Status Codes API

Anush Shankar
anush_shankar@outlook.com

eppendorf

Contents

1 Overview **3**
1.1 Features of the application . . . 3

2 Prerequisites **4**

3 Project Structure **5**

4 Solution **6**
4.1 Requirements of the Project . . . 6
4.2 Implementation . . . 6
4.3 Remote Test . . . 7

5 Quality of the Solution **8**
5.1 Static Analysis . . . 8
5.2 Functionality Check . . . 8
5.3 Performance Check . . . 8
5.4 GitHub Workflow . . . 8
5.5 Security . . . 8

6 Challenges **9**
6.1 Scalability and Performance . . . 9
6.2 Security . . . 9
6.3 Reliable and Highly Available . . . 9
6.4 Managing Configurations and Tool Updates . . . 9

7 Conclusion **10**

1 Overview

This documentation provides an overview guide to deploying a serverless application named "Status Codes" on AWS. The application exposes a RESTful API that returns a random HTTP status code from a predefined set. The deployment includes infrastructure setup using Terraform, CI/CD pipeline configuration with GitHub Actions, and necessary scripts for testing and monitoring.

1.1 Features of the application

- The API is exposed via an API Gateway
  - An API key is generated and used for authentication
- 180 GET requests per 60 secs
- A Cloudwatch alarm informs the owner via mail, if the API responds with more than 10 5xx status codes within 10 secs
- Infrastructure is deployed on AWS using Terraform
- GitHub actions are used for CI-CD. It builds, tests and deploys the infrastructure and API to AWS

2 Prerequisites

Please ensure that you have installed the following

- Git
- GitHub Account
- Python - 3.9
- boto3: AWS SDK for Python
- pytest: Performs unit and integration tests
- pylint: Performs code quality check
- AWS CLI: Client application to interact with AWS
- AWS Account
- Terraform: IaaS

3 Project Structure

```
statusCodes/
            src/
                  functions/
                        get_status_code.py
                  libs/
                        common.py
                  tests/
                        test_unit.py
                        test_integration.py
            terraform/
                  main.tf
                  variables.tf
                  outputs.tf
                  provider.tf
            .github/
                  workflows/
                       ci-cd.yml
            README.md
```

4 Solution

4.1 Requirements of the Project

- RESTful API: I need to expose a RESTful API the returns random HTTP status codes
- Event-Driven: The API is triggered by HTTP GET requests
- Scalable: The application needs to handle up to 180 GET requests per 60 seconds
- Monitoring: Alerts should be created for if the API responds more than 10 5xx status codes in 10 seconds

AWS Lambda functions suits my requirements. It is **serverless** and eliminates the need to provision or manage servers. Lambda can be integrated with API Gateway, which handles the HTTP requests. The API Gateway invokes the lambda function, making it **event-driven**. It automatically **scales** out to handle the incoming requests, hence can respond to up to 180 GET requests per minute. Lambda integrates with CloudWatch to provide **monitoring**. I can set up CloudWatch alarms to notify me if there are more than 10 5xx status codes within 10 seconds.

4.2 Implementation

1. Create a Python function (Lambda function) that returns a random HTTP status code
2. Set up an API Gateway to expose the Lambda function as an API endpoint
3. Configure the API Gateway to require an API key for authentication
4. Terraform to define and provision my infrastructure. The infrastructure includes the following
    - Lambda function
    - API Gateway : REST API, API key, method, integration, usage plan, deployment
    - IAM : roles, policy, policy attachment,
    - SNS Topic : subscription
    - CloudWatch : alarm

The following sensitive data are stored securely using GitHub secrets.

- API_KEY
- API_URL
- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_REGION

To run the application locally, one has to set the $PYTHONPATH environment variable to include the ./src directory

- Add the following to /.bashrc or /.zshrc file
  - export PYTHONPATH=”$PYTHONPATH:/path/to/project-root/src”
- Reload your /.zshrc file
  - source /.zshrc

4.3 Remote Test

Execute cURL test to ensure the integration of API Gateway with the Lambda function. Please find the below snapshot depicting the response from the deployed application. The application returns random status codes.

5 Quality of the Solution

To ensure the quality of the solution, I have incorporated the following techniques.

5.1 Static Analysis

- Pylint ensures the code quality and its adherence to Python coding standards
- One of the GitHub actions installs Pylint, executes over the src folder of the project

5.2 Functionality Check

- Unit Tests: To check the logical units of the code - functions
- Integration Tests: Testing integration of different parts of the application - API Gateway with Lambda function.

5.3 Performance Check

- Monitoring: CloudWatch alarms to monitor the performance and health of Lambda function

5.4 GitHub Workflow

- Automated Testing: Tests are run automatically in the CI/CD workflow on Git

5.5 Security

- Managing Secrets: Sensitive information is managed securely using GitHub Secrets.

6 Challenges

This section focuses on the challenges that arouse while deploying the application in the cloud environment.

6.1 Scalability and Performance

- Ensuring that the application can handle varying loads with a burst limit of 180 requests per minute. Monitoring requests and responses via CloudWatch

6.2 Security

- Established IAM roles, security groups and encryption of sensitive data, ensuring security standards

6.3 Reliable and Highly Available

- To ensure that the application is highly available, the application is ready to be deployed across multiple regions and availability zones

6.4 Managing Configurations and Tool Updates

- Terraform is used to manage application configurations and dependency updates in timely manner. Cloud resources are managed to avoid under or over provisioning

7 Conclusion

In conclusion, I have a few suggestions that could be undertaken as future work.

- Simulating the incoming load to ensure that the API can handle expected traffic using tools like 'Locust'.
- Avoiding vendor lock-in. Applications to be designed not to have dependency on a single cloud provider.