Azure App Service with Terraform and Node.js

1. Solution Overview

This solution provisions an Azure Linux Web App running a Node.js application that dynamically displays a string from Azure App Configuration. The entire solution is automated using Terraform, ensuring that the infrastructure is defined as code.

Key Components:

- **Azure Resource Group**: Contains all cloud resources.
- **Azure App Service Plan**: Provides the hosting environment for the Node.js application.
- **Azure Linux Web App**: Runs the Node.js application.
- **Azure App Configuration**: Manages dynamic configuration values (like saved_string).
- **Terraform**: Manages the Infrastructure as Code (IaC).

2. Design Decisions

Why Terraform?

- Terraform provides a declarative, repeatable, and scalable way to provision cloud infrastructure.
- The Infrastructure as Code (IaC) approach allows for version control and automation.
- I chose Terraform because it meets the job requirements. Arqiva is seeking a developer with Terraform experience.

Why Azure App Configuration?

- Provides centralized management of configuration settings.

- The saved_string value can be dynamically updated without redeploying the Node.js app.

Why Azure App Service (Linux)?

- Supports easy deployment of Node.js applications.
- Provides a scalable and secure hosting environment.
- Managed service with automated scaling and monitoring.

3. Available Options Considered

Option 1: Azure App Service with Node.js and Azure App Configuration (Chosen)

- Pros:
  - Secure and scalable.
  - Supports dynamic configuration.
  - Easy to automate with Terraform.
- Cons:
  - Dependency on Azure App Configuration availability.

Option 2: Node.js Application with Configuration Hardcoded

- Pros:
  - Simpler setup.
  - No dependency on Azure App Configuration.
- Cons:
  - Static configuration requires redeployment for changes.
  - Less secure.

Option 3: Azure Key Vault for Secure Configuration (Future Enhancement)

- Pros:
  - Secure secret management.
  - Centralized secret control.
- Cons:
  - More complex setup.

4. Design Considerations and Best Practices

- Used System-Assigned Managed Identity for secure access to Azure App Configuration.
- Used a customizable saved_string value in Terraform for flexibility.
- Node.js app is enhanced for resilient operation with detailed logging.

5. How the Solution Can Be Enhanced

- Add Azure Key Vault for securely storing sensitive settings.
- Implement automatic scaling based on app load.
- Add monitoring and alerting using Azure Monitor.
- Use a CI/CD pipeline for automated deployment and updates.

6. Conclusion

This solution provides a robust and scalable architecture for running a Node.js application on Azure with dynamic configuration management. It leverages Terraform for full automation, ensuring reproducibility and ease of management.