Platform Engineering

High level Design

Contents

Feature 1: Kubernetes Cluster Provisioning REST API Service

Introduction:

Kubernetes Cluster Provisioning is developed in SpringBoot- A Java based Web Framework that serves REST APIs for the internal developer platform. It provides a kubernetes Environment by setting up the required kubernetes configuration according to the demand of the user.

It has following major features:

1. Provision kubernetes Environment.
2. Destroy kubernetes Environment.
3. Number of nodes is configured based on users requirement
4. Instance size & type can also be configured based on requirements
5. Instance size & type can be increased if required at any point of time

For all this features, this Kubernetes Cluster service can be used. In Java, for segregating the functionality and making the code more maintainable and readable , 3 different web controllers (named cluster controller , jenkins controller , database controller) in one spring boot project is developed.

1. cluster controller: This controller used to provision/destroy/update the kubernetes cluster.

2. jenkins controller: This controller is used to trigger the respective jenkins pipeline to manage cloud infrastructure

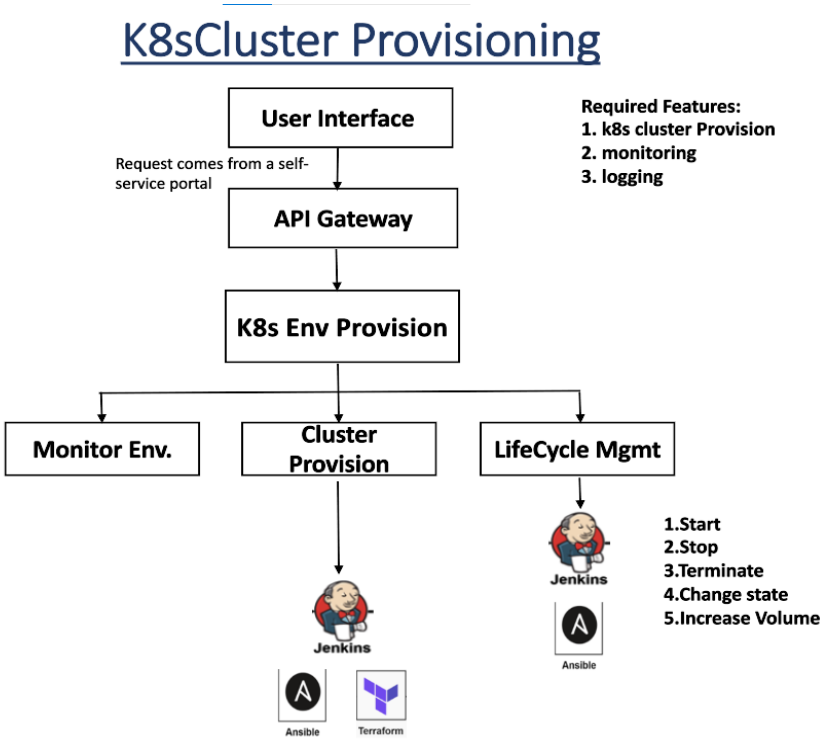
3. database controller: This controller is used to manage & update database changes.

It also supports various functionalities like provisioning, deprovisioning, lifecycle management and logging for the infrastructure which it is provisioning. In Lifecycle management, it is supporting starting and stopping of the environment , changing the instance type of the environment, increasing the storage for the environment. It also provides apis to get the details of the cluster which is provisioned & configuration files to connect with the cluster remotely .

API Documentation:

API Documentation can be found in GET method of swagger/ endpoint of the application.

Architecture:



Feature 2. Sandbox Environment Provisioning API Service

Introduction:

Sandbox Environment Provisioning is developed in Django - A Python based Web Framework that serves REST APIs for the internal developer platform. It provides Environment by setting up the required tools and libraries according to the demand of the user.

It has 5 major offerings:

6. Provision Infrastructure along with tools.
7. Provides Environment on the existing infrastructure.
8. Provide Big Data Cluster as a Service.
9. Provide Bigtop as a Service with the tools specified by the user.
10. VDE as a Service

For all those offerings, this Sandbox Provisioning service can be used. In Django, for segregating the functionality and making the code more maintainable and readable , 3 different apps (named sandboxApp, bigDataCluster, bigtop) in one django project (sandbox) has been developed.

1. sandboxApp: This django app's services can be used to provision the new infrastructure (virtual machines, security groups for the machines, volumes, etc.) alongs with the tools. It can also be used to setup an environment on the exisiting machines.

Tools Offered: 'AWScli2', 'Ansible', 'Django', 'Docker', 'Gerrit', 'GitLab', 'Grafana', 'Java', 'Jenkins', 'JupyterNotebook', 'Kafka', 'Keras', 'Maven', 'Projectlibre', 'MySQL', 'Nexus', 'PostgreSQL', 'Python3', 'MongoDB', 'Prometheus'.

It also supports various functionalities like provisioning, deprovisioning, lifecycle management and logging for the infrastructure which it is provisioning. In Lifecycle management, it is supporting starting and stopping of the environment , changing the instance type of the environment, increasing the storage for the environment. It also provides VDE for the user to access common services provided within the existing environment.

2. **bigDataCluster**: This sandbox's app can be used to provision big data cluster on the kubernetes environment. Currently Provisioning and deprovisioning of Big data cluster is supported using helm charts. Using EFS, it is also able to handle the creation of PV/PVC.

Tools Offered: 'Kafka', 'Spark', 'Grafana', 'Postgres', 'Mysql'.

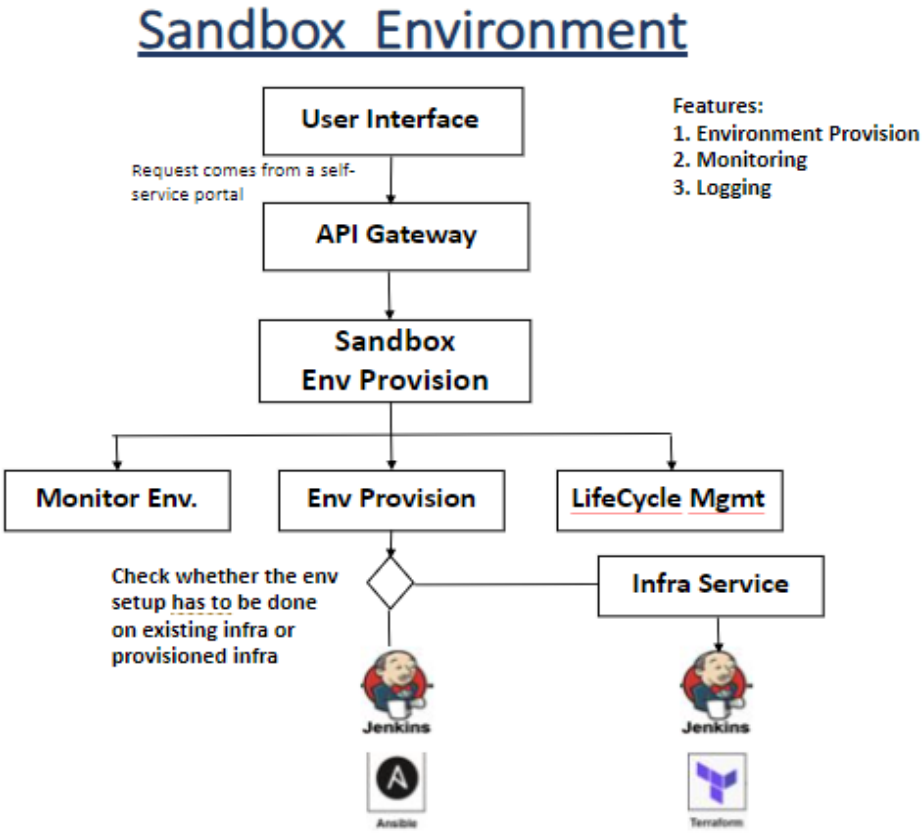
3. **bigtop**: Bigtop is an open source project that provides a framework for packaging, testing and configuring the Apache Hadoop Ecosystem. It provides cluster with bigdata tools required by the user using bigtop package. This sandbox's app can be used to provision big data cluster on kubernetes and docker environment. Currently Provisioning and deprovisioning of Big data cluster using Bigtop is supported. Bigtop service can be used in both docker and kubernetes platform. For Docker setup which

it is providing, It will actually build the image by using the configuration, based on the user needs. For Kubernetes Setup, it will use the existing docker images provided by official Bigtop Community.

Tools Offered: Hadoop**,** HDFS, Yarn, Mapreduce, Hive, Pig, Spark, HBase.

The tools installed are verified by testing the all the functionalities of the various tools. Multiple cluster support is also present within the same environment.

Architecture:



API Documentation:

API Documentation can be found in GET method of swagger/ endpoint of the application.

Feature 3: Database Provisioning REST API Service

Database Provisioning is developed in SpringBoot- A Java based Web Framework that serves REST APIs for the internal developer platform. It provides a SQL or NO-SQL Server by setting up the required db configuration according to the demand of the user.

It has following major features:

1. Provision Postges or MongoDB Server.
2. De-provision the Server.
3. Create Users in database.
4. Give permissions to the users.
5. Create Databases in the Server.
6. It provides 2 modes for a server which will be provisioned Exclusive & shared mode.

For all this features, this Database Provisioning service can be used. In Java, for segregating the functionality and making the code more maintainable and readable , 2 different web controllers (named jenkins controller , database controller) in one spring boot project is developed.

1. jenkins controller: This controller used to provision/destroy/update the database Server , users , and their db permissions.

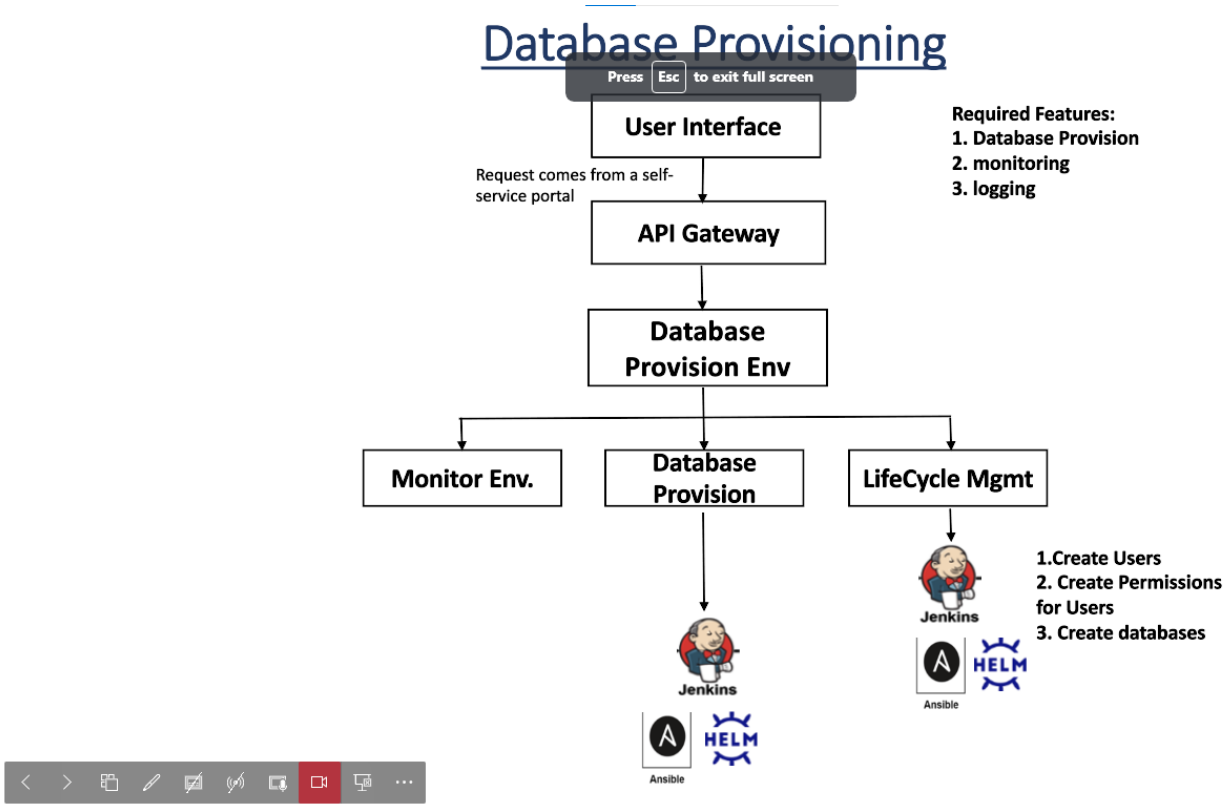
2. database controller: This controller is used to manage & update database changes.

It also supports various functionalities like provisioning, deprovisioning, lifecycle management and logging for the server which it is provisioned. In Lifecycle management, it is supporting creating new database, users of the database, changing permissions of the database users. It also provides apis to get the details of the server which is provisioned. A server can be provisioned in 2 modes, in exclusive & shared mode , in exclusive mode only one user (Admin user) can be created for a database. And for the shared mode any number of users can be created & each user can be provided with different set of permissions.

API Documentation:

API Documentation can be found in GET method of swagger/ endpoint of the application.

Architecture:



Feature 4: Vulnerability Scanning in Docker images REST Server

Vulnerability Scanning is developed in SpringBoot- A Java based Web Framework that serves REST APIs for the internal developer platform. It provides service to scan docker images of a public & private docker registries , It also list all types of possible vulneralibities in the docker image & creates a pdf which is downloadable.

It has following major features:

1. Scan a docker image from Private or Public docker registry.
2. Provides all possible vulnerabilites in the docker image
3. Provides a url to access & download the vulnerability report..

For all this features, this Vulnerability Scanning service can be used. In Java, for segregating the functionality and making the code more maintainable and readable , 2 different web controllers (named jenkins controller , database controller) in one spring boot project is developed.

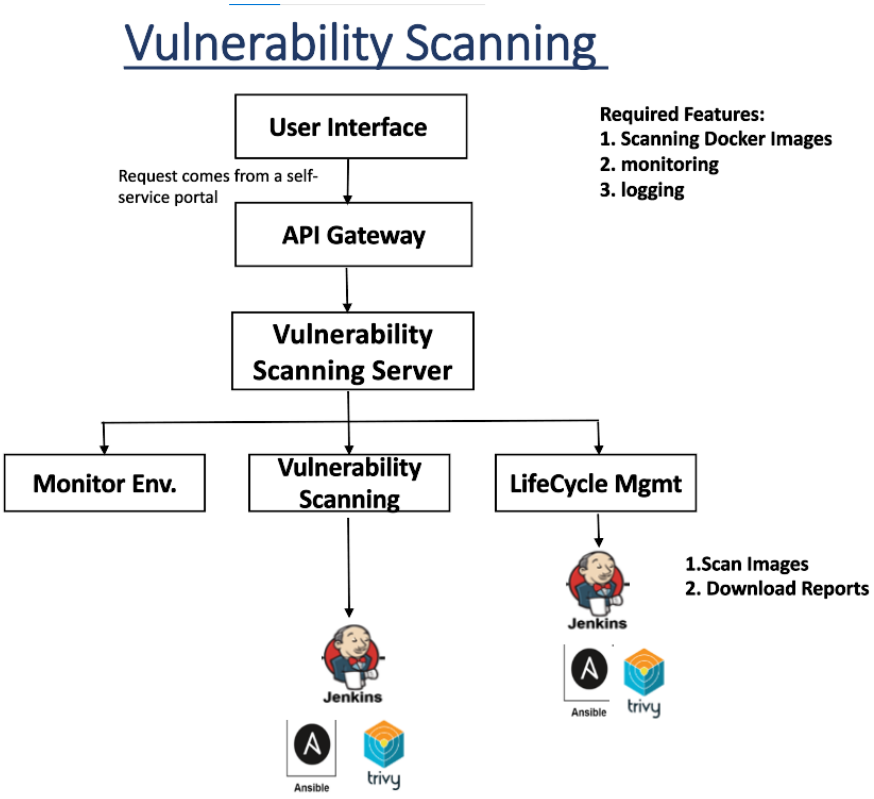
1. jenkins controller: This controller used to Scan docker images & generate reports.

2. database controller: This controller is used to manage & update database changes.

API Documentation:

API Documentation can be found in GET method of swagger/ endpoint of the application.

Architecture:



Feature 5. Template Management Bankend API Service

Introduction

Template Management Service has been developed to manage the complex deployment structure for various use-cases in an effective manner and to ensure the reusability of Template parameter values for multiple use-cases.

Template Management Backend service is developed in Django - A Python based Web Framework that serves REST APIs for the internal developer platform. It provides a Backend service for Template Management UI that will deploy template paramaters for the respective Use-cases and provision the resources through the template UI only. Template Management is basically a central point from where we can provision resource or can manage the provisioned resources.

Database Details

- template_table – All the template details get store in this table. At the time of Template Creation , there is a associated service which maps a particular template to a use-case.
- Request_table – All the template deployment requests get store in this table. Whenever user initiates Template Deployment, the request gets stored in the Request Table with the Pending status. Then once the request is approved by the Admin, then the request resource will get provisioned.
- Template_table(template_id) is used as a Foreign Key in Request_table(template_id).

Template Backend Service supports CRUD operations for Template and Request Table , also accomplish the corresponding functionality along with that.Template Backend Service also provides support for serving Service Details, Available Instance Types, Number of nodes, Regions to the UI while selecting the Template parameters for deployment.

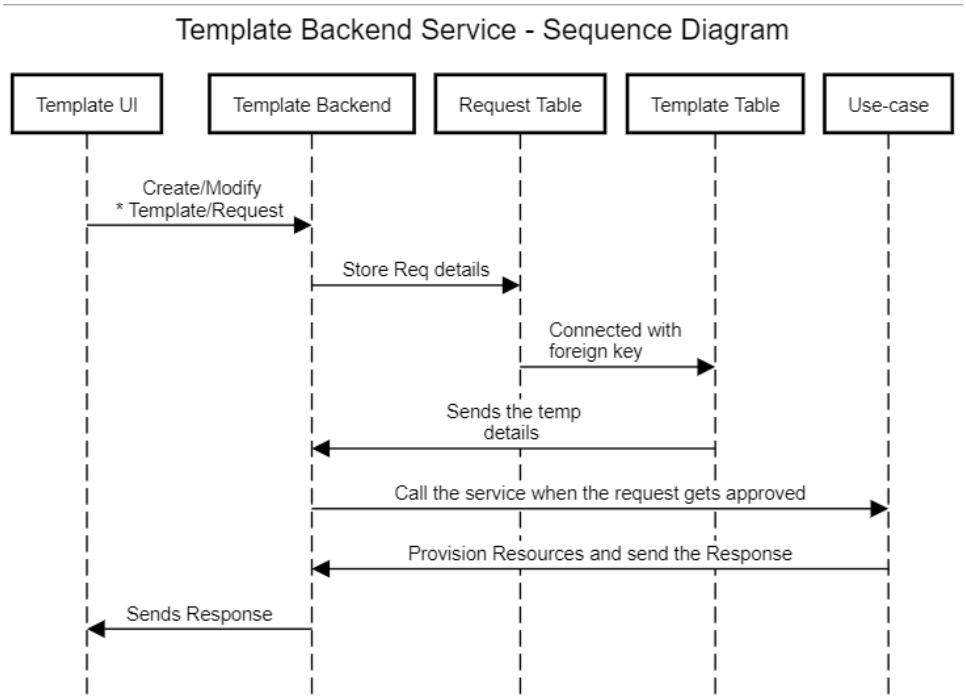
Feature 812 of 45

Template Backend Service is tested for Cluster Provisioning and Sandbox Provisioning.

API Documentation:

API Documentation can be found in GET method of swagger/ endpoint of the application.

Sequence Diagram



Feature 6: Gitlab Admin Activities REST API Service

Gitlab API Service is a Spring Boot-based application that serves REST APIs for the admin to configure resources on Gitlab

The API Server provides the following endpoints:

1) Create Repo:

Method: POST

Endpoint: /git/createrepo

Request Params: NA

Request Body: (type = JSON)

{

"name": "platformenv",

"private": "false"

}

Request Header: { Content-Type: “application/json” }

2) Create User:

Method: POST

Endpoint: / git/createUser

Request Params:NA

Request Body:

{

"username": "demouser",

"name": "demouser",

"email": "demouser@gmail.com",

"password": "admin@12345",

}

Request Header: { Content-Type: “application/json” }

3) Create Group

Method: POST

Endpoint:/git/createGroup

Request Params:NA

Request Body:

{

"name": "demoGroup",

"path": "demoGroup",

"visibility": "public"

}

Request Header: { Content-Type: “application/json” }

4) Add user to Project:

Method: POST

Endpoint: /git/addUserToProject

Request Params:NA

Request Body:

{

"user_id": "2",

"projectid": "1",

"access_level": "30"

}

Request Header: { Content-Type: “application/json” }

5) Change access of a user:

Method: POST

Endpoint: /git/changeAccessOfProjectUser

Request Params: NA

Request Body:

{

"user_id": "2",

"projectid": "1",

"access_level": "20"

}

Request Header: { Content-Type: “application/json” }

6) Assign Reviewer to Merge Req:

Method: POST

Endpoint: /git/assignReviewer

Request Params:NA

Request Body:

{

"reviewer_id": "2",

"projectid": "1",

"id": "1"

}

7) Merge a request:

Method: POST

Endpoint: /git/mergePullRequest

Request Params:NA

Request Body:

{

"projectid": "1",

"id": "1"

}

Request Header: { Content-Type: “application/json” }

Feature 5: Sonarqube Admin Activities REST API Service

Sonarqube API Service is a Spring Boot-based application that serves REST APIs for the admin to configure resources on Sonarqube

The API Server provides the following endpoints:

1) Create User:

Method: POST

Endpoint: /sonar/createUser

Request Params: NA

Request Body: (type = JSON)

{

"login": "sonaruser",

"name": "sonaruser",

"password": "admin@12345"

}

Request Header: {Content-Type: “application/json”}

2) Create Project:

Method: POST

Endpoint: / git/createProject

Request Params:NA

Request Body:

{

"key": "sonarproj",

"name": "sonarproject",

"visibility": "public"

}

Request Header: { Content-Type: “application/json” }

3) Create Group

Method: POST

Endpoint:/sonar/createGroup

Request Params:NA

Request Body:

{

"name": "sonarGroup"

}

Request Header: { Content-Type: “application/json” }

4) ChangeUserPermission:

Method: POST

Endpoint: /sonar/changeUserPermission

Request Params:NA

Request Body:

Feature 819 of 45

{

"login": "sonaruser",

"permission": "admin"

}

Request Header: { Content-Type: “application/json” }

Feature 6: Nexus Admin Activities REST API Service

Nexus API Service is a Spring Boot-based application that serves REST APIs for the admin to configure resources on Nexus

The API Server provides the following endpoints:

1) Create User:

Method: POST

Endpoint: /nexus/createUser

Request Params: NA

Request Body: (type = JSON)

{

"userId": "nexususer",

"firstName": "nexus",

"lastName": "user",

"emailAddress": "nexususer@gmail.com",

"password": "admin@12345",

"status": "active",

"roles": ["nx-admin"]

}

Request Header: { Content-Type: “application/json” }

2) Create Role:

Method: POST

Endpoint: / nexus/createRole

Request Params:NA

Request Body:

{

"id": "nexusrole",

"name": "nexusrole",

"description": ["nx-all"],

"roles": ["nx-admin"],

}

Request Header: { Content-Type: “application/json” }

3) Create Privilege

Method: POST

Endpoint:/nexus/createPrivilege

Request Params:NA

Request Body:

{

"type": "repository-view",

"name": "nexus privilege",

"description": "my nexus privilege",

"actions": ["READ"],

"format": "rmaven2 ",

"repository": "maven-central",

}

Request Header: { Content-Type: “application/json” }

4) Create Repository:

Method: POST

Endpoint: /sonar/changeUserPermission

Request Params:NA

Request Body:

{

{

"name": "test",

"online": true,

"storage": {

"blobStoreName": "default",

"strictContentTypeValidation": true,

"writePolicy": "allow_once"

},

"cleanup": {

"policyNames": [

"string"

]

},

"component": {

"proprietaryComponents": true

},

"maven": {

"versionPolicy": "MIXED",

"layoutPolicy": "STRICT",

"contentDisposition": "ATTACHMENT"

}

}

Request Header: { Content-Type: “application/json” }

Feature 7: Jenkins Job Trigger REST API Service

Jenkins API Service is a Spring Boot-based application that serves REST APIs for the triggering a job in jebkins

The API Server provides the following endpoints:

1) Job Trigger:

Method: POST

Endpoint: /platform/v1/pipeline/trigger

Request Params: NA

Feature 824 of 45

Request Body: (type = JSON)


```
{


    "repourl":"http://10.63.33.181:32471/root/durga.git",

    "programminglanguage":"java",

    "imagename":"webapp",

    "tag":"latest",

    "buildtool":"Maven",

    "testingtool":"null",

    "codequality":"SonarQube",

    "artifact":"Nexus",

    "vulnerabilityscan":"Grype",

    "job":"AppDeployment",

    "environment":"QA",

    "users":"alice"

}
```


Request Header: { Content-Type: “application/json” }

Feature 8: API Gateway integration with REST server

Install Istio Controller in k8s cluster

1. Deploy Istio:

   Install Istio on your Kubernetes cluster following the official documentation: Istio Installation

Note: istio controller installation is already done in platformEngg's eks cluster , one can check all the pods & services in istio-system namespace

Install Application:

Step-1: Create a deployment namespace in k8s cluster or add a label to it if already exists

kubectl create namespace <namespace-name> --labels=istio-injection=enabled

Or

kubectl label namespace <namespace-name> istio-injection=enabled

Step-2: install application in the namespace

Note: Once application is installed in the labeled namespace , make sure one sidecar container is running along with the application container , if not check istio-injection label for the namespace.

Step-3: Create the application gateway & virtual routing service k8s object in the labeled namespace.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
 name: cluster-gateway
spec:
 selector:
  istio: ingressgateway
 servers:
 - hosts:
  - "cluster.example.com"  # replace with your actual hostname
  port:
   number: 80
   name: http
   protocol: HTTP
```

Note: make sure selector label { istio: ingressgateway } , is applied or use the correct istio controller label, provided while **installing Istio Controller in k8s cluster**

Note: for eks cluster istio installation label { istio: ingressgateway }

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
 name: cluster-virtualservice
spec:
 hosts:
 - "cluster.example.com"  # replace with your actual hostname
 gateways:
 - cluster-gateway
 http:
  - match:
    - uri:
       prefix: "/"
    route:
    - destination:
       host: cluster-service.testings.svc.cluster.local
       port:
         number: 8080
    rewrite:
     uri: "/"
     authority: cluster-service.testings.svc.cluster.local:8080
```

Note: replace host with the your application's k8s hostname , example

Host: <k8s-service-for-application>. <namespace-name>.svc.cluster.local

This application now can be accessed through the api gateway

Step-4: Access Application through gateway

curl -X GET -H “Host: cluster.example.com” http://<gateway-dns>/getStatus

Note: to get the api-gateway loadbalancer ip , list the istio controller services in the istio installation namespace

Note: for eks cluster api-gateway DNS: aa7eabfa05ed24fc3ad6d2c4007e805c-1204085443.us-east-1.elb.amazonaws.com

Step-5: Apply Authentication & Authorization through the api gateway

(1) Deploy request authentication k8s object

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
 name: cluster-request-authentication
spec:
 selector:
  matchLabels:
   app: cluster
 jwtRules:
 - issuer: "http://10.63.35.142:8080/realms/istio"
  jwksUri: "http://10.63.35.142:8080/realms/istio/protocol/openid-connect/certs"
```

Note: use the correct label used inside the application pods inplace of matchLabels { app: cluster }

Note: this issuer url & jwksUri is set for platformEngg's keycloak setup incase of other keycloak setup provide the correct one.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: cluster-auth
spec:
 selector:
  matchLabels:
   app: cluster
 rules:
 - from:
  - source:
     requestPrincipals: ["*"]
  to:
  - operation:
     methods: ["GET"]
     paths: ["/platform/v1/database/*"]
  when:
  - key: request.auth.claims[realm_access][roles]
   values: ["reader"]
 - from:
  - source:
     requestPrincipals: ["*"]
  to:
  - operation:
     methods: ["POST"]
     paths: ["/platform/v1/database/*"]
  when:
  - key: request.auth.claims[realm_access][roles]
   values: ["writer"]
```

(2) Deploy the authorization Policy for virtual service using above script

Note: replace paths with your application's url paths, also make sure matchLabels is the application pod labels & values are the keycloak realm roles

Step-6: Configure Keycloak realm

Note: for eks cluster keycloak realm is istio & UI path is : http://10.63.35.142:8080/

Note: step (a) & (b) can be ignored if using eks keycloak setup

(a) Create Realm:

Click on "Create realm" after selecting the word "master" in the top-left corner. Enter "Istio" in the Realm name field and click "Create."

(b) Create OAuth Client:

Select the newly created realm "Istio."

Click on "Clients" and create a new client named "Istio" with OpenID Connect as the client type.

(C) Create Roles:

Create roles under the realm based on your requirement

(D)Create Users:

Add users to the realm

(E)Set Passwords for Users:

Set strong passwords for users.

(F)Role Assignments:

Assign the required roles to the users.

(G)Get JWT Token:
This is one example url with eks keycloak server

curl --location -X POST 'http://10.63.35.142:8080/realms/istio/protocol/openid-connect/token' \

--header 'Content-Type: application/x-www-form-urlencoded' \

--data-urlencode 'client_id=istio' \

--data-urlencode 'grant_type=password' \

--data-urlencode 'username=test' \

--data-urlencode 'password=test'

Step-7: Accessing Application Endpoint Without token will give rbac errors now so use the keycloak token generated in above steps

curl -X GET -H “Host: cluster.example.com” -H “Authorization: Bearer {{token-generated}} http://<gateway-dns>/getStatus

Feature 9. Application Development Common Services

9.1 Logging Service (EFK)

Logging API Service is a Spring Boot-based application that serves REST APIs for logging some info to elastic search

API Gateway: aa7eabfa05ed24fc3ad6d2c4007e805c-1204085443.us-east-1.elb.amazonaws.com
Host: efk.example.com

1) POST log

Method: POST

Endpoint: /efk/postLog

Request Params: NA

Request Body: (type = JSON)

```
{
"msg": "any message"
}
```

9.2 Caching Service

Caching API Service is a Spring Boot-based application that serves REST APIs for storing cache info to Redis database.

host: caching.example.com

(1) Save cache

Method: POST

Endpoint: /Platform_Caching/

Request Params: NA

Request Body: (type = JSON)

```
{
    “key”: “unique key”,
    “data”: {
          Json data
     }
}
```

(2) Retrieve Cache

Method: GET

Endpoint: /Platform_Caching/{key}

Request Params: NA

Request Body: NA

(3) Delete Cache

Method: DELETE

Endpoint: /Platform_Caching/{key}

Request Params: NA

Request Body: NA

(4) Update Cache

Method: UPDATE

Endpoint: /Platform_Caching/{key}

Request Params: NA

Request Body: (type = JSON)

```
{
   “data”: {
        Json data
    }
}
```

9.3 Messaging Service (PUB/SUB)

Messaging API Service is a Spring Boot-based application that serves REST APIs for writing messages to different topics and retrieving them.

RabbitMQ –API's

Feature 835 of 45

Sequence: create queue -> create topic Exchange -> push message to topic Exchange -> retrieve messages using topic Exchange

Host : rabbitmq.example.com

1) Create Queue

Method: POST

Endpoint: /rabbitMq/createQueue

Request Params: name

Request Body: NA

2) Create Exchange

Method: POST

Endpoint: /rabbitMq/createTopic

Request Params: topicName

Request Body: NA

3) Push Message to Exchange

Method: POST

Endpoint: /rabbitMq/pushMessage

Request Params: topicName

Request Body: String

4) Retrieve messages

Method: POST

Endpoint: /rabbitMq/getMessages

Request Params: queueName

Request Body: NA

5) Get queue details

Method: POST

Endpoint: /rabbitMq/

Request Params: name

Request Body: NA

6) Delete Exchange

Method: POST

Endpoint: /rabbitMq/createQueue

Request Params: name

Request Body: NA

7) Delete Queue

Method: POST

Endpoint: /rabbitMq/createQueue

Request Params: name

Request Body: NA

Kafka – API's

Host: kafka.example.com

1) Create Topic

Method: POST

Endpoint: /createTopic

Feature 837 of 45

Request Params: topicName

Request Body: NA

2) Message to Topic

Method: POST

Endpoint: /sendMessage

Request Params: topicName

Request Body: String

3) Retrieve Messages from topic

Method: GET

Endpoint: /getMessages

Request Params: topicName

Request Body: NA

4) Delete Topic

Method: DELETE

Endpoint: /deleteTopic

Request Params: topicName

Request Body: NA

5) Retrieve all topics

Method: GET

Endpoint: /getTopics

Request Params: NA

Request Body: NA

9.2 Secret Management Service (Vault)

Secret Management API Service is a Spring Boot-based application that serves REST APIs for Creating , Retrieving and Deleting Secrets in Vault

1) **Adding a Secret to Vault**

   Method: POST

   Endpoint: /secret-management/addSecret

   Request Header: { Content-Type: “application/json”, vault-token: “token”,”Host”:”vault.example.com” }

   Request Params: NA

   Request Body: (type = JSON)

   ```
   {
       "username" : "admin",
      "password" : "admin@12345",
      "applicationName" : "mysql"
   }
   ```

2) **Retrieving a Secret from Vault**

   Method: GET

   Endpoint: /secret-management/getSecret

   Request Header: { vault-token: “token” ,”Host”:”vault.example.com”}

   Request Params: {“uuid”:”ID”}

   Request Body: NA

3) **Delete a Secret from Vault**

Method: DELETE

Endpoint: /secret-management/ deleteSecret

Request Header: { vault-token: “token” ,”Host”:”vault.example.com”}

Request Params: {“uuid”:”ID”}

Request Body: NA

4) **Update a Secret from Vault**

Method: Update

Endpoint: /secret-management/ updateSecret

Request Header: { Content-Type: “application/json”, vault-token: “token” ,”Host”:”vault.example.com”}

Request Body: (type = JSON)

```
{
    “id”:”id”,
     "username" : "admin",
    "password" : "admin@12345",
    "applicationName" : "mysql"
}
```

9.3 Data Encryption and Decryption Service (Ubiq)

Data Encryption and Decryption API Service is a Spring Boot-based application that serves REST APIs for Data Encryption and Decryption using Ubiq.

1) **Encryption Of Data**

   Method: POST

   Endpoint: /data-encrypt/encrypt

   Request Header: {Content-Type: “application/json”,”Host”:”cryptography.example.com”}

   Request Params: NA

   Request Body: {Json Data}

2) **Decryption of Data**

   Method: POST

   Endpoint: data-encrypt/decrypt

   Request Header: {Content-Type: “text/plain”,”Host”:”cryptography.example.com”}

   Request Params: NA

   Request Body: “Encrypted String”

9.3 Distributed Tracing (Jaeger)

Jaeger uses distributed tracing to follow the path of a request through different microservices. Rather than guessing, we can see a visual representation of the call flows.

Jaeger includes tools to monitor distributed transactions, optimize performance and latency, and perform root cause analysis (RCA), a method of problem-solving.

The REST APIs that we are used for Tracing are:

1) **Jaeger Dashboard:**

Method: GET

Port: 16686

Request Header: NA

Request Params: NA

Request Body: NA

9.4 License Management (Keygen)

The license key generation microservice is designed to securely generate unique license keys for software applications. It ensures that only authorized users have access to the software, controls license durations.

Sequence: generate token -> create product -> create policy -> create license -> validate license

REST API's

accountId: 93210a0c-0d44-47d1-aada-25769c4dca38

Token Generation: Generates a token for the provided account, which can be used for different operations

Method: POST

Endpoint: http://10.63.35.28:8080/token

Params: accountId

Product Creation: Creating a product for which we require a license

Method: POST

Endpoint: http://10.63.35.28:8080/createProduct

Params: accountId , token

Request Body:

{

```
   "name":"....",
   "url":"........."
}
```

Policy Creation: Creating a policy for the product which sets the duration of a license

Method: POST

Endpoint: http://10.63.35.28:8080/createPolicy

params: accountId,token,productId,duration

License Generation: Generates a unique license based on the policy passed

Method: POST

Endpoint: http://10.63.35.28:8080/createLicense

params: accountId,token,policyId

Get License by Id:

Method: GET

Endpoint: http://10.63.35.28:8080/licenses/{{id}}

Params: accountId,token

Get all licenses:

Method: GET

Endpoint: http://10.63.35.28:8080/licenses

Params: accountId,token

Validate license by key:

Feature 843 of 45

Method: POST

Endpoint: http://10.63.35.28:8080/validate/{{key}}

params: accountId,token

Validate license by id:

Method: POST

Endpoint: http://10.63.35.28:8080/validate/{{id}}

params: accountId,token

9.5 Authentication and Authorization of API’s using keycloack and istio

Steps to Eable Authentication and Authorization for API's

1. Enable istio-injection on the namespace

kubectl create label namespace namespace-name istio-injection=enabled

2. Create RequestAuthentication and AuthorizationPolicy and apply in the same namespace

RequestAuthentication.yaml

apiVersion: security.istio.io/v1beta1

kind: RequestAuthentication

metadata:

  name: {name}

  namespace: {namespace}

spec:

```
  selector:
    matchLabels:
      app: {application-label}
  jwtRules:
    - issuer: "http://10.63.33.181:32569/realms/AuthTest"
      jwksUri: "http://10.63.33.181:32569/realms/AuthTest/protocol/openid-connect/certs"
```

AuthorizationPolicy.yaml

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: {name}
  namespace: {namespace}
spec:
  selector:
    matchLabels:
      app: {application-label}
  rules:
  - from:
    - source:
        requestPrincipals: ["*"]
```

3. Testing

Token Generation

```
curl --location 'http://10.63.33.181:32569/realms/AuthTest/protocol/openid-connect/token' \

--header 'Content-Type: application/x-www-form-urlencoded' \

--data-urlencode 'client_id=istio' \

--data-urlencode 'username=admin' \

--data-urlencode 'password=admin' \

--data-urlencode 'grant_type=password'
```