.SerializeObject(objToSend);
var payload = Encoding.UTF8.GetBytes(jsonString);

await using (var client = new ServiceBusClient(connectionString))
{
    // Sending the serialized object
    var sender = client.CreateSender(queueName);
    await sender.SendMessageAsync(new ServiceBusMessage(payload)
    {
        ContentType = "application/json;charset=utf-8"
    });

    // Receiving and deserializing the message
    var receiver = client.CreateReceiver(queueName);
    var receivedMessage = await receiver.ReceiveMessageAsync();
    if (receivedMessage != null)
    {
        var body = Encoding.UTF8.GetString(receivedMessage.Body.ToArray());
        var deserializedObj = JsonConvert.DeserializeObject<MyObject>(body);
        Console.WriteLine($"Received: Name: {deserializedObj.Name}, Age:
{deserializedObj.Age}");
        await receiver.CompleteMessageAsync(receivedMessage);
    }
}
```

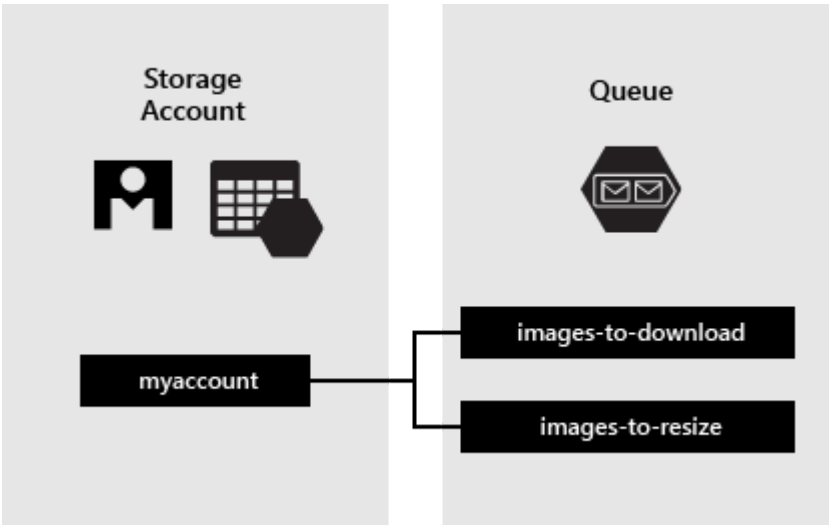
Replace `<YOUR_CONNECTION_STRING>` with your actual Service Bus connection string. This example uses JSON serialization for compatibility across different protocols, ensuring the payload can be understood by both AMQP and HTTP clients.

Explore Azure Queue Storage

Overview:

- **Purpose**: To store large numbers of messages accessible globally via HTTP/HTTPS with authenticated calls.
- **Message Size**: Up to 64 KB per message.
- **Capacity**: Queues can contain millions of messages, limited by the storage account's capacity.
- **Use Case**: Ideal for creating a backlog of work for asynchronous processing.

Components of Queue Service:



- **URL Format**:
  - Queues are accessible via URLs like: `https://<storage account>.queue.core.windows.net/<queue>`
  - Example: `https://myaccount.queue.core.windows.net/images-to-download`
- **Storage Account**:
  - The gateway for all Azure Storage interactions.
- **Queue**:
  - A collection of messages.
  - **Naming**: Must be in lowercase.
- **Message**:
  - **Content**: Can be in any format.
  - **Size Limit**: 64 KB maximum.
  - **Time-to-Live (TTL)**:
    - **Pre-2017-07-29**: Max TTL is 7 days.
    - **2017-07-29 or later**: TTL can be any positive number or -1 for no expiration. Default is 7 days if not specified.

Code Example:

```
using Azure.Storage.Queues;
```

```
string connectionString = "<YOUR_STORAGE_CONNECTION_STRING>";
string queueName = "myqueue";

// Create a queue client
QueueClient queue = new QueueClient(connectionString, queueName);

// Create the queue if it doesn't exist
await queue.CreateIfNotExistsAsync();

// Add a message to the queue
await queue.SendMessageAsync("Hello, Queue!");

// Receive messages from the queue (up to 32 messages in one call for efficiency)
QueueMessage[] messages = await
queue.ReceiveMessagesAsync(32).Value.ToArrayAsync();
foreach (QueueMessage message in messages)
{
    Console.WriteLine($"Message: {message.MessageText}");

    // Process the message here

    // Delete the message after processing to remove it from the queue
    await queue.DeleteMessageAsync(message.MessageId, message.PopReceipt);
}
```

Replace `<YOUR_STORAGE_CONNECTION_STRING>` with your actual Azure Storage account connection string. This example demonstrates creating a queue, adding a message, and then processing messages by receiving and deleting them after processing. Remember, in a real-world scenario, you would handle exceptions, manage message visibility timeouts, and consider the implications of message deletion only after successful processing.

Create and Manage Azure Queue Storage and Messages by using .NET

Dependencies:

- **NuGet Packages**:
  - `Azure.Core`: Provides shared primitives for Azure SDK client libraries.
  - `Azure.Storage.Common`: Infrastructure shared by Azure Storage client libraries.
  - `Azure.Storage.Queues`: Specific for Azure Queue Storage operations.
  - `System.Configuration.ConfigurationManager`: For accessing configuration settings.

Code Examples:

- **Creating the Queue Service Client**:
  - The `QueueClient` class is used to interact with queues in Azure Queue Storage.

```
// Using statement for Azure.Storage.Queues
using Azure.Storage.Queues;
```

```
// Assuming 'connectionString' and 'queueName' are defined
QueueClient queueClient = new QueueClient(connectionString, queueName);
```

- **Creating a Queue**:

```
using Azure.Storage.Queues;
using System.Configuration;

// Retrieve the connection string from app settings
string connectionString =
ConfigurationManager.AppSettings["StorageConnectionString"];

// Create a QueueClient
string queueName = "myqueue";
QueueClient queueClient = new QueueClient(connectionString, queueName);

// Create the queue if it doesn't exist
queueClient.CreateIfNotExists();
```

Notes:

- Ensure `StorageConnectionString` is correctly configured in your application's configuration file (like `app.config` or `web.config` in .NET Framework, or `appsettings.json` in .NET Core).
- `CreateIfNotExists()` checks if the queue already exists before creating it, avoiding conflicts or unnecessary API calls.

This setup allows you to initialize and manage queues in Azure Queue Storage from a .NET application, providing a foundation for further operations like adding, reading, or deleting messages.

Managing Messages in Azure Queue Storage with .NET

- **Status**: Covered in the script

Insert a Message into a Queue:

```
using Azure.Storage.Queues;
using System.Configuration;

// Retrieve the connection string from app settings
string connectionString =
ConfigurationManager.AppSettings["StorageConnectionString"];
string queueName = "myqueue";
string message = "Hello, Queue!";

// Create a QueueClient
QueueClient queueClient = new QueueClient(connectionString, queueName);

// Create the queue if it doesn't exist
```

```
queueClient.CreateIfNotExists();

if (queueClient.Exists())
{
    // Send a message to the queue
    queueClient.SendMessage(message);
}
```

Peek at the Next Message:

```
using Azure.Storage.Queues;
using System.Configuration;

// Retrieve the connection string from app settings
string connectionString =
ConfigurationManager.AppSettings["StorageConnectionString"];
string queueName = "myqueue";

// Create a QueueClient
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    // Peek at the next message
    PeekedMessage[] peekedMessages = queueClient.PeekMessages();
    foreach (PeekedMessage peekedMessage in peekedMessages)
    {
        Console.WriteLine($"Peeked Message: {peekedMessage.MessageText}");
    }
}
```

Change the Contents of a Queued Message:

```
using Azure.Storage.Queues;
using System;
using System.Configuration;

// Retrieve the connection string from app settings
string connectionString =
ConfigurationManager.AppSettings["StorageConnectionString"];
string queueName = "myqueue";

// Create a QueueClient
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    // Get the message from the queue
```

```
    QueueMessage[] messages = queueClient.ReceiveMessages().Value.ToArray();
    if (messages.Length > 0)
    {
        QueueMessage message = messages[0];

        // Update the message contents
        queueClient.UpdateMessage(
            message.MessageId,
            message.PopReceipt,
            "Updated contents",
            TimeSpan.FromSeconds(60.0)  // Make it invisible for another 60
seconds
        );
    }
}
```

Notes:

- **SendMessage**: Adds a new message to the queue. Messages can be strings or byte arrays.
- **PeekMessages**: Allows viewing the next message(s) without dequeuing them; useful for checking queue contents or debugging.
- **UpdateMessage**: Changes the content of a message already in the queue, extending its visibility timeout to give more time for processing. This is particularly useful for long-running tasks or to update the status of a task.

Remember to handle exceptions appropriately in real applications and to manage message visibility for processing tasks efficiently.

Managing Queues and Messages in Azure Queue Storage with .NET

Dequeue the Next Message:

```
using Azure.Storage.Queues;
using System.Configuration;

// Retrieve the connection string from app settings
string connectionString =
ConfigurationManager.AppSettings["StorageConnectionString"];
string queueName = "myqueue";

// Create a QueueClient
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    // Get the next message
    QueueMessage[] retrievedMessages =
queueClient.ReceiveMessages().Value.ToArray();

    if (retrievedMessages.Length > 0)
```

```
    {
        // Process the message (must be done in less than 30 seconds by default)
        Console.WriteLine($"Dequeued message: '{retrievedMessages[0].Body}'");

        // Delete the message from the queue after processing
        queueClient.DeleteMessage(retrievedMessages[0].MessageId,
retrievedMessages[0].PopReceipt);
    }
}
```

Get the Queue Length:

```
using Azure.Storage.Queues;

// Assuming 'connectionString' and 'queueName' are defined
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    QueueProperties properties = queueClient.GetProperties();

    // Retrieve the cached approximate message count
    int cachedMessagesCount = properties.ApproximateMessagesCount;

    // Display number of messages
    Console.WriteLine($"Number of messages in queue: {cachedMessagesCount}");
}
```

Delete a Queue:

```
using Azure.Storage.Queues;
using System.Configuration;

// Retrieve the connection string from app settings
string connectionString =
ConfigurationManager.AppSettings["StorageConnectionString"];
string queueName = "myqueue";

// Create a QueueClient
QueueClient queueClient = new QueueClient(connectionString, queueName);

if (queueClient.Exists())
{
    // Delete the queue and all its messages
    queueClient.Delete();
}
```

Notes:

- **Dequeueing Messages**:
  - Involves two steps: receiving (which makes the message invisible) and deleting (to remove it permanently). This ensures fault tolerance by allowing for message reprocessing if processing fails.
  - Default invisibility timeout is 30 seconds, which can be adjusted.
- **Queue Length**:
  - `ApproximateMessagesCount` gives an estimate which might be higher than the actual count due to the nature of how messages are counted in Azure Queue Storage.
- **Queue Deletion**:
  - Deletes the entire queue and all messages within it, which is an irreversible action. Use with caution.

These operations are crucial for managing message queues in a distributed system, ensuring messages are processed only once and providing insights into queue states.

AZ-204: Troubleshoot Solutions by Using Application Insights

Overview:

- **Focus**: Learn how to use **Application Insights** for monitoring application performance and troubleshooting issues.
- **Service**: Part of **Azure Monitor**.

Key Concepts:

- **Instrumentation**: Setting up your applications to send telemetry data to Application Insights.
- **Monitoring**: Tracking performance metrics, usage patterns, and diagnosing problems in real-time or retrospectively.

Benefits:

- **Performance Insights**: Understand how your application performs under various loads or conditions.
- **Troubleshooting**: Quickly identify and resolve application issues with detailed diagnostic data.

Prerequisites:

- **Experience**: At least one year of experience in developing scalable solutions across all software development phases.
- **Azure Knowledge**:
  - Basic understanding of Azure, cloud concepts, services, and familiarity with the Azure portal.
- **Recommended for Beginners**:
  - If you're new to Azure or cloud computing, it's wise to start with **AZ-900: Azure Fundamentals** course.

Code Example:

Here's an example of how to instrument an ASP.NET Core application to use Application Insights:

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;

// This would typically go in your Startup.cs or Program.cs for ASP.NET Core
applications

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add Application Insights telemetry with your Instrumentation Key
        services.AddApplicationInsightsTelemetry("Your-Instrumentation-Key-Here");

        // Other service configurations...
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        // Middleware for handling requests, etc.
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });

        // Example of manual tracking - this would be used to track custom events
or metrics
        var telemetryClient = new TelemetryClient();
        telemetryClient.TrackEvent("Application Started");
    }
}
```

- Replace `"Your-Instrumentation-Key-Here"` with your actual Application Insights instrumentation key.

This example shows how to add Application Insights telemetry to an ASP.NET Core service. Once set up, Application Insights will automatically collect various data points like request rates, response times, and exception details, with the option to manually track custom events or metrics for deeper insights into your application's behavior.

Introduction to Application Insights

Overview:

- **Purpose**: Instrumenting and monitoring applications to ensure maximum availability and performance.

Learning Outcomes:

After completing this module, you will be able to:

- **Describe Application Insights**: Understand the functionality of Application Insights, including how it collects events and metrics from your applications.
- **Instrument Applications**:
  - Learn how to set up your app to use Application Insights for monitoring.
  - **Availability Tests**: Configure and run tests to check if your app is accessible and responds correctly.
- **Use Application Map**:
  - Utilize the Application Map feature to:
    - Monitor application performance across various components.
    - Troubleshoot issues by visualizing dependencies and interactions between application parts.

Code Example:

To give you an idea of how to instrument an application, here's a simple example for an ASP.NET Core application:

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add Application Insights telemetry with your Instrumentation Key
        services.AddApplicationInsightsTelemetry("Your-Instrumentation-Key-Here");

        // Other service configurations...
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        // Middleware for handling requests, etc.
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });

        // Example of tracking an event manually
        var telemetryClient = new TelemetryClient();
        telemetryClient.TrackEvent("Application Started");
```

```
    }
}
```

- Replace `"Your-Instrumentation-Key-Here"` with the actual Application Insights key.

This setup integrates Application Insights into your ASP.NET Core application, allowing automatic collection of telemetry data for monitoring. The `TrackEvent` method demonstrates how to manually log custom events which can be useful for tracking specific milestones or user actions within your application.

Explore Application Insights

Overview:

- **Application Insights**: An extension of Azure Monitor offering Application Performance Monitoring (APM) features.
- **Purpose**: Monitor applications throughout their lifecycle from development to production.

Key Uses:

- **Proactive Monitoring**: Understand application performance in real-time.
- **Reactive Analysis**: Review data to troubleshoot incidents post-occurrence.

Capabilities:

- **Metrics and Telemetry**: Collects data on application activities and health.
- **Trace Logging**: Stores detailed application logs which can be linked with other telemetry for comprehensive activity views. Adding trace logging is straightforward, often just needing a destination setup.

Feature Overview:

Feature	Description
Live Metrics	Real-time observation of application activity without impacting the host environment.
Availability	Synthetic Transaction Monitoring to check endpoint availability and responsiveness over time.
GitHub or Azure DevOps Integration	Automatically create work items based on Application Insights data for better issue tracking and resolution.
Usage	Analyze user interactions to see which features are popular, aiding in UX/UI improvements.
Smart Detection	Proactively detects failures and anomalies by analyzing telemetry, reducing the need for manual monitoring.
Application Map	Provides a visual, top-down view of your application's architecture, highlighting component health and performance.

Feature	Description
Distributed Tracing	Enables tracing of transactions across services, offering an end-to-end view of request processing for complex systems.

Code Example:

Here's a basic example of how to implement trace logging with Application Insights in a .NET application:

```
using Microsoft.ApplicationInsights;
using Microsoft.Extensions.Logging;

public class MyService
{
    private readonly ILogger<MyService> _logger;

    public MyService(ILogger<MyService> logger)
    {
        _logger = logger;
    }

    public void DoSomethingImportant()
    {
        try
        {
            // Simulating some work
            _logger.LogInformation("Starting an important operation.");
            // ... some code here ...
            _logger.LogInformation("Important operation completed successfully.");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while performing an important
operation.");
        }
    }
}
```

- Ensure Application Insights is configured in your application, usually done in the `Startup` class or `Program.cs` for newer .NET versions:

```
services.AddApplicationInsightsTelemetry("Your-Instrumentation-Key-Here");
```

This setup allows automatic logging to Application Insights, where you can then leverage all the features mentioned for monitoring and diagnostics. Remember to replace `"Your-Instrumentation-Key-Here"` with your actual key.

What Application Insights Monitors

- **Metrics and Telemetry**:
  - **Request Metrics**: Rates, response times, and failure rates to identify popular pages, user locations, and response performance.
  - **Dependency Metrics**: Rates, times, and failure rates to detect if external services are affecting your app's performance.
  - **Exceptions**: Aggregated stats or specific instances with stack traces for both server and browser errors.
  - **Page Views & Load Performance**: Data on how quickly pages load from users' browsers.
  - **AJAX Calls**: Metrics on rates, times, and failures from web pages.
  - **User & Session Counts**: Track user engagement and session data.
  - **Performance Counters**: CPU, memory, network usage from Windows or Linux servers.
  - **Host Diagnostics**: From Docker or Azure environments.
  - **Trace Logs**: Correlate application trace events with requests for detailed diagnostics.
  - **Custom Events & Metrics**: User-defined metrics to track business-specific events.

Getting Started with Application Insights

- **Service Overview**:
  - Part of Microsoft Azure, collects telemetry for analysis.
  - **Pricing**: Free to start with, basic plan incurs no charge until substantial usage.
- **Ways to Implement Monitoring**:
  1. **Run Time Instrumentation**:
     - **Method**: Instrument your web app on the server without code changes.
     - **Use Case**: Best for already deployed applications.

```
// Example in ASP.NET Core to add Application Insights at runtime
services.AddApplicationInsightsTelemetry("Your-Instrumentation-Key-
Here");
```

  2. **Development Time Instrumentation**:
     - **Method**: Integrate Application Insights SDK into your code for custom telemetry.
     - **Benefit**: More control over what telemetry is collected.

```
// Example in ASP.NET Core during development
using Microsoft.ApplicationInsights;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddApplicationInsightsTelemetry("Your-Instrumentation-
Key-Here");
```

```
    }
}
```

3. **Web Page Instrumentation**:
    - **Purpose**: Collect client-side data like page views, AJAX calls, etc.

```
<!-- Add this script to your HTML for client-side telemetry -->
<script type="text/javascript">
    var appInsights=window.appInsights||function(a){
        function b(a){c[a]=function(){var
b=arguments;c.queue.push(function(){c[a].apply(c,b)})}}var c=
{config:a},d=document,e=window;setTimeout(function(){var
b=d.createElement("script");b.src=a.url||"https://az416426.vo.msecnd.ne
t/scripts/a/ai.0.js",d.getElementsByTagName("script")
[0].parentNode.appendChild(b)});try{c.cookie=d.cookie}catch(a)
{}c.queue=[];for(var f=
["Event","Exception","Metric","PageView","Trace","Dependency"];f.length
;)b("track"+f.pop());if(b("setAuthenticatedUserContext"),b("clearAuthen
ticatedUserContext"),b("startTrackEvent"),b("stopTrackEvent"),b("startT
rackPage"),b("stopTrackPage"),b("flush"),!a.disableExceptionTracking)
{f="onerror",b("_"+f);var g=e[f];e[f]=function(a,b,d,e,h){var
i=g&&g(a,b,d,e,h);return!0!==i&&c["_"+f](a,b,d,e,h),i}}return c
    }({
        instrumentationKey:"Your-Instrumentation-Key-Here"
    });


window.appInsights=appInsights,appInsights.queue&&0===appInsights.queue
.length&&appInsights.trackPageView();
</script>
```

4. **Mobile App Integration**:
    - **Method**: Use Visual Studio App Center for mobile telemetry.
5. **Availability Tests**:
    - **Function**: Regularly ping your website to check its availability.

Remember to replace `"Your-Instrumentation-Key-Here"` with your actual Application Insights key in these examples.

Discover Log-Based Metrics

Overview of Metrics in Application Insights:

- **Two Types of Metrics**:
    1. **Log-Based Metrics**:
        - Derived from event logs stored in Application Insights.

        - Translated into Kusto queries for analysis.
    2. **Standard Metrics**:
        - Pre-aggregated time series data, offering better query performance.
        - Ideal for dashboards and real-time alerting.

Log-Based Metrics Details:

- **Event Collection**:
    - **Manual**: Through SDK methods explicitly called in your code.
    - **Automatic**: Via autoinstrumentation within the SDK.
- **Storage and Analysis**:
    - Events are stored as logs in the Application Insights backend.
    - The Azure portal's Application Insights blades serve as tools for visualization and diagnostics.
- **Benefits**:
    - **Detailed Analysis**: Provides comprehensive data, allowing for precise metrics like exact request counts by URL or user sessions with detailed traces.
    - **Diagnostic Value**: Enhances visibility into app health and usage, reducing diagnostic time for issues.
- **Challenges**:
    - **Volume Management**: High telemetry volume can make it impractical to store all events:
        - **Sampling and Filtering**: Techniques used to manage telemetry volume which might reduce metric accuracy due to less data for aggregation.

Usage Recommendations:

- **Metric Selection**:
    - Use log-based metrics for in-depth analysis where you need more dimensions.
    - Prefer standard metrics for scenarios requiring real-time or near-real-time insights and dashboarding due to their pre-aggregated nature.
- **Switching Metrics**: Use the namespace selector in the Metrics Explorer of Azure portal to toggle between log-based and standard metrics for different use cases.

Code Example:

Here's an example of how to manually send custom events to Application Insights for log-based metrics:

```
using Microsoft.ApplicationInsights;

public class MyService
{
    private readonly TelemetryClient _telemetryClient;
```

```
    public MyService()
    {
        _telemetryClient = new TelemetryClient();
    }

    public void PerformCriticalOperation()
    {
        try
        {
            // Some critical operation
            _telemetryClient.TrackEvent("CriticalOperationStarted");
            // Operation logic here
            _telemetryClient.TrackEvent("CriticalOperationCompleted");
        }
        catch (Exception ex)
        {
            _telemetryClient.TrackException(ex);
        }
    }
}
```

- This example shows how to manually track events and exceptions, which will then be stored as logs and can be analyzed using log-based metrics. Remember, for automatic collection, you would configure the SDK in your application's startup or configuration phase.

Preaggregated Metrics

Overview:

- **Storage**: Preaggregated metrics are stored as time series with key dimensions rather than individual events, enhancing query performance.
- **Benefits**: Faster data retrieval, less compute power needed, supports near real-time alerting, more responsive dashboards.

Distinction in Application Insights:

- **Naming**:
  - **Standard Metrics (preview)**: Refers to preaggregated metrics.
  - **Log-based Metrics**: Refers to traditional metrics derived from event logs.

SDK and Collection:

- **Newer SDKs (Application Insights 2.7 or later for .NET)**:
  - **Preaggregation**: Metrics are preaggregated during collection, ensuring accuracy despite sampling or filtering.
  - **Custom Metrics**: Using `GetMetric` for custom metrics results in less data ingestion and lower costs.

```
// Example using newer SDK for custom metric preaggregation
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;

public class MyService
{
    private readonly TelemetryClient _telemetryClient;
    private readonly Metric _customMetric;

    public MyService()
    {
        _telemetryClient = new TelemetryClient();
        // Create a metric for aggregation
        _customMetric = _telemetryClient.GetMetric("MyCustomMetric", "Dimension1",
"Dimension2");
    }

    public void DoWork()
    {
        _customMetric.TrackValue(42); // Track a value for aggregation
    }
}
```

- **Older SDKs**:
  - While they don't preaggregate during collection, the Application Insights backend still aggregates events for these metrics.
  - **Performance**: Users can leverage preaggregated metrics for better query performance even with older SDKs, though without the immediate reduction in data volume transmitted.

Ingestion Sampling:

- **Preaggregation Before Sampling**: Events are preaggregated at the collection endpoint before ingestion sampling occurs, ensuring that sampling does not affect the accuracy of preaggregated metrics.

Key Points:

- Preaggregated metrics provide a performance advantage in querying and visualization, particularly for real-time scenarios.
- The distinction between standard and log-based metrics helps in choosing the appropriate metric type for different analytical needs and performance requirements.

This setup in Application Insights aims to balance detailed diagnostics with efficient data handling and analysis, catering to both real-time and historical data needs.

Instrument an App for Monitoring

Instrumentation Methods:

1. **Automatic Instrumentation (Autoinstrumentation)**:
    - **Description**: Enables telemetry collection through configuration without code changes.
    - **Pros**: Convenient, no code modification needed.
    - **Cons**: Less configurable, not supported by all languages/platforms.
    - **Use When**: It's the easiest way to start monitoring with Application Insights, especially when available.
2. **Manual Instrumentation**:
    - **Description**: Involves coding with Application Insights or OpenTelemetry APIs by installing language-specific SDKs.
    - **Pros**: Offers more control, allows for custom telemetry.
    - **Cons**: Requires managing SDK updates, more effort.
    - **Options**:
        - **Application Insights SDKs**: For fine-grained control and custom telemetry.
        - **Azure Monitor OpenTelemetry Distros**: For a more standardized, future-proof approach.

Using Application Insights SDKs:

- **When to Use SDKs**:
    - Need for custom events and metrics.
    - Requirement for controlling telemetry flow.
    - Autoinstrumentation isn't an option due to language or platform constraints.
- **Process**:
    - **Installation**: Add the Application Insights SDK to your application.
    - **Instrumentation**: Cover the web app, background services, and client-side JavaScript.
    - **Hosting**: No need to host in Azure; telemetry is sent to an Application Insights resource using a unique token.

Code Example for Manual Instrumentation with Application Insights SDK in .NET:

```
using Microsoft.ApplicationInsights;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Install the Application Insights SDK
        services.AddApplicationInsightsTelemetry("Your-Instrumentation-Key-Here");
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        // Middleware for handling requests, etc.
        app.UseRouting();

        app.UseEndpoints(endpoints =>
```

```
        {
            endpoints.MapControllers();
        });

        // Example of manual tracking
        var telemetryClient = new TelemetryClient();
        telemetryClient.TrackEvent("Application Started");
        telemetryClient.TrackMetric("UserCount", 1); // Example of custom metric
    }
}
```

- Replace `"Your-Instrumentation-Key-Here"` with your actual Application Insights key.

Resources:

- **SDK Versions**: Check GitHub for the latest SDK versions and names.

This approach allows for detailed monitoring tailored to your application's specific needs, giving you control over what data is collected and how it's used for analysis and troubleshooting.

Enable via OpenTelemetry

Background:

- **OpenTelemetry**: Result of merging OpenCensus and OpenTracing, supported by Microsoft and major cloud/APM vendors, now under the Cloud Native Computing Foundation (CNCF).
- **Microsoft's Role**: Platinum Member of the CNCF, actively contributing to OpenTelemetry.

Terminology Transition:

Application Insights Term	OpenTelemetry Term	Description
Autocollectors	Instrumentation libraries	Libraries that automatically collect telemetry without code changes.
Channel	Exporter	Component responsible for sending telemetry data to a backend.
Codeless / Agent-based	Autoinstrumentation	The process of enabling telemetry collection without modifying application code.
Traces	Logs	Detailed records of application execution paths or events.
Requests	Server Spans	Represents a server-side operation or request.
Dependencies	Other Span Types (Client, Internal, etc.)	Non-server operations, like calls to external services or database queries.
Operation ID	Trace ID	Unique identifier for a distributed trace that spans across multiple services.

Application Insights Term	OpenTelemetry Term	Description
ID or Operation Parent ID	Span ID	Identifier for a specific operation within a trace, used to denote parent-child relationships.

Using OpenTelemetry for Monitoring:

- **Incorporating OpenTelemetry**:
  - **Instrumentation Libraries**: Use these to collect telemetry data automatically or semi-automatically depending on the setup.
  - **Exporters**: Configure to send your telemetry to Azure Monitor Application Insights or any other compatible backend.

Code Example:

Here's a basic example of how you might enable OpenTelemetry in a .NET application to send data to Application Insights:

```
using OpenTelemetry;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using Azure.Monitor.OpenTelemetry.Exporter;

public class Program
{
    public static void Main(string[] args)
    {
        using var tracerProvider = Sdk.CreateTracerProviderBuilder()
            .AddSource("MyCompany.MyProduct.MyLibrary") // Name of the
instrumentation library
            .SetResourceBuilder(
                ResourceBuilder.CreateDefault()
                    .AddService(serviceName: "MyServiceName", serviceVersion:
"1.0.0")
            )
            .AddAspNetCoreInstrumentation() // This adds automatic instrumentation
for ASP.NET Core
            .AddHttpClientInstrumentation() // For HTTP client calls
            .AddAzureMonitorTraceExporter(o =>
            {
                // Configures the exporter to send data to Application Insights
                o.ConnectionString =
"InstrumentationKey=YOUR_INSTRUMENTATION_KEY_HERE";
            })
            .Build();

        // Your application code here
        // ...
    }
}
```

- Replace `"YOUR_INSTRUMENTATION_KEY_HERE"` with your actual Application Insights key.

This example demonstrates setting up OpenTelemetry in a .NET application to trace operations, including ASP.NET Core requests and HTTP client calls, and exporting this data to Azure Monitor using the Azure Monitor OpenTelemetry exporter. Remember, the exact setup might vary depending on the language and framework you're using.

Select an Availability Test

Overview:

- **Purpose**: Monitor the availability and responsiveness of your web app or website post-deployment.
- **Mechanism**: Application Insights sends periodic web requests from various global locations to check your application's health.

Key Points:

- **Limit**: Up to 100 availability tests per Application Insights resource.
- **No Code Changes Required**: Tests work for any HTTP/HTTPS endpoint accessible from the public internet.
- **API Testing**: Can also test the availability of REST APIs your service relies on.

Types of Availability Tests:

1. **Standard Test**:
    - **Functionality**: Checks website availability with a single request, similar to the deprecated URL ping test but with more features.
    - **Features**:
        - Checks endpoint response and performance.
        - Includes TLS/SSL certificate validity check.
        - Supports different HTTP verbs (GET, HEAD, POST).
        - Allows custom headers and data with requests.
2. **Custom TrackAvailability Test**:
    - **Use Case**: For when you need a custom application to perform availability checks.
    - **Implementation**: Use the `TrackAvailability()` method to send test results to Application Insights.

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;

public class CustomAvailabilityTest
{
    private readonly TelemetryClient _telemetryClient;

    public CustomAvailabilityTest()
```

```
    {
        _telemetryClient = new TelemetryClient();
    }

    public void RunTest(string url, bool success)
    {
        var availability = new AvailabilityTelemetry
        {
            Name = "Custom Test",
            Duration = TimeSpan.FromSeconds(1),
            Success = success,
            RunLocation = "CustomLocation",
            Timestamp = DateTimeOffset.UtcNow
        };

        availability.Properties.Add("Url", url);

        _telemetryClient.TrackAvailability(availability);
    }
}
```

3. **URL Ping Test (Classic)**:
    - **Status**: Will be retired by September 30, 2026.
    - **Functionality**: Basic check to see if an endpoint responds, with the ability to measure performance and set custom success criteria.
    - **Action**: Transition to Standard tests before the retirement date to maintain single-step availability testing.

Important Note:

- **Retirement of URL Ping Test**:
    - URL ping tests will be removed from Application Insights resources by September 30, 2026. Review and transition to standard tests to continue availability monitoring.

This setup ensures your application's uptime and responsiveness are monitored effectively, helping in maintaining service quality and user satisfaction.

Troubleshoot App Performance by Using Application Map

Overview:

- **Application Map**: A tool to visualize and troubleshoot performance issues across a distributed application's components.
- **Purpose**: Spot bottlenecks, failure points, or performance issues in a microservices or distributed environment.

Key Features:

- **Components and Dependencies**:

    - **Components**: Represent parts of your application (e.g., microservices, roles) with visibility to code or telemetry.
    - **Dependencies**: External services like SQL or Event Hubs, which might not have direct access to telemetry.

- **Visualization**:

    - **Nodes**: Each node on the map signifies a component or dependency, showing health metrics, KPIs, and alert status.
    - **Navigation**: Click any component for deeper diagnostics or Azure service-specific insights (like SQL Database Advisor).

- **Discovery and Loading**:

    - **Progressive Discovery**: Starts with discovering related components through HTTP dependency calls tracked by the Application Insights SDK.
    - **Update Mechanism**: An "Update map components" button to refresh the map with newly discovered components.
    - **Single Resource**: If all components are under one Application Insights resource, all components are loaded at once without additional discovery.

- **Complexity Management**:

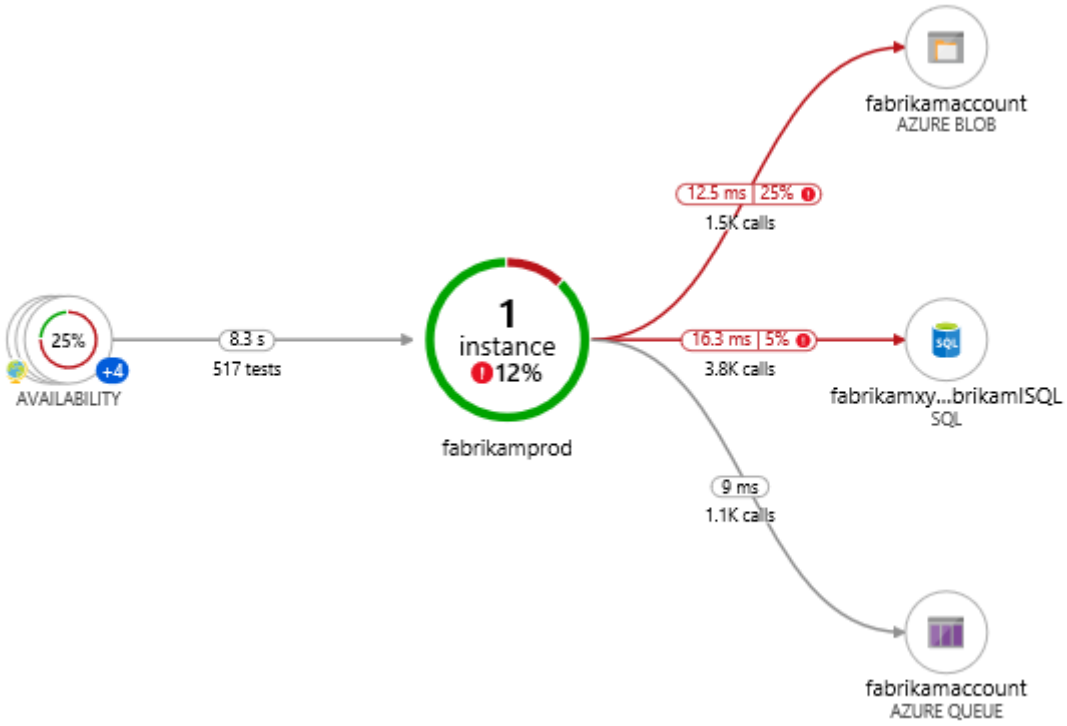
    - Designed to handle complex topologies with numerous components.
    - Performance might take time to load depending on application size.

Usage:

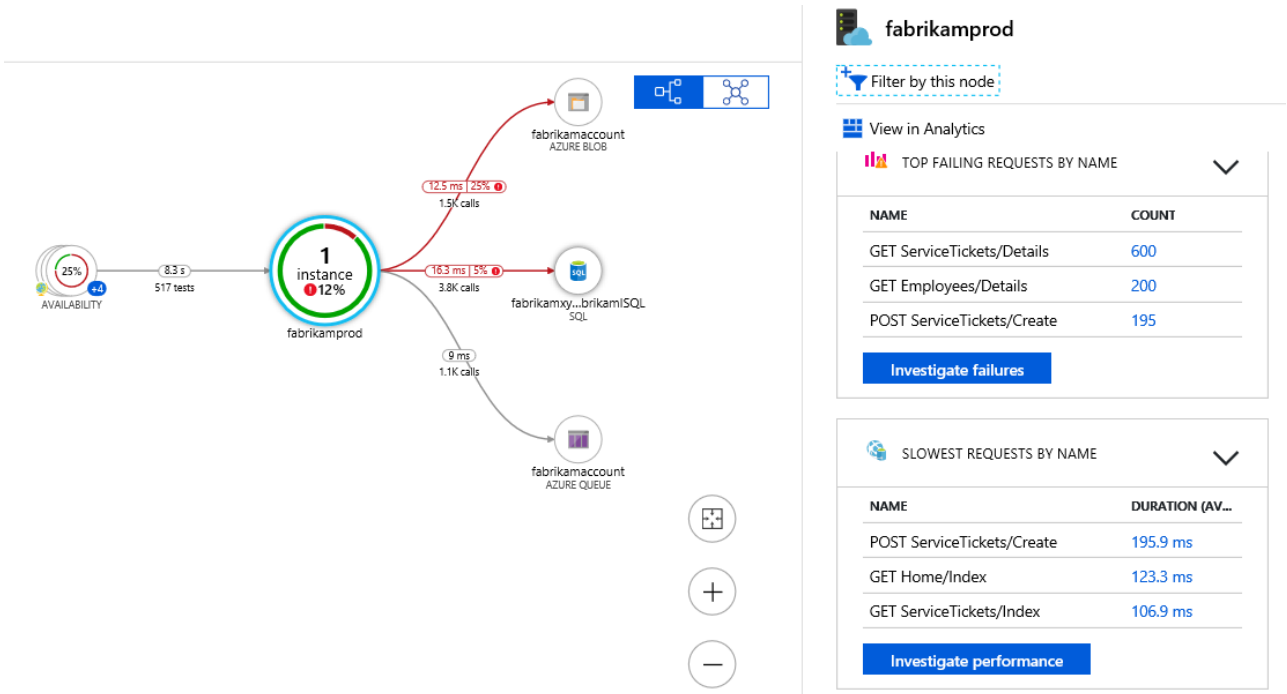
- **Component Identification**: Uses the `cloud role name` property to label components. This can be manually set or overridden for custom mapping.

Screenshots:

- **Initial Load**: Shows all components within a single Application Insights resource.



- **Component Details**: Clicking a component reveals performance metrics or failure data for triage.



Code Example:

Here's how you might set or override the `cloud role name` in C# to ensure correct representation in the Application Map:

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;

public class Startup
```

```
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddApplicationInsightsTelemetry();

        // Configure the TelemetryInitializer
        services.AddSingleton<ITelemetryInitializer>(new
CloudRoleNameInitializer());
    }
}

public class CloudRoleNameInitializer : ITelemetryInitializer
{
    public void Initialize(ITelemetry telemetry)
    {
        // Set or override the cloud role name for this component
        telemetry.Context.Cloud.RoleName = "MyCustomComponentName";
    }
}
```

This ensures that your component is correctly labeled in the Application Map, which can be crucial for complex applications with many services or roles. Remember, this code should be part of your application's configuration or startup logic.

AZ-204: Implement Caching for Solutions

Overview:

- **Focus**: Enhancing application performance and scalability through caching mechanisms.
- **Services**:
  - **Azure Cache for Redis**: For in-memory caching to reduce data access latency.
  - **Azure Content Delivery Network (CDN)**: For distributing content globally to reduce load and improve response times.

Learning Objectives:

- Learn to integrate caching solutions like Azure Cache for Redis to manage data more efficiently.
- Understand how to use Azure CDN to serve static and dynamic content closer to users.

Benefits:

- **Performance**: Improve response times by caching frequently accessed data.
- **Scalability**: Handle increased load without proportional increases in backend computation or database access.

Prerequisites:

- **Experience**: At least one year developing scalable solutions across all software development phases.
- **Azure Knowledge**:
  - Basic understanding of Azure, cloud concepts, services, and familiarity with the Azure portal.

- **Recommended for Beginners**:
  - If you're new to Azure or cloud computing, start with the **AZ-900: Azure Fundamentals** course.

Code Example:

Here's a simple example of how you might use Azure Cache for Redis in a .NET application:

```
using System;
using StackExchange.Redis;

public class RedisCacheService
{
    private readonly ConnectionMultiplexer _redisConnection;
    private readonly IDatabase _database;

    public RedisCacheService(string connectionString)
    {
        _redisConnection = ConnectionMultiplexer.Connect(connectionString);
        _database = _redisConnection.GetDatabase();
    }

    public async Task SetCacheValueAsync(string key, string value)
    {
        await _database.StringSetAsync(key, value);
    }

    public async Task<string> GetCacheValueAsync(string key)
    {
        return await _database.StringGetAsync(key);
    }
}

// Usage in your application
public class Program
{
    public static async Task Main()
    {
        string redisConnectionString = "your-redis-connection-string-here";
        var cacheService = new RedisCacheService(redisConnectionString);

        await cacheService.SetCacheValueAsync("testKey", "Hello, Redis!");

        var cachedValue = await cacheService.GetCacheValueAsync("testKey");
        Console.WriteLine(cachedValue); // Output: Hello, Redis!
    }
}
```

- Replace `"your-redis-connection-string-here"` with your actual Azure Cache for Redis connection string.

This example demonstrates basic operations like setting and retrieving values from Redis, which can significantly speed up data access in your applications by reducing database calls for frequently accessed data.

Develop for Azure Cache for Redis

Overview:

- **Objective**: Learn to leverage Azure Cache for Redis to speed up data retrieval in applications.
- **Key Concepts**:
  - **Configuration**: Setting up Azure Cache for Redis.
  - **Interaction**: How to read from and write to the cache.
  - **Integration**: Connecting your .NET applications to Azure Cache for Redis.

Key Learning Points:

- **Configuring Azure Cache for Redis**:
  - Understand how to create and configure a Redis cache in Azure, including choosing the right tier (Basic, Standard, or Premium) based on your application needs.
- **Interacting with Redis Cache**:
  - Learn about basic Redis commands for setting, getting, and managing data in the cache.
  - Explore how to implement caching strategies like lazy loading, write-through, or read-through caching.
- **Connecting .NET Applications**:
  - Use the StackExchange.Redis client library to connect to and interact with Azure Cache for Redis from .NET applications.

Code Example:

Here's a basic example of how to connect to Azure Cache for Redis and perform some operations using StackExchange.Redis in a .NET application:

```
using System;
using System.Threading.Tasks;
using StackExchange.Redis;

public class RedisService
{
    private readonly ConnectionMultiplexer _connection;
    private readonly IDatabase _database;

    public RedisService(string connectionString)
    {
        _connection = ConnectionMultiplexer.Connect(connectionString);
        _database = _connection.GetDatabase();
```

```
    }

    public async Task SetStringAsync(string key, string value)
    {
        await _database.StringSetAsync(key, value);
    }

    public async Task<string> GetStringAsync(string key)
    {
        return await _database.StringGetAsync(key);
    }

    public async Task<bool> KeyExistsAsync(string key)
    {
        return await _database.KeyExistsAsync(key);
    }

    public async Task DeleteKeyAsync(string key)
    {
        await _database.KeyDeleteAsync(key);
    }
}

class Program
{
    static async Task Main(string[] args)
    {
        string redisConnectionString = "your-redis-connection-string-here";

        var redisService = new RedisService(redisConnectionString);

        // Set a value
        await redisService.SetStringAsync("exampleKey", "Hello, Redis!");

        // Get the value
        string value = await redisService.GetStringAsync("exampleKey");
        Console.WriteLine("Retrieved value: " + value);

        // Check if key exists
        bool exists = await redisService.KeyExistsAsync("exampleKey");
        Console.WriteLine("Key exists: " + exists);

        // Delete the key
        await redisService.DeleteKeyAsync("exampleKey");
    }
}
```

- Replace `"your-redis-connection-string-here"` with your actual Azure Cache for Redis connection string.

This example shows how to perform basic operations like setting, retrieving, checking for existence, and deleting data in Redis from a .NET application. Remember, in a real-world scenario, you'd also handle errors,

manage connection lifetimes, and perhaps implement more sophisticated caching strategies.

Introduction to Azure Cache for Redis

Overview:

- **Caching**: A technique for enhancing performance and scalability by storing data in fast, local storage.
- **Benefits**:
  - **Faster Response Times**: By keeping data closer to the application, response times are significantly reduced.
  - **Scalability**: Helps manage increased load by offloading the database.

Learning Outcomes:

After completing this module, you will be able to:

- **Understand Scenarios for Azure Cache for Redis**:
  - Learn about different use cases like session state management, caching database query results, or speeding up data retrieval for web applications.
  - **Service Tiers**:
    - **Basic**: Simple caching for development/testing.
    - **Standard**: Offers 99.9% SLA with replication for improved reliability.
    - **Premium**: Includes all Standard tier features plus advanced capabilities like clustering, Redis persistence, and virtual network support.
- **Configure and Interact with Azure Cache for Redis**:
  - Identify necessary parameters like size, region, and tier when creating an instance.
  - Understand how to use Redis commands or client libraries to manage data in the cache.
- **Integrate .NET Applications with Azure Cache for Redis**:
  - Use the StackExchange.Redis library to connect and interact with the cache from .NET applications.

Code Example:

Here's a simple example of how to connect a .NET application to Azure Cache for Redis:

```
using System;
using StackExchange.Redis;

class Program
{
    static async Task Main(string[] args)
    {
        // Connection string to your Azure Cache for Redis instance
        // Format: <redis-server-name>.redis.cache.windows.net:6380,password=
<access-key>,ssl=True,abortConnect=False
```

```
        string connectionString = "your-redis-connection-string-here";

        // Connect to Redis
        ConnectionMultiplexer redis = await
ConnectionMultiplexer.ConnectAsync(connectionString);
        IDatabase cache = redis.GetDatabase();

        // Store a value
        await cache.StringSetAsync("key1", "Hello, Redis!");

        // Retrieve the value
        string value = await cache.StringGetAsync("key1");
        Console.WriteLine($"Retrieved value: {value}");

        // Close the connection
        redis.Close();
    }
}
```

- Replace `"your-redis-connection-string-here"` with your actual Redis connection string.

This example demonstrates how to set up a basic connection, store data, and retrieve it from Azure Cache for Redis using C# and the StackExchange.Redis library. Remember, in a production environment, you'd want to handle connections more robustly, manage exceptions, and possibly use dependency injection for better testability and maintainability.

Explore Azure Cache for Redis

Overview:

- **Azure Cache for Redis**: An in-memory data store based on Redis, aimed at boosting application performance and scalability.
- **Functionality**: Stores frequently accessed data in server memory for quick read/write operations, reducing latency and increasing throughput.

Service Offerings:

- **Open Source Redis (OSS Redis)** and **Redis Enterprise** from Redis Labs as managed services.
- **Features**: Secure, dedicated Redis instances with full Redis API compatibility, managed by Microsoft, accessible from within or outside Azure.

Key Scenarios for Azure Cache for Redis:

Pattern	Description
Data Cache	Implements cache-aside pattern for on-demand data loading from databases. Updates to data can also update the cache for consistency.
Content Cache	Caches static content like headers or banners, speeding up access compared to database retrieval.

Pattern	Description
Session Store	Manages session data (e.g., shopping carts) associated with user cookies, reducing the performance hit from large cookie sizes.
Job and Message Queuing	Queue tasks for later processing, enhancing application responsiveness by deferring long-running tasks.
Distributed Transactions	Supports batch execution of commands as atomic operations, ensuring data integrity across multiple commands.

Code Example:

Here's how you might implement the **Data Cache** pattern for caching database results in .NET using Azure Cache for Redis:

```
using System;
using System.Threading.Tasks;
using StackExchange.Redis;

public class DataCacheService
{
    private readonly IDatabase _cache;

    public DataCacheService(string connectionString)
    {
        var redis = ConnectionMultiplexer.Connect(connectionString);
        _cache = redis.GetDatabase();
    }

    // Fetch from cache or database if not in cache
    public async Task<string> GetDataAsync(string key)
    {
        var cachedData = await _cache.StringGetAsync(key);
        if (!cachedData.IsNullOrEmpty)
        {
            return cachedData;
        }

        // Simulating database fetch
        string dataFromDb = await FetchFromDatabaseAsync(key);
        await _cache.StringSetAsync(key, dataFromDb, TimeSpan.FromMinutes(30)); //
Cache for 30 minutes
        return dataFromDb;
    }

    // Update both database and cache
    public async Task UpdateDataAsync(string key, string data)
    {
        await UpdateInDatabaseAsync(key, data);
        await _cache.StringSetAsync(key, data, TimeSpan.FromMinutes(30)); // Cache
```

```
for 30 minutes
    }

    // Placeholder methods for actual database operations
    private Task<string> FetchFromDatabaseAsync(string key) =>
Task.FromResult("Database result for " + key);
    private Task UpdateInDatabaseAsync(string key, string data) =>
Task.CompletedTask;
}
```

- Replace the placeholder methods `FetchFromDatabaseAsync` and `UpdateInDatabaseAsync` with actual database operations.

This example demonstrates basic cache-aside pattern implementation where data is either fetched from the cache or the database, with updates reflected in both places to maintain data consistency. Remember, in a real-world scenario, you'd handle more complex scenarios like cache invalidation, error handling, and possibly implement more sophisticated caching strategies.

Service Tiers of Azure Cache for Redis

Overview:

- Azure Cache for Redis offers several tiers to cater to different needs, from development to high-performance production scenarios.

Detailed Description of Tiers:

Tier	Description
Basic	- **Infrastructure**: Runs on a single VM with open-source Redis (OSS Redis). - **SLA**: No service-level agreement. - **Use Case**: Ideal for development, testing, or non-critical workloads where high availability isn't necessary.
Standard	- **Infrastructure**: Utilizes two VMs in a replicated setup for high availability. - **SLA**: Provides a service-level agreement for 99.9% availability. - **Use Case**: Suitable for production environments where data redundancy is needed.
Premium	- **Infrastructure**: Deployed on high-performance VMs with OSS Redis. - **Features**: Higher throughput, lower latency, enhanced availability. - **Additional**: Includes clustering, persistence, and virtual network support. - **Use Case**: For applications requiring advanced Redis features and performance.
Enterprise	- **Infrastructure**: Powered by Redis Enterprise software from Redis Labs. - **Features**: Supports Redis modules like RediSearch, RedisBloom, and RedisTimeSeries for advanced data handling. - **Availability**: Offers better availability than Premium. - **Use Case**: For mission-critical applications needing specialized Redis capabilities.

Tier	Description
Enterprise Flash	- **Infrastructure**: Also uses Redis Enterprise software but extends to nonvolatile memory. - **Cost**: More cost-effective due to the use of less expensive memory types. - **Use Case**: When you need large cache sizes at a reduced cost per GB.

Code Example:

While there isn't direct code to implement a specific tier, here's how you might choose a tier when creating a Redis cache using Azure's management SDK in C#:

```
using Microsoft.Azure.Management.Redis;
using Microsoft.Azure.Management.Redis.Models;
using Microsoft.Rest;

// Assume you have already authenticated and have an instance of
RedisManagementClient
var redisClient = new RedisManagementClient(credential) { SubscriptionId = "your-
subscription-id" };

// Define parameters for the Redis Cache creation
var parameters = new RedisCreateParameters
{
    Location = "East US", // Choose an Azure region
    Sku = new Sku
    {
        Name = SkuName.Premium, // Here you choose the tier, e.g., Basic,
Standard, Premium, etc.
        Family = SkuFamily.P, // 'P' for Premium, 'C' for Basic/Standard
        Capacity = 1 // Size of cache, larger numbers for bigger caches
    },
    Properties = new RedisCreateProperties
    {
        EnableNonSslPort = false, // For security, disable non-SSL port
        RedisConfiguration = new RedisConfiguration()
        {
            { "maxmemory-policy", "volatile-lru" } // Example configuration
        }
    }
};

// Create the Redis Cache
var cache = await redisClient.Redis.CreateAsync(
    resourceGroupName: "your-resource-group",
    name: "your-cache-name",
    parameters: parameters);
```

- Replace placeholders like `"your-subscription-id"`, `"your-resource-group"`, and `"your-cache-name"` with actual values.

This example shows how to specify the tier when creating an Azure Cache for Redis instance. Note, the exact tier selection (`SkuName`) will affect the capabilities and pricing of your cache. Remember to handle exceptions and add appropriate error handling in production code.

Configure Azure Cache for Redis

Creation Methods:

- **Tools**:
  - Azure Portal
  - Azure CLI
  - Azure PowerShell

Key Configuration Parameters:

1. **Name**:
   - **Uniqueness**: Must be globally unique within Azure.
   - **Format**:
     - Between 1 and 63 characters.
     - Can include numbers, letters, and hyphens ('-').
     - Cannot start or end with a hyphen, and no consecutive hyphens allowed.
2. **Location**:
   - **Proximity**: Place the cache in the same Azure region as your application to minimize latency and maximize reliability.
   - **Non-Azure**: If connecting from outside Azure, choose an Azure region close to your application's location.
3. **Cache Type (Tier)**:
   - **Selection**: Determines size, performance, and available features.
   - **Tiers**:
     - **Basic**: No SLA, single node, for development/testing.
     - **Standard**: Includes SLA, uses replication for reliability.
     - **Premium**: High performance, includes clustering, persistence, and VNet support.
     - **Enterprise**: Supports Redis modules, higher availability.
     - **Enterprise Flash**: Cost-effective with nonvolatile memory.
   - **Microsoft Recommendation**: Use Standard tier or higher for production.
4. **Clustering Support**:
   - **Availability**: In Premium, Enterprise, and Enterprise Flash tiers.
   - **Functionality**: Automatically splits dataset across multiple nodes (shards).
   - **Configuration**: Specify up to 10 shards; cost scales with the number of shards.

Code Example:

Here's how you might create an Azure Cache for Redis instance using Azure CLI:

```
az redis create \
  --name my-redis-cache \
  --resource-group myResourceGroup \
  --location eastus \
  --sku Standard \
  --vm-size C1 \
  --redis-version 6 \
  --enable-non-ssl-port false
```

- **Explanation**:
  - `--name`: The name of your Redis cache.
  - `--resource-group`: The resource group to contain the cache.
  - `--location`: Azure region for the cache.
  - `--sku`: The tier of the cache (Basic, Standard, Premium, etc.).
  - `--vm-size`: Size of the cache VM; C1 corresponds to a cache size in the Standard tier.
  - `--redis-version`: Specifies the Redis version.
  - `--enable-non-ssl-port`: Set to false to enhance security by disabling non-SSL port.

This command creates a Standard tier cache in the East US region. Remember, for clustering, you would need to enable it in the Premium tier or higher, which involves additional configuration steps not shown here.

Accessing the Redis Instance

Command-Line Tool for Redis:

- **Windows**: Download Redis command-line tools for Windows.
- **Other Platforms**: Download from Redis.io.

Common Redis Commands:

Command	Description
`ping`	Checks the server connection; returns "PONG".
`set [key] [value]`	Sets a key/value pair in the cache; returns "OK".
`get [key]`	Retrieves a value by its key.
`exists [key]`	Checks if a key exists; returns '1' for yes, '0' for no.
`type [key]`	Returns the data type of the value stored at the key.
`incr [key]`	Increments the number stored at key by 1; returns the new value.
`incrby [key] [amount]`	Increments the number stored at key by the specified amount; returns the new value.
`del [key]`	Deletes the key; returns the number of keys deleted.
`flushdb`	Clears the current database.

Example Commands:

```
> set somekey somevalue
OK
> get somekey
"somevalue"
> exists somekey
(string) 1
> del somekey
(string) 1
> exists somekey
(string) 0
```

Adding Expiration Time to Values:

- **TTL (Time To Live)**: Used to set an expiration time for keys.
- **Precision**: Can be set in seconds or milliseconds.
- **Resolution**: 1 millisecond.
- **Behavior**: Keys are automatically removed when TTL expires, similar to using `DEL`.

Example of Setting TTL:

```
> set counter 100
OK
> expire counter 5
(integer) 1
> get counter
100
... wait ...
> get counter
(nil)
```

Connecting to Azure Cache for Redis from a Client:

- **Required Information**:
  - **Host Name**: The public address, e.g., `sportsresults.redis.cache.windows.net`.
  - **Port**: Usually 6379 for non-SSL or 6380 for SSL (SSL is recommended).
  - **Access Key**: Acts as a password, available in Azure portal under "Access Keys". Two keys (Primary & Secondary) for key rotation.
- **Security**:
  - **Confidentiality**: Treat access keys like passwords; protect them from unauthorized access.
  - **Key Rotation**: Microsoft recommends periodically regenerating keys for security.

Code Example for Connecting with C#:

```
using System;
using StackExchange.Redis;

class Program
{
    static async Task Main(string[] args)
    {
        // Connection string format:
        // <redis-server-name>.redis.cache.windows.net:6380,password=<access-
key>,ssl=True,abortConnect=False
        string connectionString = "your-redis-url:6380,password=your-primary-
key,ssl=True,abortConnect=False";

        // Connect to Redis
        ConnectionMultiplexer redis = await
ConnectionMultiplexer.ConnectAsync(connectionString);
        IDatabase cache = redis.GetDatabase();

        // Example operations
        await cache.StringSetAsync("exampleKey", "Hello, Redis!");
        string value = await cache.StringGetAsync("exampleKey");
        Console.WriteLine("Retrieved value: " + value);

        // Close the connection
        redis.Close();
    }
}
```

- Replace `"your-redis-url"` and `"your-primary-key"` with your actual Redis cache details.

This setup allows for basic interaction with Azure Cache for Redis, focusing on setting, retrieving, and managing key-value pairs with expiration. Remember to handle exceptions and manage connections appropriately in production scenarios.

Interact with Azure Cache for Redis by using .NET

Overview:

- **Client Libraries**: Typically used to interact with Redis caches. For .NET, **StackExchange.Redis** is a popular choice, available via NuGet.

Connecting to Redis with StackExchange.Redis:

- **Connection String**: Combines host, port, and access key. Example format:

```
[cache-name].redis.cache.windows.net:6380,password=[password-
here],ssl=True,abortConnect=False
```

- **Connection Parameters**:

    - **ssl**: Ensures encrypted communication.
    - **abortConnect**: Allows connection creation even if the server is temporarily unavailable.

- **Optional Parameters**: Can be added to the connection string for further configuration.

Creating a Connection:

- **ConnectionMultiplexer**: The primary class for managing connections to Redis.

```
using StackExchange.Redis;

// Define the connection string
string connectionString = "[cache-name].redis.cache.windows.net:6380,password=
[password-here],ssl=True,abortConnect=False";

// Create a connection
ConnectionMultiplexer redisConnection =
ConnectionMultiplexer.Connect(connectionString);
```

- **Usage**:
    - **Database Access**: For getting or setting data in the cache.
    - **Pub/Sub**: For real-time messaging (not covered in this module).
    - **Server Maintenance/Monitoring**: For direct interaction with servers.

Code Example:

Here's how you might interact with the cache for basic operations like setting and getting values:

```
using System;
using System.Threading.Tasks;
using StackExchange.Redis;

class Program
{
    static async Task Main(string[] args)
    {
        string connectionString = "[cache-
name].redis.cache.windows.net:6380,password=[password-
here],ssl=True,abortConnect=False";

        // Establish connection
        using var redis = await
ConnectionMultiplexer.ConnectAsync(connectionString);
        IDatabase db = redis.GetDatabase();

        // Set a value
        await db.StringSetAsync("key1", "value1");

        // Get the value
```

```
        string value = await db.StringGetAsync("key1");
        Console.WriteLine("Value from Redis: " + value);

        // Check if key exists
        bool keyExists = await db.KeyExistsAsync("key1");
        Console.WriteLine("Key exists: " + keyExists);

        // Delete the key
        await db.KeyDeleteAsync("key1");
    }
}
```

- Replace `[cache-name]` and `[password-here]` with your actual Redis cache details.

This example demonstrates establishing a connection, setting a value in Redis, retrieving it, checking for its existence, and then deleting it. Remember, in real applications, you'd manage connections more effectively (perhaps with dependency injection or a singleton pattern), handle errors, and consider connection pooling for performance.

Accessing a Redis Database

Database Object:

- **Accessing**: Use `GetDatabase()` method from `ConnectionMultiplexer`.

```
IDatabase db = redisConnection.GetDatabase();
```

- **Note**: The `IDatabase` object is lightweight, no need to store it; focus on keeping `ConnectionMultiplexer` alive.

Interacting with Cache:

- **Methods**: Both synchronous and asynchronous versions available, compatible with `async`/`await`.
- **Basic Operations**:
    - **Set Key/Value**:

```
bool wasSet = db.StringSet("favorite:flavor", "i-love-rocky-road");
```

- **Get Value**:

```
string value = db.StringGet("favorite:flavor");
Console.WriteLine(value); // displays: "i-love-rocky-road"
```

Handling Binary Data:

- **Binary Safe**: Redis keys and values can handle binary data.
- **Implicit Conversions**: `byte[]` can be used directly with `StringSet` and `StringGet`.

```
byte[] key = /* some binary key */;
byte[] value = /* some binary value */;

db.StringSet(key, value);

byte[] retrievedValue = db.StringGet(key);
```

Data Types:

- **RedisKey**: Represents keys with implicit conversions for `string` and `byte[]`.
- **RedisValue**: Represents values, similar to `RedisKey`.

Common Operations:

Method	Description
`CreateBatch`	Creates a group of operations to be sent to the server together, but not processed as a single transaction.
`CreateTransaction`	Initiates a transaction where operations are both sent and processed as a single unit.
`KeyDelete`	Removes the specified key from the cache.
`KeyExists`	Checks if a key exists in the cache, returning `true` or `false`.
`KeyExpire`	Sets an expiration time (TTL) for a key, after which it will be automatically deleted.
`KeyRename`	Renames a key to a new key name.
`KeyTimeToLive`	Returns the remaining time to live (TTL) for a key, or -1 if the key does not exist or has no expiration set.
`KeyType`	Returns the type of the value stored at the given key; possible types include string, list, set, zset (sorted set), and hash.

These methods allow for comprehensive management of data in Redis from a .NET application, providing the capability to handle different data structures and operations efficiently. Remember to use asynchronous methods where possible for better scalability in your applications.

Executing Other Commands

Using `Execute` and `ExecuteAsync`:

- **Methods**: `Execute` and `ExecuteAsync` allow for direct textual commands to the Redis server.
- **Return Type**: These methods return a `RedisResult` object.

```
var result = db.Execute("ping");
Console.WriteLine(result.ToString()); // displays: "PONG"

// Asynchronous example
var clientListResult = await db.ExecuteAsync("client", "list");
Console.WriteLine($"Type = {clientListResult.Resp2Type}\r\nResult =
{clientListResult}");
```

- **Properties of `RedisResult`**:
  - `Resp2Type`: Indicates the type of result (e.g., STRING, INTEGER).
  - `IsNull`: Checks if the result is null.

Storing Complex Values:

- **Serialization**: Convert complex objects to string formats like JSON or XML for storage in Redis.
- **Example with JSON**:

```
// GameStat class definition
public class GameStat
{
    // ... (as per your script)

    public override string ToString()
    {
        // ... (as per your script)
    }
}

// Serialization
var stat = new GameStat("Soccer", new DateTimeOffset(new DateTime(2019, 7, 16)),
"Local Game",
                        new[] { "Team 1", "Team 2" },
                        new[] { ("Team 1", 2), ("Team 2", 1) });

string serializedValue = Newtonsoft.Json.JsonConvert.SerializeObject(stat);
bool added = db.StringSet("event:1950-world-cup", serializedValue);

// Deserialization
var retrievedResult = db.StringGet("event:1950-world-cup");
var deserializedStat = Newtonsoft.Json.JsonConvert.DeserializeObject<GameStat>
(retrievedResult.ToString());
Console.WriteLine(deserializedStat.Sport); // displays "Soccer"
```

Connection Management:

- **Cleanup**: Properly dispose of the `ConnectionMultiplexer` when no longer needed.

```
redisConnection.Dispose();
redisConnection = null;
```

Key Points:

- `Execute` and `ExecuteAsync` provide flexibility to run any Redis command, useful for operations not directly supported by `IDatabase` methods.
- Serializing complex types into JSON or another format allows you to store and retrieve more detailed data structures in Redis, though this adds overhead due to serialization/deserialization.
- Proper resource management, like disposing of connections, is crucial for maintaining application health and preventing resource leaks. Remember, in a production environment, you might use dependency injection or a singleton pattern for managing `ConnectionMultiplexer` instances to avoid repeatedly creating and disposing of connections.

Develop for Storage on CDNs

Overview:

- **Focus**: Understanding and implementing Azure Content Delivery Network (CDN) for content distribution and storage.

Key Learning Outcomes:

1. **Functionality of Azure CDN**:
    - Learn how CDN reduces load times by caching content at various global locations closer to end-users, thereby improving performance and user experience.
2. **Controlling Cache Behavior**:
    - Understand how to manage caching rules, headers, and query strings to control how content is cached and served.
3. **.NET Integration**:
    - Learn how to integrate CDN capabilities into .NET applications, including how to interact with or modify CDN behaviors programmatically.

Important Concepts:

- **Edge Locations**: Points of presence where CDN caches content.
- **Cache Expiration**: Setting rules for how long content stays in the cache before re-fetching from the origin server.
- **Purge**: The process of removing content from CDN nodes to ensure that changes in the origin are reflected quickly.
- **Custom Domains**: Using your own domain names with CDN endpoints for brand consistency.

Code Example:

Here's a basic example of how you might interact with an Azure CDN from a .NET application to manage cache behavior:

```
using Microsoft.Azure.Management.Cdn;
using Microsoft.Rest;
using System;
using System.Threading.Tasks;

class CdnManager
{
    private readonly CdnManagementClient _cdnClient;

    public CdnManager(string clientId, string clientSecret, string tenantId,
string subscriptionId)
    {
        var credentials = new ClientCredential(clientId, clientSecret);
        _cdnClient = new CdnManagementClient(credentials)
        {
            SubscriptionId = subscriptionId
        };
        _cdnClient = new CdnManagementClient(credentials)
        {
            SubscriptionId = subscriptionId
        };
    }

    public async Task PurgeContentAsync(string profileName, string endpointName,
string contentPath)
    {
        try
        {
            await _cdnClient.Endpoints.PurgeContentAsync(
                resourceGroupName: "your-resource-group",
                profileName: profileName,
                endpointName: endpointName,
                contentPaths: new string[] { contentPath });
            Console.WriteLine($"Content at {contentPath} has been purged from
CDN.");
        }
        catch (Exception e)
        {
            Console.WriteLine($"An error occurred while purging content:
{e.Message}");
        }
    }

    // Example to set up cache rules might involve API calls to configure CDN
properties
    // This is more complex and would generally be done via the Azure portal or
through specific API calls
}
```

```
class Program
{
    static async Task Main(string[] args)
    {
        var manager = new CdnManager("client-id", "client-secret", "tenant-id",
"subscription-id");
        await manager.PurgeContentAsync("cdn-profile", "cdn-endpoint",
"/path/to/content");
    }
}
```

- Replace `"client-id"`, `"client-secret"`, `"tenant-id"`, `"subscription-id"`, `"your-resource-group"`, `"cdn-profile"`, `"cdn-endpoint"`, and `"/path/to/content"` with your actual Azure credentials and CDN details.

This example demonstrates how to purge content from a CDN endpoint, which is one aspect of managing cache behavior. Remember, managing CDN cache settings often involves more than just purging; it includes setting up rules for caching, handling custom domains, and potentially using Azure CDN's REST API for more granular control over CDN behavior.

Introduction to Azure Content Delivery Network (CDN)

Overview:

- **CDN Definition**: A network of servers that distribute web content from servers geographically closer to end users, reducing delivery latency.
- **Function**: Stores cached content on edge servers at various Points of Presence (POPs) to optimize content delivery speed.

Learning Outcomes:

After completing this module, you will be able to:

- **Understand Azure CDN**:
    - **Mechanics**: How Azure CDN caches and delivers content to improve performance and user experience.
    - **Benefits**: Minimizing latency, reducing load on origin servers, and enhancing global content accessibility.
- **Manage Caching Behavior**:
    - **Control**: Learn how to configure cache settings, including TTL (Time To Live), and how to invalidate (purge) content to ensure users receive the latest updates.
- **Interact with Azure CDN Using .NET**:
    - **Implementation**: Use the Azure CDN Library for .NET to programmatically manage CDN operations like setting up caching rules or purging content.

Code Example:

Here's a basic example of how you might use the Azure CDN Library for .NET to purge content:

```
using Microsoft.Azure.Management.Cdn.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent.Authentication;
using System;
using System.Threading.Tasks;

class AzureCdnManager
{
    private readonly ICdnManager _cdnManager;

    public AzureCdnManager(string clientId, string clientSecret, string tenantId,
string subscriptionId)
    {
        var credentials = new AzureCredentials(
            new ServicePrincipalLoginInformation
            {
                ClientId = clientId,
                ClientSecret = clientSecret
            },
            tenantId,
            AzureEnvironment.AzureGlobalCloud);

        _cdnManager = CdnManager.Authenticate(credentials, subscriptionId);
    }

    public async Task PurgeContentAsync(string resourceGroupName, string
profileName, string endpointName, string contentPath)
    {
        await _cdnManager.Endpoints.PurgeContentAsync(resourceGroupName,
profileName, endpointName, new string[] { contentPath });
        Console.WriteLine($"Content at {contentPath} has been purged from CDN.");
    }
}

class Program
{
    static async Task Main(string[] args)
    {
        var manager = new AzureCdnManager("client-id", "client-secret", "tenant-
id", "subscription-id");
        await manager.PurgeContentAsync("your-resource-group", "cdn-profile",
"cdn-endpoint", "/path/to/content");
    }
}
```

- Replace `"client-id"`, `"client-secret"`, `"tenant-id"`, `"subscription-id"`, `"your-resource-group"`, `"cdn-profile"`, `"cdn-endpoint"`, and `"/path/to/content"` with your actual Azure

credentials and CDN details.

This example illustrates how you can interact with Azure CDN to manage content, specifically for purging content from the cache to ensure users see the most current version of your assets. Remember, actual implementation might require additional error handling, configuration for different CDN profiles, and possibly managing other CDN features like setting custom cache rules.

Here's an organized set of notes based on your Azure notes script for "Explore Azure Content Delivery Networks":

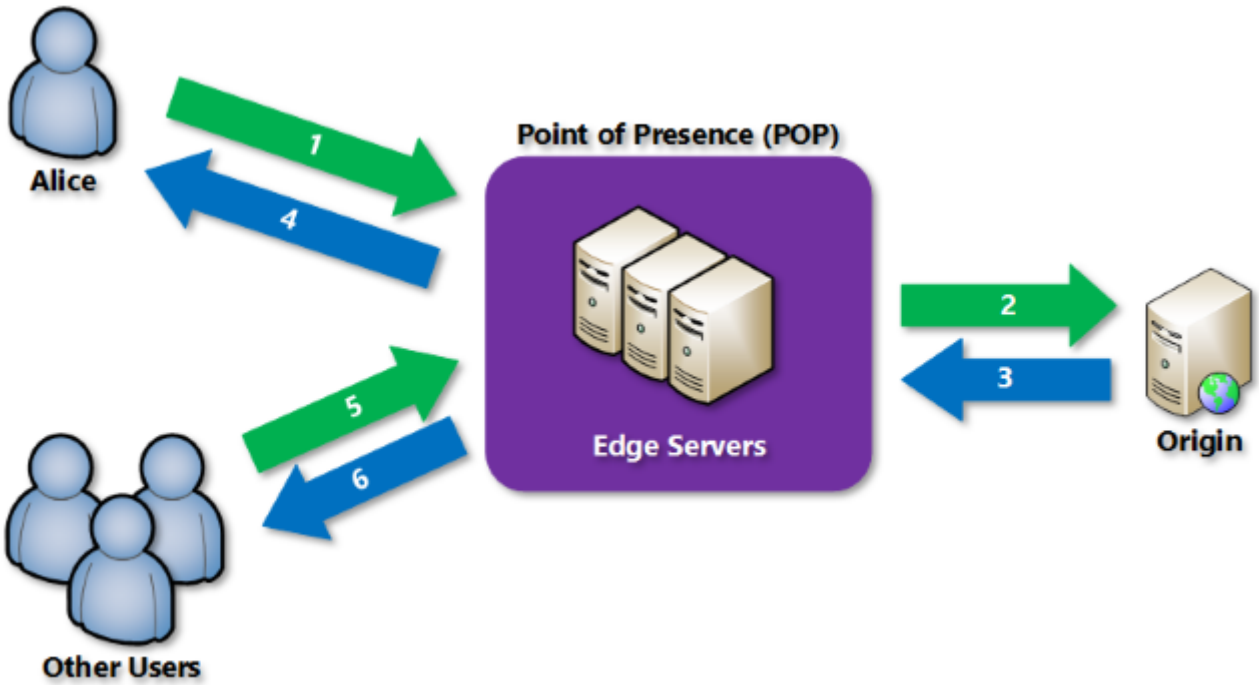
Explore Azure Content Delivery Networks

- **Status**: Completed
- **XP**: 100 XP
- **Duration**: 3 minutes

Overview:

- **Azure CDN**: A global solution for high-bandwidth content delivery by caching at strategic locations worldwide.
- **Benefits**:
  - **Performance**: Enhances user experience by minimizing latency, particularly for applications requiring multiple data requests.
  - **Scalability**: Capable of handling sudden spikes in traffic, like product launches.
  - **Load Distribution**: Reduces traffic to origin servers by serving content from edge servers.

How Azure CDN Works:



- **User Request**: Alice requests content using a CDN-specific URL (like `<endpoint name>.azureedge.net`).
- **DNS Routing**: The request is directed to the nearest or best-performing Point of Presence (POP).

- **Cache Check**:
  - If content is not cached, the POP fetches it from the origin server (Azure Web App, Storage, etc.).
  - Once fetched, the file is cached at the POP.
- **Content Delivery**: The file is served to Alice from the cache until its TTL (Time-To-Live) expires.
- **Subsequent Requests**: Other users requesting the same content can access it from the cached location, enhancing speed and responsiveness.

Requirements:

- **Azure Subscription**: Mandatory for using Azure CDN.
- **CDN Profile**: A collection of CDN endpoints; multiple profiles can be used for different configurations or pricing tiers.
- **Endpoints**: Customizable configurations for content delivery behavior.

Limitations:

- **Resource Limits**: Per Azure subscription, there are limits on:
  - Number of CDN profiles.
  - Endpoints per profile.
  - Custom domains per endpoint.

For detailed limits, refer to the CDN limits documentation.

Azure CDN Features:

- **Dynamic Site Acceleration**: Optimizes delivery of dynamic content.
- **CDN Caching Rules**: Control how content is cached.
- **HTTPS Custom Domain Support**: Secure content delivery with custom domains.
- **Azure Diagnostics Logs**: Monitor CDN performance and issues.
- **File Compression**: Reduces bandwidth usage.
- **Geo-Filtering**: Control content access based on user location.

For a comprehensive feature list across different Azure CDN products, see Compare Azure CDN product features.

Code Example:

While direct interaction with CDN for these high-level features might not involve code in day-to-day operations (often managed through the Azure portal or REST API), here's how you might programmatically create a CDN endpoint using Azure SDK in C#:

```
using Microsoft.Azure.Management.Cdn.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent.Authentication;
using System;
using System.Threading.Tasks;

class Program
{
```

```
    static async Task Main(string[] args)
    {
        var credentials = new AzureCredentials(
            new ServicePrincipalLoginInformation
            {
                ClientId = "your-client-id",
                ClientSecret = "your-client-secret"
            },
            "your-tenant-id",
            AzureEnvironment.AzureGlobalCloud);

        var cdnManager = CdnManager.Authenticate(credentials, "your-subscription-
id");

        // Create a new CDN profile and endpoint
        var cdnProfile = await cdnManager.Profiles.Define("myCdnProfile")
            .WithRegion(Region.USEast)
            .WithExistingResourceGroup("your-resource-group")
            .CreateAsync();

        await cdnProfile.Endpoints.Define("myEndpoint")
            .WithOrigin("origin-hostname")
            .WithOriginHostHeader("origin-host-header")
            .WithHttpAllowed()
            .WithHttpsAllowed()
            .CreateAsync();

        Console.WriteLine("CDN profile and endpoint created.");
    }
}
```

- Replace placeholders with your actual credentials, resource group, and endpoint details.

This example shows how to set up a CDN profile and endpoint but remember, most CDN operations like setting rules or managing custom domains are typically done through Azure portal or via specific API calls.

Control Cache Behavior on Azure Content Delivery Networks

Overview:

- **Cache Control**: Managing when content is refreshed is crucial to ensure users receive the latest information.

Methods to Control Caching Behavior:

1. **Caching Rules**:
    - **Global Caching Rules**:
        - Applies one rule to all requests for an endpoint, overriding HTTP cache-directive headers.
    - **Custom Caching Rules**:

        - Allows specifying rules for particular paths or file extensions. Multiple rules can be set per endpoint, processed in sequence, and they override the global rule.

    Note: Caching rules are available only for **Azure CDN Standard from Edgio**. For other profiles:

    - **Azure CDN from Microsoft**: Use the Standard rules engine.
    - **Azure CDN Premium from Edgio**: Use the Edgio Premium rules engine in the Manage portal.

2. **Query String Caching**:

    - Controls how CDN handles caching when query strings are involved. If content isn't cacheable, this setting doesn't apply.

Standard Rules Engine:

- **Purpose**: Defines the final processing rules for requests by Azure CDN Standard.
- **Components**:
    - **Match Conditions**: Identify specific request types for applying actions. Up to four conditions per rule using AND logic.
    - **Actions**: What occurs when conditions are met.

Common Use Cases:

- Custom cache policies.
- Request redirection.
- Modification of HTTP headers.

Examples of Match Conditions:

Match Condition	Description
Device type	Targets requests from mobile or desktop devices.
HTTP version	Filters requests by the HTTP version used.
Request cookies	Evaluates requests based on cookie data from the client.
Post argument	Matches requests with specific POST method arguments.
Query string	Identifies requests containing specific query string parameters with matching patterns.

For a full list of match conditions, refer to Match conditions in the Standard rules engine for Azure Content Delivery Network.

Code Example:

While the actual implementation of these rules is usually done through the Azure Portal or via Azure CDN's REST API, here's a conceptual example of how you might set up a rule using Azure SDK in C#:

```
using Microsoft.Azure.Management.Cdn.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent;
using Microsoft.Azure.Management.ResourceManager.Fluent.Authentication;
using System;
using System.Threading.Tasks;

class CdnRulesEngine
{
    private readonly ICdnManager _cdnManager;

    public CdnRulesEngine(string clientId, string clientSecret, string tenantId,
string subscriptionId)
    {
        var credentials = new AzureCredentials(
            new ServicePrincipalLoginInformation
            {
                ClientId = clientId,
                ClientSecret = clientSecret
            },
            tenantId,
            AzureEnvironment.AzureGlobalCloud);

        _cdnManager = CdnManager.Authenticate(credentials, subscriptionId);
    }

    public async Task CreateCustomRuleAsync(string resourceGroupName, string
profileName, string endpointName)
    {
        var ruleSet = new RuleSet(
            new Rule(
                new RuleCondition[] {
                    new RequestHeaderCondition("User-Agent", Operator.Contains,
"Mobile")
                },
                new RuleAction[] {
                    new CachingAction(true, new TimeSpan(0, 0, 1)), // 1 minute
cache duration for mobile devices
                    new UrlRedirectAction("https://example.com/mobile")
                }
            )
        );

        await _cdnManager.Endpoints.UpdateRulesAsync(resourceGroupName,
profileName, endpointName, ruleSet);
        Console.WriteLine("Custom rule applied to the CDN endpoint.");
    }
}

class Program
{
    static async Task Main(string[] args)
    {
        var manager = new CdnRulesEngine("client-id", "client-secret", "tenant-
```

```
id", "subscription-id");
        await manager.CreateCustomRuleAsync("your-resource-group", "cdn-profile",
"cdn-endpoint");
    }
}
```

- Replace the placeholders with your actual Azure credentials and CDN details.

This example demonstrates creating a rule that checks for mobile devices in the user-agent header and applies caching and URL redirection actions. Note, this is a conceptual example; actual implementation would require more detailed interaction with Azure's CDN API.

Caching and Time to Live (TTL)

Overview:

- **Cache Control**: The `Cache-Control` header from the origin server dictates how long content is cached by Azure CDN.
- **Default TTL**: If not specified:
  - **Web Delivery**: 7 days
  - **Large Files**: 1 day
  - **Media Streaming**: 1 year

Setting Cache-Control Header:

- **Methods**:
  - **Configuration Files**: Can be set in web.config for IIS or App Service, or other server configurations.
  - **Programmatically**: Via server-side code in your application.

Content Updating:

- **TTL Expiration**: Content is refreshed when TTL expires upon the next client request.
- **Best Practice**: Use versioning of assets with new URLs for each update to ensure immediate CDN cache refresh.

Purging Content:

- **Methods**:
  - **Endpoint Specific or All**: Purge can be applied to one or all endpoints.
  - **Specific Files or All**: Specify file paths or use wildcards for broader purging.
  - **Using Azure CLI**:

```
az cdn endpoint purge \
    --content-paths '/css/*' '/js/app.js' \
    --name ContosoEndpoint \
    --profile-name DemoProfile \
    --resource-group ExampleGroup
```

Preloading Content:

- **Use Case**: Improve user experience by preloading content into cache.

```
az cdn endpoint load \
    --content-paths '/img/*' '/js/module.js' \
    --name ContosoEndpoint \
    --profile-name DemoProfile \
    --resource-group ExampleGroup
```

Geo-Filtering:

- **Functionality**: Allows or blocks content based on country/region codes.
- **Limitation**: For Azure CDN Standard from Microsoft, you can only allow or block the entire site, not individual paths.

Code Example:

Here's how you might programmatically set `Cache-Control` headers in an ASP.NET application:

```
using System.Web;

public class CacheControlHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        HttpResponse response = context.Response;

        // Set Cache-Control header for one hour
        response.Cache.SetCacheability(HttpCacheability.Public);
        response.Cache.SetMaxAge(TimeSpan.FromHours(1));
        response.Cache.SetLastModified(DateTime.UtcNow);

        // Your logic to serve the content here
        response.Write("Your content");
    }

    public bool IsReusable
    {
        get { return false; }
    }
}
```

- This example sets a public cache with a max-age of one hour. In a real-world scenario, you'd adjust this based on your content's needs.

Remember, the actual setting of `Cache-Control` headers might involve different approaches depending on whether you're using IIS, Azure App Service, or another hosting solution. Additionally, managing CDN through APIs or CLI as shown above can be part of a deployment or content management strategy.

Interact with Azure Content Delivery Networks by using .NET

Overview:

- **Azure CDN Library for .NET**: Used for automating the management of CDN profiles and endpoints.

Setup:

- **Installation**: Use Visual Studio Package Manager or .NET CLI to install `Microsoft.Azure.Management.Cdn`.

Code Examples:

1. **Create a CDN Client**:

```
static void Main(string[] args)
{
    // Create CDN client
    CdnManagementClient cdn = new CdnManagementClient(new
TokenCredentials(authResult.AccessToken))
        { SubscriptionId = subscriptionId };
}
```

- **Note**: `authResult.AccessToken` and `subscriptionId` should be defined elsewhere, typically through authentication processes.

2. **List CDN Profiles and Endpoints**:

```
private static void ListProfilesAndEndpoints(CdnManagementClient cdn)
{
    // List all the CDN profiles in this resource group
    var profileList = cdn.Profiles.ListByResourceGroup(resourceGroupName);
    foreach (Profile p in profileList)
    {
        Console.WriteLine("CDN profile {0}", p.Name);
        if (p.Name.Equals(profileName, StringComparison.OrdinalIgnoreCase))
        {
            // Hey, that's the name of the CDN profile we want to create!
            profileAlreadyExists = true;
        }

        // List all the CDN endpoints on this CDN profile
        Console.WriteLine("Endpoints:");
        var endpointList = cdn.Endpoints.ListByProfile(p.Name, resourceGroupName);
        foreach (Endpoint e in endpointList)
```

```
        {
            Console.WriteLine("-{0} ({1})", e.Name, e.HostName);
            if (e.Name.Equals(endpointName, StringComparison.OrdinalIgnoreCase))
            {
                // The unique endpoint name already exists.
                endpointAlreadyExists = true;
            }
        }
        Console.WriteLine();
    }
}
```

- **Note**:
  - `resourceGroupName`, `profileName`, and `endpointName` should be predefined constants or variables.
  - `profileAlreadyExists` and `endpointAlreadyExists` are used to check for existing resources to avoid duplication in further operations.

Key Points:

- **Authentication**: The example assumes you have an authentication token. In practice, you'd need to manage authentication, potentially using Azure AD for service principals or managed identities.
- **Resource Management**: The code checks for existing profiles and endpoints to manage resource creation or updates without conflicts.
- **Error Handling**: Real applications should include error handling for API calls, as network operations can fail or return unexpected results.

These examples demonstrate basic interactions with Azure CDN using the .NET SDK, focusing on client initialization and resource listing. Further operations like creating, updating, or deleting CDN resources would follow a similar pattern using the respective SDK methods.

Create CDN Profiles and Endpoints

Creating a CDN Profile:

```
private static void CreateCdnProfile(CdnManagementClient cdn)
{
    if (profileAlreadyExists)
    {
        Console.WriteLine("Profile {0} already exists.", profileName);
    }
    else
    {
        Console.WriteLine("Creating profile {0}.", profileName);
        ProfileCreateParameters profileParms =
            new ProfileCreateParameters() { Location = resourceLocation, Sku = new
Sku(SkuName.StandardVerizon) };
        cdn.Profiles.Create(profileName, profileParms, resourceGroupName);
```

```
    }
}
```

- **Note**:
  - `profileAlreadyExists` checks if the profile already exists to avoid duplication.
  - `ProfileCreateParameters` specifies the location and SKU (pricing tier) for the CDN profile.

Creating a CDN Endpoint:

```
private static void CreateCdnEndpoint(CdnManagementClient cdn)
{
    if (endpointAlreadyExists)
    {
        Console.WriteLine("Endpoint {0} already exists.", endpointName);
    }
    else
    {
        Console.WriteLine("Creating endpoint {0} on profile {1}.", endpointName,
profileName);
        EndpointCreateParameters endpointParms =
            new EndpointCreateParameters()
            {
                Origins = new List<DeepCreatedOrigin>() { new
DeepCreatedOrigin("Contoso", "www.contoso.com") },
                IsHttpAllowed = true,
                IsHttpsAllowed = true,
                Location = resourceLocation
            };
        cdn.Endpoints.Create(endpointName, endpointParms, profileName,
resourceGroupName);
    }
}
```

- **Note**:
  - `endpointAlreadyExists` checks for existing endpoints.
  - `EndpointCreateParameters` includes details like origins, HTTP/HTTPS allowance, and location.

Purging an Endpoint:

```
private static void PromptPurgeCdnEndpoint(CdnManagementClient cdn)
{
    if (PromptUser(String.Format("Purge CDN endpoint {0}?", endpointName)))
    {
        Console.WriteLine("Purging endpoint. Please wait...");
        cdn.Endpoints.PurgeContent(resourceGroupName, profileName, endpointName,
new List<string>() { "/*" });
        Console.WriteLine("Done.");
        Console.WriteLine();
```

```
    }
}
```

- **Note**:
  - `PromptUser` is assumed to be a method asking for user confirmation.
  - The purge operation clears all content from the endpoint with "/*" indicating all paths.

Key Points:

- **Resource Management**: These methods demonstrate how to check for existing resources before creation to avoid conflicts.
- **Security and Performance**: Configuring HTTPS alongside HTTP for endpoints ensures secure content delivery when possible.
- **Content Management**: Purging content is crucial for ensuring users receive the latest updates, especially for dynamic content.

These examples illustrate basic operations for managing Azure CDN resources through the .NET SDK. In practice, you would also need to handle exceptions, manage retries for API calls, and potentially deal with asynchronous operations for better application responsiveness.