.NET Microservices – Azure DevOps and AKS

Section 14: Azure DevOps - Notes

Introduction to Azure DevOps

Azure DevOps is a suite of development tools offered by Microsoft that provides a set of services for software development teams. It facilitates the entire DevOps lifecycle from planning to deployment, helping teams to plan, develop, deliver, and monitor applications.

Key Components of Azure DevOps:

1. **Azure Boards**: A project management tool that supports agile methodologies like Scrum and Kanban. It helps in tracking work with features such as backlogs, boards, and sprints.
2. **Azure Repos**: Provides Git repositories or Team Foundation Version Control (TFVC) for source control. It allows teams to collaborate on code development.
3. **Azure Pipelines**: A service that supports Continuous Integration (CI) and Continuous Deployment (CD) of applications. It automates the build, test, and deployment processes.
4. **Azure Test Plans**: Offers a set of tools for managing your testing efforts. It includes manual testing, exploratory testing, and user acceptance testing.
5. **Azure Artifacts**: Allows teams to share packages, manage dependencies, and integrate with CI/CD pipelines.

Benefits of Azure DevOps

- **Integrated Platform**: Combines several DevOps tools in one platform, providing seamless integration and communication between different stages of the development lifecycle.
- **Scalability**: Can handle projects of all sizes, from small teams to large enterprises, with its robust and scalable infrastructure.
- **Flexibility**: Supports various workflows and can integrate with other tools and services, offering flexibility in how teams manage their development process.
- **Automation**: Automates repetitive tasks like building, testing, and deployment, which improves efficiency and reduces the risk of human error.
- **Collaboration**: Facilitates better collaboration among team members through integrated tools like issue tracking, code review, and chat.

Key Points for Interview Preparation

- **Azure DevOps Services**: Familiarize yourself with the core services offered by Azure DevOps (Azure Boards, Repos, Pipelines, Test Plans, Artifacts).
- **DevOps Lifecycle**: Understand how Azure DevOps supports the entire software development lifecycle from planning to deployment.
- **Benefits**: Be able to explain the benefits of using Azure DevOps, including its integration, scalability, flexibility, and automation features.

Azure DevOps Organization

An Azure DevOps organization is a fundamental unit for managing your projects and users. It acts as a container for multiple projects and offers a centralized place for managing resources and settings.

- **Creating an Organization**: When you first sign up for Azure DevOps, you'll create an organization. This organization serves as a namespace for your projects and teams.
- **Organization Settings**: You can configure settings like security, billing, and policies at the organization level. This includes managing user access, setting up policies for code repositories, and configuring integrations with other services.
- **Permissions**: Organizations allow you to define and manage permissions at various levels. You can control who has access to what within your organization, including who can create projects, manage repositories, and access pipelines.

Projects in Azure DevOps

Projects are containers within an Azure DevOps organization that hold all the assets related to a particular software project.

- **Creating a Project**: You can create multiple projects within an organization. Each project can contain its own set of boards, repos, pipelines, test plans, and artifacts.
- **Project Settings**: Within a project, you can configure settings like boards, repositories, pipelines, and security policies. Each project can have its own unique settings and configurations.
- **Project Features**: Azure DevOps projects come with features like work item tracking, source control, build and release pipelines, and test management.
- **Project Access**: Access to projects can be controlled via security groups and permissions. You can specify who can view, edit, and manage different aspects of the project.
- **Project Templates**: When creating a new project, you can select from various templates that help set up your project with predefined settings and configurations based on your project type (e.g., Agile, Scrum).

Managing Multiple Projects

- **Project Collection**: An organization can have multiple projects, and these projects are grouped together under a project collection. This allows for better organization and management of related projects.
- **Cross-Project Collaboration**: Azure DevOps enables collaboration across different projects within the same organization. Teams can share resources, track work items, and manage dependencies across projects.

Key Points for Interview Preparation

- **Organization Structure**: Understand the concept of Azure DevOps organization and how it relates to projects. Be able to explain the purpose and benefits of creating multiple projects within an organization.
- **Permissions and Access**: Be familiar with how permissions are managed at the organization and project levels. Know how to configure user access and control permissions.
- **Project Features**: Have a good grasp of the features available within Azure DevOps projects and how they support various stages of the software development lifecycle.
- **Managing Projects**: Be able to discuss how to manage multiple projects and the benefits of organizing related projects within a single Azure DevOps organization.

Parallelism Request for Pipelines

Parallelism in Azure DevOps pipelines refers to the ability to run multiple jobs or stages simultaneously. This helps in speeding up the overall build and deployment process by utilizing concurrent execution.

Pipeline Parallelism

1. **Jobs and Stages**: In a pipeline, you can define multiple jobs that can run in parallel. Jobs within a stage can run concurrently, and different stages can also run in parallel if they are not dependent on each other.
2. **Concurrency Limits**: Azure DevOps provides a default level of parallelism, which can be increased by requesting additional parallel jobs if needed. Each organization has a predefined number of parallel jobs available based on the pricing tier.

3. **Parallel Jobs in Azure DevOps**:
    - **Free Tier**: Includes a limited number of parallel jobs (e.g., 1 parallel job for public projects and 1 parallel job for private projects).
    - **Paid Plans**: Additional parallel jobs can be purchased if the default number is insufficient for your needs.

Requesting Additional Parallel Jobs

1. **Pricing Tiers**: Understand the different pricing tiers offered by Azure DevOps for parallel jobs. Upgrading your plan may provide additional parallel jobs and other benefits.
2. **How to Request More Jobs**:
    - **Navigate to Organization Settings**: Go to your Azure DevOps organization and find the settings for parallel jobs.
    - **Select the Appropriate Plan**: Choose the plan that meets your needs for parallel job execution.
    - **Request Additional Parallel Jobs**: Follow the process to request or purchase additional parallel jobs if necessary.
3. **Usage Considerations**:
    - **Cost Management**: Keep an eye on the cost associated with additional parallel jobs. Efficient use of parallelism can help reduce the time taken for builds and deployments.
    - **Resource Utilization**: Ensure that your build agents are capable of handling the increased parallelism without performance degradation.

Configuring Parallelism in Pipelines

1. **Pipeline YAML Configuration**: Configure parallel jobs in your pipeline YAML file by defining multiple jobs within stages. Azure DevOps will automatically handle the parallel execution based on the available agents and configured limits.

Example YAML configuration:

```
jobs:
 - job: Job1
   pool:
    vmImage: 'ubuntu-latest'
   steps:
    - script: echo This is Job 1
```

```
- job: Job2
  pool:
    vmImage: 'ubuntu-latest'
  steps:
    - script: echo This is Job 2

- job: Job3
  dependsOn: []
  pool:
    vmImage: 'ubuntu-latest'
  steps:
    - script: echo This is Job 3
```

In this example, Job1, Job2, and Job3 will run in parallel because they are not dependent on each other.

2. **Managing Dependencies**: Use dependsOn to specify dependencies between jobs and control the order of execution while still allowing parallelism where possible.

Key Points for Interview Preparation

- **Parallel Jobs Concept**: Be able to explain what parallelism is in the context of Azure DevOps pipelines and how it benefits the build and deployment process.
- **Configuring Parallelism**: Know how to configure parallel jobs in Azure DevOps pipelines, including YAML configuration and setting up parallel jobs in the pipeline definition.
- **Managing Parallelism**: Understand the limitations and costs associated with parallel jobs and how to request or purchase additional parallel jobs if needed.
- **Resource Utilization**: Be familiar with the implications of parallelism on resource utilization and performance.

Introduction to Azure Boards (Scrum)

Azure Boards is a service provided by Azure DevOps that helps manage and track work in software development projects. It supports various methodologies, including Scrum, Kanban, and Agile, allowing teams to plan, track, and discuss work in a collaborative environment.

Key Features of Azure Boards

1. **Work Items**: Azure Boards uses work items to track tasks, bugs, features, user stories, and other work-related activities. Each work item has a unique ID and can be customized with fields, states, and workflows.
2. **Boards**: The boards provide a visual representation of work items and their states. For Scrum, this includes Sprint Boards and Backlogs.
3. **Backlogs**: The backlog is a prioritized list of work items that need to be completed. It can include product backlog items (PBIs), user stories, tasks, and bugs.
4. **Sprints**: In Scrum, work is organized into sprints, which are time-boxed iterations. Azure Boards supports sprint planning, tracking, and reviewing.
5. **Queries**: Queries help filter and search for work items based on specific criteria. They can be saved and shared with team members.
6. **Dashboards**: Customizable dashboards provide insights and visualizations of project metrics, such as work item progress, sprint burndown charts, and team velocity.

Using Azure Boards with Scrum

1. **Creating and Managing Teams**:
    - **Teams**: Create and manage teams within Azure Boards to organize work and manage sprints. Each team can have its own backlog, board, and sprints.
    - **Team Settings**: Configure team settings, including iteration paths, areas, and team capacity.
2. **Creating a Scrum Project**:
    - **New Project**: When creating a new project, select the "Scrum" process template. This template includes predefined work item types and workflows suited for Scrum methodology.
    - **Configuring Backlogs and Sprints**: Set up and configure product and sprint backlogs. Define iteration paths to represent different sprints.
3. **Work Item Types**:
    - **User Stories**: Represent features or functionality from the user's perspective.
    - **Tasks**: Break down user stories or features into actionable items.
    - **Bugs**: Track defects or issues found during development or testing.
    - **Epics and Features**: Higher-level work items that represent significant work or project milestones.
4. **Planning and Tracking Sprints**:
    - **Sprint Planning**: Plan and prioritize work items for each sprint. Add user stories and tasks to the sprint backlog.

    - **Sprint Execution**: Track progress during the sprint using the sprint board. Update work items as they move through different states.
    - **Sprint Review and Retrospective**: Review completed work at the end of the sprint and conduct a retrospective to discuss improvements.
5. **Reporting and Metrics**:
    - **Burndown Charts**: Visualize the progress of work completed versus work remaining during a sprint.
    - **Velocity Charts**: Track the amount of work completed in each sprint to measure team performance and capacity.
6. **Customizing Boards and Backlogs**:
    - **Columns and States**: Customize board columns and work item states to fit your team's workflow.
    - **Work Item Types and Fields**: Modify work item types and add custom fields as needed for your project.

Configuring Azure Boards with Scrum

1. **Setting Up Boards**:
    - Navigate to Azure Boards and select the Scrum process template.
    - Configure boards for each team, ensuring they reflect the Scrum workflow.
2. **Managing Backlogs**:
    - Access the product backlog and add user stories, bugs, and tasks.
    - Prioritize backlog items based on business value and project requirements.
3. **Planning Sprints**:
    - Define sprint iterations and assign backlog items to each sprint.
    - Adjust sprint goals and capacity based on team availability and workload.

Key Points for Interview Preparation

- **Understanding Azure Boards**: Be able to explain the key features and benefits of Azure Boards, including work items, boards, backlogs, and sprints.
- **Scrum Methodology**: Understand how Azure Boards supports Scrum methodology, including sprint planning, tracking, and review.
- **Configuring Azure Boards**: Know how to create and configure Scrum projects, manage teams, and customize boards and backlogs.

- **Reporting and Metrics**: Be familiar with the types of reports and metrics available in Azure Boards, such as burndown charts and velocity charts.
- **Customizing Work Items**: Understand how to customize work items, fields, and workflows to fit the needs of your project.

Azure Repos

Azure Repos is a set of version control tools that you can use to manage your code. It offers Git repositories or Team Foundation Version Control (TFVC) to help manage your source code and track changes over time. Azure Repos is integrated with Azure DevOps and supports collaborative development, code reviews, and versioning.

Key Features of Azure Repos

1. **Git Repositories**: Git is a distributed version control system that allows multiple developers to work on code simultaneously. Azure Repos provides Git-based repositories that offer powerful branching and merging capabilities.
2. **TFVC (Team Foundation Version Control)**: TFVC is a centralized version control system where changes are tracked in a central repository. It is less commonly used compared to Git but is still supported by Azure Repos.
3. **Branching and Merging**: Azure Repos supports branching strategies that help manage code changes and feature development. You can create branches for new features, bug fixes, or experiments and merge them into the main branch when ready.
4. **Pull Requests**: Pull requests are a way to review and merge changes from one branch into another. They facilitate code reviews and discussions among team members before changes are integrated.
5. **Code Reviews**: Code reviews are integrated into the pull request process. Team members can comment on code changes, request modifications, and approve or reject pull requests.
6. **Branch Policies**: Branch policies enforce rules for code quality and process. For example, you can require that code passes automated tests before it can be merged into a branch.
7. **Commit History**: Azure Repos tracks the history of all changes made to the codebase. You can view the commit history, compare changes between commits, and revert changes if necessary.

Using Azure Repos

1. **Creating a Repository**:
    - **Create a New Repository**: Navigate to Azure Repos in Azure DevOps, and create a new Git repository. You can choose to initialize it with a README file or import an existing repository.
    - **Clone a Repository**: Clone the repository to your local machine using Git commands or Git clients. For example:

git clone https://dev.azure.com/{organization}/{project}/_git/{repository}

2. **Branching**:
    - **Create a Branch**: Create a new branch to work on a specific feature or bug fix. For example:

git checkout -b feature/new-feature

    - **Merge a Branch**: Once your changes are complete, merge the branch back into the main branch. For example:
    - git checkout main

git merge feature/new-feature

3. **Pull Requests**:
    - **Create a Pull Request**: Open a pull request to propose changes from one branch to another. Add a title, description, and reviewers.
    - **Review and Merge**: Review the code changes, discuss any issues, and address feedback. Approve the pull request and merge it into the target branch.
4. **Branch Policies**:
    - **Configure Policies**: Set up branch policies to enforce code quality standards. For example, require successful build validation and code reviews before merging.
5. **Commit History**:
    - **View Commit History**: Access the commit history to view changes, compare versions, and find specific commits.
6. **Code Reviews**:
    - **Request a Code Review**: Request a code review from team members as part of the pull request process. Provide feedback and make changes as needed.

Branching Strategies

1. **Feature Branching**:
    - **Create a Feature Branch**: Develop new features in separate branches. This isolates feature development from the main branch.

2. **Release Branching**:
    - **Create Release Branches**: Manage releases by creating branches for different versions or stages of your application.
3. **Hotfix Branching**:
    - **Create Hotfix Branches**: Address critical issues in production by creating hotfix branches that can be quickly merged into the main branch.
4. **Main Branch**:
    - **Main Branch**: The main branch (often called main or master) represents the stable version of your codebase. Only tested and approved changes should be merged into this branch.

Key Points for Interview Preparation

- **Understanding Azure Repos**: Be able to explain the key features and benefits of Azure Repos, including Git repositories, TFVC, and branching strategies.
- **Branching and Merging**: Understand how to create and manage branches, merge changes, and resolve conflicts.
- **Pull Requests**: Know how to create, review, and merge pull requests. Understand the role of code reviews in maintaining code quality.
- **Branch Policies**: Be familiar with configuring branch policies to enforce code quality standards and automated builds.
- **Commit History**: Understand how to view and manage commit history, and use it to track changes and revert if necessary.

Git Branches

Branches in Git allow you to work on different versions of a project simultaneously. They are essential for managing features, bug fixes, and experiments in isolation from the main codebase.

Key Concepts:

1. **Main Branch**: The default branch in a Git repository, usually named main or master. This branch should contain the stable code that is ready for production.
2. **Feature Branch**: Created for developing new features. Named descriptively to indicate the feature being developed, e.g., feature/user-authentication.
3. **Bugfix Branch**: Used to fix bugs or issues in the code. Named to reflect the bug or issue being addressed, e.g., bugfix/fix-login-error.
4. **Release Branch**: Used to prepare for a new release. Allows for final testing and minor adjustments before merging into the main branch, e.g., release/v1.0.

5. **Hotfix Branch**: Used to address critical issues in the production code quickly. Created from the main branch, e.g., hotfix/fix-critical-bug.

Creating and Managing Branches:

1. **Create a Branch**: Use the following command to create a new branch and switch to it:

git checkout -b branch-name

Example:

git checkout -b feature/new-login

2. **List Branches**: View all branches in your repository with:

git branch

3. **Switch Branches**: Change to a different branch using:

git checkout branch-name

Example:

git checkout main

4. **Merge Branches**: Integrate changes from one branch into another. Switch to the target branch and use:

git merge branch-name

Example:

git checkout main

git merge feature/new-login

5. **Delete a Branch**: Remove a branch that is no longer needed:

git branch -d branch-name

Example:

git branch -d feature/new-login

Pull Requests

Pull Requests (PRs) are a way to propose changes to the codebase and facilitate code review before merging changes into the main branch.

Creating a Pull Request:

1. **Create a Pull Request**: Go to the Azure Repos or GitHub web interface and navigate to the repository. Choose the branch you want to merge from and the target branch (usually main or master).
2. **Add a Title and Description**: Provide a clear title and description of the changes made. This helps reviewers understand the purpose of the PR.
3. **Add Reviewers**: Assign team members to review the PR. Reviewers can comment, suggest changes, and approve or reject the PR.
4. **Review Process**: Reviewers examine the code changes, provide feedback, and request modifications if necessary. This process ensures code quality and consistency.
5. **Merge the Pull Request**: Once the PR is approved, it can be merged into the target branch. This is usually done via the web interface, but can also be done using Git commands:
6. git checkout main

git merge feature/new-login

7. **Close the Pull Request**: After merging, the PR is closed. This can also be done manually if the PR is not to be merged.

Best Practices for Pull Requests:

1. **Small, Focused PRs**: Keep pull requests small and focused on a single change or feature to make them easier to review.
2. **Descriptive Titles and Descriptions**: Use clear and descriptive titles and descriptions for pull requests to provide context.
3. **Automated Checks**: Integrate automated checks (e.g., CI/CD pipelines) to validate code changes before review.
4. **Address Feedback Promptly**: Respond to reviewer feedback promptly and make necessary changes to facilitate smooth merging.
5. **Ensure Code Quality**: Review code for adherence to coding standards, best practices, and potential issues before merging.

Key Points for Interview Preparation

- **Understanding Git Branches**: Be able to explain the different types of branches (main, feature, bugfix, release, hotfix) and their purposes.

- **Branch Management**: Understand how to create, switch, list, and delete branches. Be familiar with the commands used for these operations.
- **Pull Requests**: Know the process for creating, reviewing, and merging pull requests. Understand the importance of pull requests in code quality and collaboration.
- **Review Practices**: Be aware of best practices for managing pull requests, including maintaining small, focused PRs, using descriptive titles, and addressing feedback promptly.
- **Branch Policies**: Understand how branch policies can enforce code quality and ensure that code meets certain standards before being merged.

Introduction to Azure Pipelines

Azure Pipelines is a cloud-based service within Azure DevOps that provides Continuous Integration (CI) and Continuous Deployment (CD) capabilities. It automates the process of building, testing, and deploying code, allowing teams to deliver software faster and more reliably.

Key Concepts of Azure Pipelines

1. **CI/CD Pipelines**:
    - **Continuous Integration (CI)**: Automates the process of building and testing code changes to ensure they integrate well into the shared codebase.
    - **Continuous Deployment (CD)**: Automates the deployment of code changes to various environments (e.g., staging, production) after passing the CI stage.
2. **Pipeline Components**:
    - **Pipeline**: Defines the workflow for building, testing, and deploying code. It consists of stages, jobs, and steps.
    - **Stage**: A phase in the pipeline that groups jobs. Stages are used to represent different stages in the deployment process, such as build, test, and deploy.
    - **Job**: A collection of steps that are executed sequentially. Jobs run on agents and can be parallelized across multiple agents.
    - **Step**: The smallest unit of work in a pipeline. Steps are individual tasks like running a script or executing a command.
3. **Pipeline YAML vs. Classic Editor**:
    - **YAML Pipelines**: Written in YAML syntax, allowing you to define the pipeline configuration as code. This is more flexible and version-controlled.

- **Classic Editor**: A graphical interface to create pipelines, which provides a drag-and-drop experience for configuring pipeline tasks.

Creating Azure Pipelines

1. **Setting Up a Pipeline**:
   - **Navigate to Azure Pipelines**: In your Azure DevOps project, go to Pipelines > New Pipeline.
   - **Choose a Repository**: Select the source repository for the code. This can be Azure Repos, GitHub, or another Git repository.
   - **Configure Pipeline**: Choose to use the YAML file or the Classic Editor to configure the pipeline.
2. **YAML Pipeline Configuration**:
   - **Define Pipeline**: Create a YAML file (e.g., azure-pipelines.yml) in the root of your repository to define the pipeline.
   - **Basic YAML Structure**:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
  - script: echo Hello, world!
    displayName: 'Run a one-line script'
```

     - trigger: Specifies which branches trigger the pipeline.
     - pool: Defines the VM image to use for the build agent.
     - steps: Lists the tasks to run, such as scripts or commands.
3. **Classic Pipeline Configuration**:
   - **Add Tasks**: Use the Classic Editor to add tasks such as build, test, and deploy. Drag and drop tasks onto the pipeline canvas and configure them as needed.

- **Define Pipeline Variables**: Set pipeline variables that can be used across the pipeline to manage configuration settings and secrets.

Adding Stages for Deploying ASP.NET Core Microservice Apps to AKS Cluster

1. **Defining Stages**:
   - **Build Stage**: Compiles the code and creates a Docker image.
   - **Test Stage**: Runs unit tests and integration tests.
   - **Deploy Stage**: Deploys the Docker image to an Azure Kubernetes Service (AKS) cluster.
2. **Sample YAML Configuration**:

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

stages:
  - stage: Build
    jobs:
      - job: Build
        steps:
          - task: Docker@2
            inputs:
              command: 'build'
              dockerfile: '**/Dockerfile'
              tags: '$(Build.BuildId)'

  - stage: Test
    dependsOn: Build
    jobs:
```

```
    - job: Test
      steps:
        - script: dotnet test
          displayName: 'Run unit tests'

  - stage: Deploy
    dependsOn: Test
    jobs:
      - job: Deploy
        steps:
          - task: AzureCLI@2
            inputs:
              azureSubscription: 'YourAzureSubscription'
              scriptType: 'bash'
              scriptPath: 'deploy-to-aks.sh'
```

- **Build Stage**: Uses Docker to build the image.
- **Test Stage**: Runs tests to ensure code quality.
- **Deploy Stage**: Uses Azure CLI to deploy to AKS using a script.

3. **Deployment Script (deploy-to-aks.sh)**:
4. kubectl apply -f k8s/deployment.yaml

kubectl apply -f k8s/service.yaml

- **deployment.yaml**: Defines the Kubernetes deployment.
- **service.yaml**: Defines the Kubernetes service.

Working with Environments in Azure DevOps

Environments in Azure DevOps help manage deployments to different environments like development, staging, and production.

1. **Creating Environments**:
    - Navigate to Pipelines > Environments.
    - Create a new environment and define the deployment targets, such as Kubernetes clusters or virtual machines.
2. **Using Environments in Pipelines**:
    - Add environment references in the pipeline YAML to deploy to specific environments.

```
jobs:
 - deployment: DeployToProduction
   environment: 'production'
   strategy:
    runOnce:
     deploy:
      steps:
       - task: AzureCLI@2
         inputs:
          azureSubscription: 'YourAzureSubscription'
          scriptType: 'bash'
          scriptPath: 'deploy-to-production.sh'
```

Variable Groups

Variable Groups allow you to manage and share variables across multiple pipelines. They are useful for managing configuration settings and secrets.

1. **Creating Variable Groups**:
    - Navigate to Pipelines > Library > Variable groups.
    - Create a new variable group and add variables.
2. **Using Variable Groups in Pipelines**:
    - Reference variable groups in your pipeline YAML.
3. variables:

- group: my-variable-group

Azure Key Vault Integration

Azure Key Vault is a service that securely stores and manages sensitive information like secrets, keys, and certificates.

1. **Setting Up Azure Key Vault**:
    - Create a Key Vault in the Azure portal.
    - Add secrets to the Key Vault.
2. **Accessing Key Vault Secrets in Pipelines**:
    - Use the Azure Key Vault task to fetch secrets.

```
- task: AzureKeyVault@2
  inputs:
    azureSubscription: 'YourAzureSubscription'
    KeyVaultName: 'YourKeyVaultName'
    SecretsFilter: '*'
    RunAsPreJob: false
```

    - Secrets can then be accessed in subsequent pipeline steps using the variable names defined in the Key Vault.

Key Points for Interview Preparation

- **Azure Pipelines Overview**: Understand the role of Azure Pipelines in CI/CD and its components (stages, jobs, steps).
- **YAML vs. Classic Pipelines**: Be able to explain the difference between YAML and Classic pipelines and their use cases.

- **Pipeline Stages**: Know how to configure stages for building, testing, and deploying applications, including examples of YAML configurations.
- **Environments**: Understand how to create and use environments in Azure DevOps for managing different deployment targets.
- **Variable Groups**: Be familiar with creating and using variable groups to manage configuration settings across pipelines.
- **Azure Key Vault**: Know how to integrate Azure Key Vault with Azure Pipelines to manage secrets securely.

Adding Stages for Deploying ASP.NET Core Microservice Apps to AKS Cluster

In this part, we'll dive into how to add stages to your Azure Pipelines YAML configuration to deploy ASP.NET Core microservice applications to an Azure Kubernetes Service (AKS) cluster.

Overview of Pipeline Stages for AKS Deployment

A typical CI/CD pipeline for deploying to AKS involves several stages:

1. **Build**: Compile the code and package it into Docker images.
2. **Test**: Run unit tests and integration tests to ensure code quality.
3. **Deploy**: Deploy the Docker images to an AKS cluster.

Detailed Pipeline Configuration

1. **Define Your Pipeline**

Start by creating a azure-pipelines.yml file in your repository. This YAML file defines your CI/CD pipeline.

```
trigger:
  branches:
    include:
      - main
```

- **trigger**: Specifies the branch that triggers the pipeline. Here, it is set to trigger on changes to the main branch.

2. **Define Pipeline Stages**

The pipeline is divided into multiple stages: Build, Test, and Deploy.

Build Stage

```
stages:
 - stage: Build
   displayName: 'Build Docker Image'
   jobs:
    - job: Build
      pool:
       vmImage: 'ubuntu-latest'
      steps:
       - task: Docker@2
         inputs:
           command: 'build'
           dockerfile: '**/Dockerfile'
           tags: '$(Build.BuildId)'
           containerRegistry: 'myContainerRegistry'
```

- **stage: Build**: Defines the build stage of the pipeline.
- **displayName**: Provides a human-readable name for the stage.
- **jobs**: Contains the jobs to be executed in this stage.
- **job: Build**: Defines a job named Build.
- **pool**: Specifies the VM image to use for the build agent.
- **steps**: Lists the tasks to perform. In this case, the Docker task builds the Docker image from the Dockerfile.

Test Stage

```
 - stage: Test
   displayName: 'Run Unit Tests'
   dependsOn: Build
   jobs:
```

```
  - job: Test
    pool:
      vmImage: 'ubuntu-latest'
    steps:
      - script: dotnet test
        displayName: 'Run unit tests'
```

- **stage: Test**: Defines the test stage.
- **dependsOn: Build**: Specifies that this stage depends on the successful completion of the Build stage.
- **steps**: Runs the dotnet test command to execute unit tests.

Deploy Stage

```
- stage: Deploy
  displayName: 'Deploy to AKS'
  dependsOn: Test
  jobs:
    - job: Deploy
      pool:
        vmImage: 'ubuntu-latest'
      steps:
        - task: AzureCLI@2
          inputs:
            azureSubscription: 'YourAzureSubscription'
            scriptType: 'bash'
            scriptPath: 'scripts/deploy-to-aks.sh'
```

- **stage: Deploy**: Defines the deployment stage.
- **dependsOn: Test**: Specifies that this stage depends on the successful completion of the Test stage.
- **steps**: Executes a script using Azure CLI to deploy to AKS.

3. **Deployment Script (deploy-to-aks.sh)**

Create a script named deploy-to-aks.sh in the scripts directory. This script contains the commands to deploy your application to AKS.

```
#!/bin/bash

# Set variables
IMAGE_NAME="myapp"
IMAGE_TAG="$(Build.BuildId)"
REGISTRY_NAME="myContainerRegistry"
AKS_CLUSTER_NAME="myAKSCluster"
RESOURCE_GROUP="myResourceGroup"

# Log in to Azure
az login --service-principal -u $SP_APP_ID -p $SP_PASSWORD --tenant $TENANT_ID

# Set the current subscription
az account set --subscription $AZURE_SUBSCRIPTION_ID

# Get AKS credentials
az aks get-credentials --resource-group $RESOURCE_GROUP --name $AKS_CLUSTER_NAME

# Deploy the Docker image
kubectl apply -f k8s/deployment.yaml
kubectl apply -f k8s/service.yaml
```

- **az login**: Authenticates with Azure using service principal credentials.
- **az aks get-credentials**: Configures kubectl to use the AKS cluster credentials.
- **kubectl apply**: Deploys the Docker image to the AKS cluster using Kubernetes manifests.

4. **Kubernetes Manifests**

Deployment Manifest (k8s/deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myContainerRegistry.azurecr.io/myapp:$(Build.BuildId)
          ports:
            - containerPort: 80
```

- **apiVersion**: Specifies the Kubernetes API version.
- **kind**: Defines the resource type (Deployment).
- **replicas**: Specifies the number of replicas (instances) of the application.
- **containers**: Defines the Docker container image and port configuration.

Service Manifest (k8s/service.yaml)

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: myapp-service
spec:
  type: LoadBalancer
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

- **kind**: Defines the resource type (Service).
- **type**: Specifies the service type (LoadBalancer to expose the service externally).
- **selector**: Matches the service to the deployment using labels.

Key Points for Interview Preparation

- **Pipeline Stages**: Understand how to define and configure stages for build, test, and deploy operations in Azure Pipelines.
- **YAML Configuration**: Be familiar with YAML syntax and structure for creating pipeline configurations.
- **Deployment Scripts**: Know how to write and use deployment scripts to deploy applications to AKS.
- **Kubernetes Manifests**: Understand the purpose of deployment and service manifests in Kubernetes and how to configure them.

Working with Environments in Azure DevOps

In Azure DevOps, **Environments** provide a way to manage and control the deployment of applications to various stages or environments such as Development, Testing, Staging, and Production. They offer a more structured approach to manage deployments and enforce deployment policies.

Overview of Environments

Environments in Azure DevOps help manage the deployment of your applications by providing:

1. **Deployment Resources**: Track and manage resources like Kubernetes clusters or virtual machines.
2. **Approval Gates**: Implement pre-deployment and post-deployment approvals.
3. **Checks**: Enforce security and compliance checks before deployment.

Creating and Configuring Environments

1. **Creating an Environment**

To create an environment in Azure DevOps:

- Go to your Azure DevOps project.
- Navigate to **Pipelines** > **Environments**.
- Click on **New environment**.
- Provide a name for the environment (e.g., Staging or Production).
- Optionally, add resources like Kubernetes clusters or virtual machines.

2. **Configuring Environment Approvals and Checks**

Approvals and Checks are set up to control when and how deployments happen. This is crucial for maintaining quality and compliance.

- **Approvals**: Configure who must approve deployments to the environment. You can add multiple approvers and set conditions for approval.
- **Checks**: Set up checks for security, compliance, and quality. Examples include checking for the latest security patches or verifying that all tests pass.

To configure approvals and checks:

- Go to the environment you created.
- Click on **Approvals and Checks**.
- Add new checks or approvals as needed.

Example Configuration:

- **Approvals**: Require approval from a specific user or group before proceeding with the deployment.
- **Checks**: Ensure that a security scan is completed or a specific compliance requirement is met before deployment.

3. **Using Environments in Pipelines**

To use environments in your Azure Pipelines YAML file, specify the environment in your deployment steps:

```
stages:
  - stage: Deploy
    displayName: 'Deploy to AKS'
    jobs:
      - deployment: DeployToAKS
        environment: 'Staging'
        strategy:
          runOnce:
            deploy:
              steps:
                - task: AzureCLI@2
                  inputs:
                    azureSubscription: 'YourAzureSubscription'
                    scriptType: 'bash'
                    scriptPath: 'scripts/deploy-to-aks.sh'
```

4. **environment**: Specifies the environment to which the application will be deployed (e.g., Staging).

The pipeline will use the configuration and policies associated with the specified environment during the deployment process.

5. **Monitoring and Managing Deployments**
    - **Deployment History**: View the history of deployments to the environment. This helps track successful and failed deployments.
    - **Resource Health**: Monitor the health of resources associated with the environment. Ensure that all deployed applications are running smoothly.

Key Points for Interview Preparation

- **Environments**: Understand the concept and purpose of environments in Azure DevOps.
- **Approvals and Checks**: Know how to configure approvals and checks to control and enforce deployment policies.
- **Pipeline Integration**: Be familiar with how to reference environments in Azure Pipelines YAML configurations.
- **Resource Management**: Be aware of how to manage and monitor resources and deployments within an environment.

Variable Groups in Azure DevOps

Variable Groups in Azure DevOps are a way to manage and share values across multiple pipelines. They are useful for storing and reusing common configuration settings, secrets, and environment-specific values.

Overview of Variable Groups

- **Purpose**: To centralize the management of variables used in different pipelines, avoiding duplication and easing management.
- **Scope**: Variables can be scoped to different pipelines, stages, or environments. They can also be used across different projects.

Creating and Configuring Variable Groups

1. **Creating a Variable Group**

To create a Variable Group:

   - Go to your Azure DevOps project.
   - Navigate to **Pipelines** > **Library**.
   - Click on **Variable groups**.
   - Click on **+ Variable group** to create a new group.
   - Provide a name for the variable group (e.g., ProductionSecrets).

Example:

- **Name**: ProductionSecrets
- **Description**: Variables used in production deployments

2. **Adding Variables to the Group**
    - After creating the variable group, click on it to open.
    - Click on **Add** to add new variables.
    - Enter the name and value for each variable. For sensitive data, you can mark the variable as **secret**, which will encrypt the value and hide it in logs.

Example:

- **Variable Name**: ConnectionString
- **Value**: Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
- **Secret**: Checked

3. **Linking Variable Groups to Pipelines**

To use the variable group in your pipelines:

- Edit your pipeline YAML file.
- Add a reference to the variable group under the variables section.

Example YAML Configuration:

```
variables:
 - group: ProductionSecrets
```

- This includes all variables defined in the ProductionSecrets group into your pipeline, making them available for use.

4. **Using Variables in Pipelines**

Once linked, you can use these variables in your pipeline tasks, scripts, and other steps.

Example Usage in a Pipeline:

```
jobs:
 - job: Deploy
   steps:
    - script: |
```

```
echo $(ConnectionString)
displayName: 'Print connection string'
```

- **$(ConnectionString)**: This references the variable ConnectionString from the ProductionSecrets variable group.

5. **Managing Variable Groups**
   - **Editing Variables**: Update the value of variables if needed. Changes are automatically reflected in all pipelines using the group.
   - **Deleting Variable Groups**: Remove a variable group when it is no longer needed. Ensure that no pipeline relies on the variables before deletion.

Key Points for Interview Preparation

- **Variable Groups**: Understand what variable groups are and why they are used.
- **Creating and Managing Groups**: Know how to create, configure, and manage variable groups.
- **Linking to Pipelines**: Be familiar with how to link variable groups to pipelines and use variables in pipeline configurations.
- **Secrets Management**: Understand how to manage sensitive data using secret variables in Azure DevOps.

Azure Key Vault and Accessing Secrets in Pipelines

Azure Key Vault is a cloud service that helps securely store and access secrets, such as API keys, passwords, and certificates. It provides a centralized place to manage sensitive information and integrates well with Azure DevOps for secure secret management in CI/CD pipelines.

Overview of Azure Key Vault

- **Purpose**: To safeguard secrets, keys, and certificates used by cloud applications and services.
- **Features**: Secure storage, access policies, logging, and integration with other Azure services.

Creating and Configuring an Azure Key Vault

1. **Creating a Key Vault**
    - **Azure Portal**:
        - Navigate to the **Azure Portal**.
        - Search for **Key Vault** and select **Create**.
        - Enter the **Name**, **Subscription**, **Resource Group**, and **Location**.
        - Click **Review + create** and then **Create**.

Example:

- **Name**: MyKeyVault
- **Resource Group**: MyResourceGroup
- **Location**: East US

2. **Adding Secrets to the Key Vault**
    - Go to your newly created Key Vault in the Azure Portal.
    - Navigate to **Secrets** under the **Settings** section.
    - Click on **+ Generate/Import** to add a new secret.
    - Enter a **Name** and **Value** for the secret.

Example:

- **Secret Name**: MyDatabasePassword
- **Value**: P@ssw0rd1234

3. **Configuring Access Policies**
    - Go to **Access policies** in the Key Vault settings.
    - Click **+ Add Access Policy**.
    - Select the appropriate **Secret permissions** (e.g., **Get**, **List**).
    - Assign the policy to a **Service Principal** or **Azure DevOps** to allow access from your pipeline.

Example:

- **Secret permissions**: Get, List
- **Principal**: MyAzureDevOpsServicePrincipal

Accessing Azure Key Vault Secrets in Azure DevOps Pipelines

1. **Setting Up the Service Connection**
    - **Azure DevOps Portal**:
        - Navigate to your Azure DevOps project.
        - Go to **Project settings** > **Service connections**.
        - Click **New service connection** and select **Azure Resource Manager**.
        - Authenticate using the appropriate method (e.g., Service Principal).
        - Select the subscription and the Key Vault to which you want to grant access.

2. **Using Key Vault Secrets in Pipeline YAML**
    - To use secrets from Azure Key Vault, update your pipeline YAML file to reference the Key Vault.

Example YAML Configuration:

```
variables:
  - group: ProductionSecrets


stages:
  - stage: Build
    jobs:
      - job: BuildJob
        steps:
          - task: AzureKeyVault@2
            inputs:
              azureSubscription: 'MyAzureServiceConnection'
              KeyVaultName: 'MyKeyVault'
              SecretsFilter: '*'
          - script: |
              echo $(MyDatabasePassword)
            displayName: 'Print secret'
```

    - **AzureKeyVault@2**: This task fetches secrets from the specified Key Vault.

- **azureSubscription**: Service connection name for Azure.
- **KeyVaultName**: The name of your Key Vault.
- **SecretsFilter**: Filters secrets to be fetched (use '*' to fetch all).

3. **Accessing Secrets in Pipeline Tasks**
    - Once fetched, secrets can be accessed as pipeline variables.
    - They can be used directly in scripts and tasks.

Example Usage in a Script:

```
- script: |
    echo $(MyDatabasePassword)
  displayName: 'Display database password'
```

- **$(MyDatabasePassword)**: Accesses the secret value from the Key Vault.

Key Points for Interview Preparation

- **Azure Key Vault**: Understand the purpose, features, and configuration of Azure Key Vault.
- **Creating Secrets**: Know how to add and manage secrets in Key Vault.
- **Access Policies**: Be familiar with configuring access policies for secure access.
- **Service Connections**: Understand how to set up Azure DevOps service connections to Azure.
- **Pipeline Integration**: Know how to access and use Key Vault secrets in Azure DevOps pipelines.