Azure Node.js App with Terraform

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?

- Provides a declarative, repeatable, and scalable way to provision cloud infrastructure.
- Enables version control and automation through IaC.
- Works across multiple cloud providers, offering greater flexibility and portability compared to tools like ARM templates.
- Chosen because Arqiva seeks developers with Terraform expertise.

Why Azure App Configuration?

- Centralized management of configuration settings.
- The saved_string value can be updated dynamically without needing to redeploy the Node.js app.

Why Azure App Service (Linux)?

- Native support for Node.js deployments.
- Scalable and secure hosting environment.
- Managed service with automated patching, 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 Hardcoded Configuration

Pros:

- Simpler setup.
- No dependency on external configuration services.

Cons:

- Static configuration requires redeployment for any change.
- Reduced flexibility and security.

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

Pros:

- Secure secret management.
- Centralized control over sensitive data.

Cons:

- Adds complexity to setup and maintenance.

4. Design Considerations and Best Practices

- Used **System-Assigned Managed Identity** for secure access to Azure App Configuration.
- Allowed a customizable saved_string value through Terraform variables for flexibility.
- App is designed to retrieve dynamic settings and can include retry logic.

Performance and Scalability:

- **Can it scale?**
    - Azure App Service Plans can be scaled up or out to handle increased load.
    - Azure App Configuration could become a bottleneck under very high throughput. Caching or replication (e.g., using Redis or a traditional database) can help mitigate this.

Reliability:

- **Is it reliable?**
    - Azure App Configuration has a 99.99% SLA. This still allows for brief unavailability (~8 seconds/day).
    - To handle this:
        - Implement **retry logic** in the Node.js app.
        - Use **failover caching** or database replication to serve stale data if needed.

5. How the Solution Can Be Enhanced

- Add **Azure Key Vault** to manage secrets and credentials securely.
- Enable **automatic scaling** based on CPU/memory metrics.
- Integrate **monitoring and alerts** using Azure Monitor and Application Insights.
- Add **logging** to track application behavior and errors:
    - Helps detect issues before users report them.

    - Supports alerting based on log severity.
- Set up a **CI/CD pipeline** using GitHub Actions or Azure DevOps for automated testing and deployment.

6. Conclusion

This solution provides a robust, scalable architecture for deploying a Node.js application on Azure with dynamic configuration management. It leverages Terraform for full automation and reproducibility, and lays a strong foundation for enhancements like monitoring, security, and scalability.

By combining modern cloud-native services with infrastructure as code, this architecture meets both development and operational needs effectively.