RealTime Handson

25

DevOps & Azure Cloud Interview Questions

Shruthi Chikkela
learnwithshruthi

Terraform

1. Managing the Statefile in Pipelines

Terraform uses a statefile to track real-world resources and map them to your configuration. In production environments, it is best practice to store the statefile remotely with locking enabled.

For example, you might configure a remote backend such as an Azure Storage Account, AWS S3 (often with DynamoDB for locking), or HashiCorp's Terraform Cloud. In a CI/CD pipeline, you would:

- **Configure the backend** in your Terraform configuration so that state is stored securely off the build agent.
- **Ensure proper access controls** are set up so that the pipeline account has the necessary credentials to read/write the state.
- **Lock state during operations** to prevent concurrent changes.
- **Integrate state management tasks** (like `terraform init`, `plan`, and `apply`) into your pipeline YAML definitions, ensuring that error handling and rollbacks are configured appropriately.

2. Managing Resources Created Outside Terraform

When you have resources that were created manually (e.g., via a cloud provider's portal) but you want to manage them with Terraform, you use the **terraform import** command.

This command brings the existing resource into your Terraform state file. The steps are:

- Write the resource configuration in your Terraform files as if you were creating it.
- Run `terraform import <resource_type>.<resource_name> <resource_identifier>` to associate the existing resource with that configuration.
- After import, run `terraform plan` to verify that no further changes are pending, ensuring that Terraform now acknowledges and manages the external resource.

3. Differences Between Terraform Init and Refresh

- **Terraform Init**: This command initializes your working directory.
- It downloads provider plugins, sets up the backend configuration, and prepares your Terraform environment for subsequent commands.
- **Terraform Refresh**: This command updates the state file with the latest real-world resource status. It ensures that the state reflects any manual changes or drift by querying the current infrastructure.
  While `init` is setup-related, `refresh` ensures state accuracy and can be particularly useful before planning or applying changes.

4. Key Terraform Commands

Common commands you might use are:

- **`terraform init`**: Initializes a new or existing configuration.
- **`terraform plan`**: Creates an execution plan, showing which actions Terraform will take to reach the desired state.
- **`terraform apply`**: Applies the changes required to reach the desired state of the configuration.
- **`terraform destroy`**: Destroys the managed infrastructure.
- **`terraform validate`**: Validates the configuration syntax.
- **`terraform fmt`**: Formats the configuration files to a canonical style.
- **`terraform import`**: Imports external resources into the Terraform state.
- **`terraform output`**: Extracts output variables from the state file.

5. Terraform Modules: Functionality and Benefits

Terraform modules are containers for multiple resources that are used together. They allow you to:

- **Encapsulate configuration** for reusability, which makes it easier to manage repeated patterns.
- **Organize code** by separating concerns and promoting modular design.
- **Standardize resource creation** across environments, which results in more consistent deployments.
- **Share and version modules** internally and externally, improving collaboration and reducing duplicated efforts in infrastructure management.

DevOps & Pipelines

1. Branching Strategy and Environment Deployment

A well-defined branching strategy is crucial for managing code and deployments. Common strategies include:

- **Gitflow**: Uses feature branches, develop and master branches, and supports parallel development.
- **Trunk-based Development**: Encourages frequent integration with a single main branch. Deployment to different environments is often managed by:

- Mapping branches to environments (e.g., feature branches to dev, integration branches to staging, and the main branch to production).
- Using separate pipelines or environments in your CI/CD tool to automate builds, tests, and deployments across different stages.

2. Integrating Azure DevOps / GitHub Actions with Azure

Both Azure DevOps and GitHub Actions can integrate with Azure through:

- **Service connections (Azure DevOps)** or **service principals (GitHub Actions)**.
- Using tasks such as the Azure CLI, ARM deployment tasks, or pre-built GitHub Actions to interact with Azure resources.
- Establishing secure authentication and authorization measures so that pipelines can deploy infrastructure and applications directly into Azure.

3. Pipeline Structure: Steps and Stages

A well-organized pipeline typically includes:

- **Stages**: Logical groupings such as build, test, staging, and production deployments.
- **Jobs**: Parallelizable units within stages, often tailored to specific tasks.
- **Steps/Tasks**: Individual commands or scripts that perform actions (e.g., compiling code, running tests, deploying artifacts).
  Each pipeline is written using YAML (or a visual designer) where you specify the sequence, dependencies, environment variables, and conditions under which certain jobs execute.

4. Handling Secrets and Environment Variables

Secrets should never be hard-coded in your pipeline or code. Best practices include:

- Using secured secret stores like **Azure Key Vault**, or secret management provided by your CI/CD system (e.g., GitHub Secrets, Azure Pipelines Library).
- Injecting secrets as environment variables at runtime.
- Using RBAC and encryption to ensure that only authorized processes and users have access to these secrets.

5. Configuring Pipeline Approvals for Higher Environments

For deployments to staging or production, manual approvals ensure quality control and risk mitigation. You can configure:

- **Pre-deployment gates** that pause the pipeline until an authorized person approves the change.
- **Release pipelines** with explicit approval steps that trigger deployments only after validation checks are passed.

- **Notifications and audit trails** so every approval is tracked and logged.

6. Azure Boards Familiarity

Azure Boards is an agile project management tool integrated with Azure DevOps. It provides:

- **Backlogs, Kanban boards, and sprint planning tools** to manage work.
- **Customizable work item tracking** to capture bugs, features, and tasks.
- This tool helps teams to plan, track, and discuss work across the entire development cycle.

7. Experience with Ticketing Systems

In many DevOps environments, ticketing systems such as **Jira**, **ServiceNow**, or Azure Boards are used to manage work. Experience with these systems involves:

- Creating, updating, and tracking tickets.
- Integrating issue trackers with source code management and CI/CD pipelines to maintain traceability from commit to deployment.
- Collaborating across teams to ensure accountability and continuous delivery.

Kubernetes

1. Application Deployment Using Kubernetes

Applications in Kubernetes are typically deployed using declarative manifests (YAML files) such as Deployments, Services, and ConfigMaps. This can be done via:

- **kubectl apply** for creating or updating resources.
- Using package managers like **Helm** to template and manage complex deployments.
- Leveraging CI/CD pipelines to automatically deploy and update Kubernetes resources.

2. What is a Deployment?

A Deployment in Kubernetes provides a declarative way to manage application updates. It:

- **Manages ReplicaSets**, ensuring that a specified number of pod replicas are running at any given time.
- Supports **rolling updates and rollbacks**, allowing seamless upgrades or reverting to a previous version if needed.
- Helps maintain high availability and fault tolerance of the application.

3. How Ingress Works

Kubernetes Ingress acts as a layer 7 (HTTP/HTTPS) load balancer. It:

- Routes external traffic to your services based on defined rules.
- Provides features such as **TLS/SSL termination**, URL-based routing, and host-based routing.
- Works in conjunction with an Ingress Controller (like Nginx or Traefik) that enforces the rules and configurations.

4. Networking and Configuration in Previous Projects

Networking in Kubernetes can involve a variety of configurations:

- **Service types** such as ClusterIP, NodePort, and LoadBalancer.
- **Network Policies** to control traffic flow between pods.
- Configuring **Ingress rules and controllers** for external access.
- In some projects, integration with service meshes (e.g., Istio) has been implemented for advanced traffic management and observability.

5. Storing Secrets in Kubernetes

Kubernetes provides a native way to handle secrets via **Secret objects**. Best practices include:

- Storing sensitive data (passwords, tokens, keys) in these objects.
- Enabling encryption at rest for the secret data in the etcd cluster.
- Limiting access using RBAC to ensure only the necessary pods or services have access.

6. Monitoring Pod Failures and Viewing Logs

When a pod fails, several tools and commands come into play:

- Use `kubectl get pods` to identify pods not in a running state.
- Run `kubectl describe pod <pod-name>` for event and error details.
- Check logs using `kubectl logs <pod-name>` to troubleshoot the specific error.
- Often, centralized logging solutions (like ELK or Fluentd) and monitoring systems (like Prometheus/Grafana) are set up for real-time alerting and historical analysis.

Azure Services

1. Working on an ADF Project

An ADF (Azure Data Factory) project involves orchestrating data workflows across various data sources. A typical configuration includes:

- Defining **pipelines** that orchestrate the data movement and transformation processes.
- Using **activities** that perform data extraction, transformation, and loading (ETL).
- Integrating with both on-premises and cloud data sources through self-hosted or Azure integration runtimes.

2. ADF Data Flow Sources

ADF pipelines may involve both self-hosted and Azure cloud sources.

- **Self-hosted Integration Runtime (SHIR)** allows connecting to on-premises data stores.
- The **Azure Integration Runtime** facilitates data movement and transformation using cloud-based compute resources.
  The selection depends on where the source data resides and the security requirements.

3. Datasets vs. Linked Services

- **Linked Services**: These are connection strings or configuration details required for ADF to connect to a data source (e.g., a SQL Server, Blob Storage, etc.).
- **Datasets**: These describe the data structure and location within the data source that ADF will work with. In short, linked services provide connectivity, while datasets provide the schema and details about the actual data.

4. Parameters in Pipeline Configuration

Parameters in ADF pipelines improve reusability and flexibility. For instance:

- **Pipeline Parameters** allow you to pass different values into pipelines dynamically at runtime.
- They can be used in activities, linked services, and datasets, enabling a single pipeline definition to work across multiple scenarios.
- The output of parameterized runs can be fed into subsequent activities for further processing or decision-making.

5. Secure Access to Key Vault Secrets

Azure Key Vault is the recommended way to store and manage sensitive configuration details. To access key vault secrets securely:

- Use **Managed Identities** to give Azure services identity and permission to access the Key Vault without hardcoding credentials.
- Reference key vault secrets in your pipelines or resource configurations so that they are pulled in securely at runtime.

6. Prioritizing High Availability, Security, and Scalability

While all three elements are critical, many organizations prioritize:

- **High availability** to ensure continuous service.
- **Security** to protect data and comply with regulations.
- **Scalability** to handle variable loads.
  A balanced approach involves designing distributed systems with redundancy, proper network segmentation (e.g., NSGs, firewalls), and auto-scaling strategies to dynamically adjust resources as needed.

7. Designing a Highly Available Web App

For a highly available web app, Azure App Service is a common choice. It offers:

- Built-in auto-scaling
- Regional redundancy
- Integration with Azure Front Door or Traffic Manager to route users to the healthiest available endpoint
- Managed security and simplified deployment pipelines

8. Secure Access to a SQL Database from a Web App

A secure setup typically involves:

- Configuring **firewall rules** or using **private endpoints** to restrict access.
- Enabling **Managed Identity Authentication** so that the web app can securely connect to SQL without storing credentials.
- Employing encryption (both at rest and in transit) and auditing to ensure data security.

9. Secure Networking for Multi-Regional Web App Access

To securely expose a web app across multiple regions, you could use:

- **Azure Front Door** or **Traffic Manager** to balance traffic globally.

- **VPNs or private connections** (like ExpressRoute) in conjunction with network security groups for granular control.
- Proper endpoint configurations and DNS management to ensure connectivity while enforcing security policies.

10. Understanding Managed Identities

Managed Identities provide an automatically managed identity for Azure resources. They work by:

- Allowing services to authenticate to Azure Key Vault, SQL databases, and other resources without managing credentials explicitly.
- Being assigned through Azure RBAC so that they only get the exact permissions required.

11. Enforcing Least Privilege Access

Using Azure's RBAC, you can assign roles like the **Reader** role to limit permissions. This ensures that a resource (or identity) can only view resources without making any modifications. You can scope these roles to a resource group, subscription, or individual resource level.

12. How Network Security Groups (NSGs) Work

NSGs are used to filter network traffic to and from Azure resources. They function by:

- Defining inbound and outbound rules based on source/destination IP addresses, ports, and protocols.
- Being associated with subnets or individual network interfaces, thereby providing granular control over the traffic flow.

13. Creating an AKS Cluster

Creating an Azure Kubernetes Service (AKS) cluster typically involves:

- Using the **Azure CLI**, ARM templates, Terraform, or the Azure Portal to configure parameters like node count, node size, and Kubernetes version.
- Setting up integration with Azure Active Directory for authentication and RBAC for authorization.
- Configuring additional features, such as network policies, monitoring (using Azure Monitor), and auto-scaling.
- Following security best practices, including private cluster configuration and integrating with Azure Key Vault for secret management.

For any RealTime Handson Projects

And for more tips like this

Follow

Shruthi Chikkela

Like & ReShare

learnwithshruthi