
System Diagram:-

- Deployment – containers to be deployed
- Service – will create a LoadBalancer for each services to expose our containers to the internet as both requires an endpoint that we are going to make request to.
- **μ-service-2 (backend) will reverse the string when it receives a POST on /reverse endpoint .**

```
curl -X POST http://35.202.23.223:8080/reverse -d '{"messages" : "digneadsax"}'  
--header 'Content-Type: application/json'
```

It is going to respond with

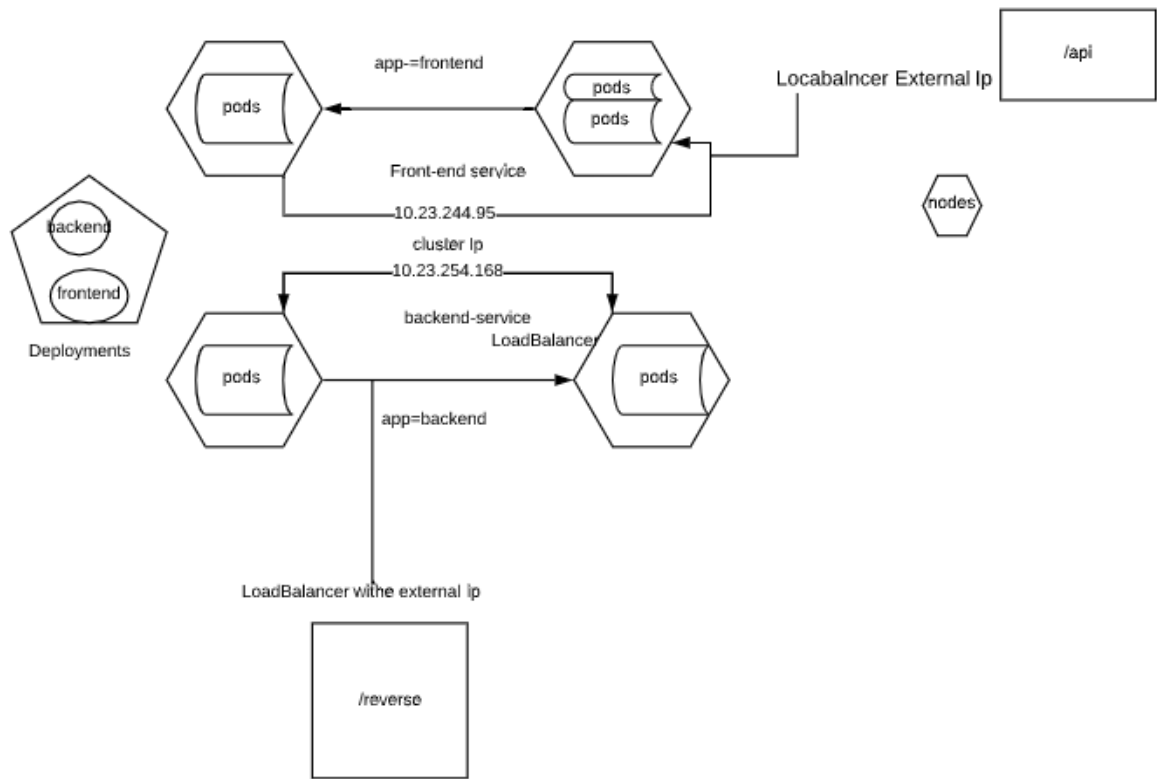
```
{"messages": "xasdaenigid"}
```

- **μ-service-1 (frontend) will reverse the string in which it receives in a Post request by sending a Post request internally to the backend service. It is going to resolve the name of backend service with the help of kubedns as both service objects are in the same namespaces .**
- **Then it will seed a random number ,add it to the payload and will create a response to serve the end user .**

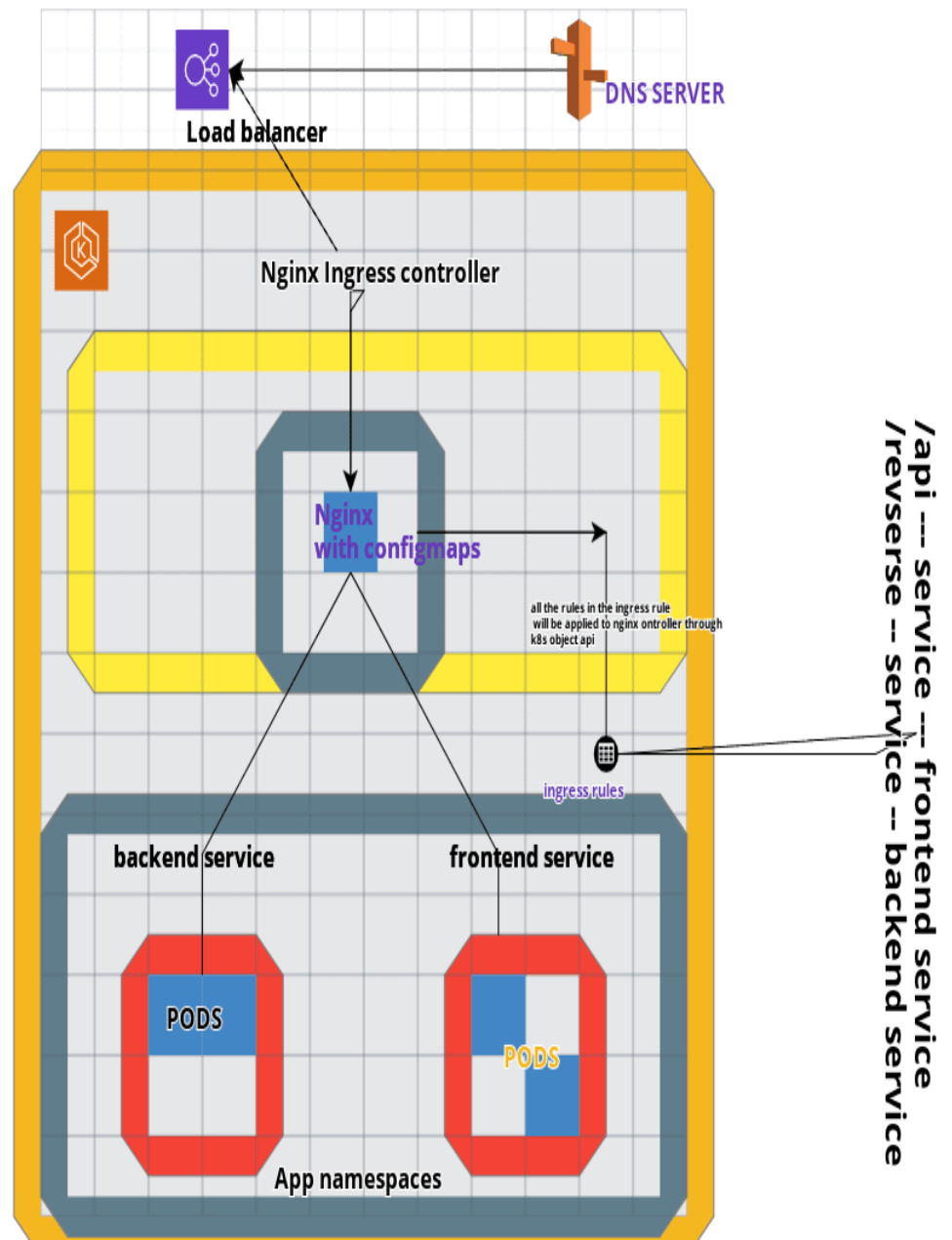
```
curl -X POST http://35.202.24.221:8081/api -d '{"messages" : "digneadsax"}'  
--header 'Content-Type: application/json'
```

Response

```
{"messages": "xasdaenigid", "rand": 0.8474337369372327}
```



We are using two load balancers for each service which is not ideal for working with Production environment on clouds like GKE or AWS with multiple services . Instead of this we can Use an ingress Controller like Nginx with a single load balancer and let nginx do the path based routing on the requests served on the external load balancer . nginx will then do the routing according to the config . We can map a domain name to this load balancer's externally-accessible IP address that sends traffic to the nginx ingress .



CI/CD Integration :-

Continuous integration with Gitlab .

- 1) Create a GitLab repo for each Microservice .
- 2) For CI , Create a .gitlab-ci.yml in the root directory of each microservice . For our microservice we are going to do a static analysis and linting to test the code quality .
- 3) Using the gitlab YML we have to create pipeline which consist set of tests to run against a single git commit every time in each stage of development .
- 4) We are using a share runner to run all the jobs in the pipeline but we can also create our own runner (nodes) and authorize it with gitlab .

```
image: "python:3.7-alpine"

before_script:

  - python --version

  - pip3 install flake8 pylint

stages:

  - Static Analysis

flake8:

  stage: Static Analysis

  script:

    - flake8 --max-line-length=120 app.py

pylint:

  stage: Static Analysis










  allow_failure: true

  script:

    - pylint -d C0301 app*.py
```

5). We are using a docker container to run the tests which usage same base image as our application .

6) Based upon the stages defined in pipeline gitlab will execute the CI Pipeline and shows the

	#83987913 latest		🔗 master → d84b4609 static	
	#83987471		🔗 master → 06ad064e static	
	#83986606		🔗 master → 482acf53 static	

the status according to the commit that happens in different branches of our VCS .

7) We can now create a different branches in our VCS according to the organisation Branch and release strategy of the code release and organisation .

For simplicity this is what it should be :

- Every commit (every branch and PR) triggers a build. Built artifacts are tagged with a version string after passing the test cases (branch-yyyy-mm-dd.buildnumber.gitcommithash)
- Manual approval step required for Pull Request approval, required for merge from feature to other stages .
- From master branch create a release branch where we take all changes with git rebase so we can track every release instead of the commits which we are doing in other stages to test the code and application either manually or through automation .
- For every successful build and for the deployments test we can push to artifacts to a central repository from where we can do the deployments .
- Branching strategy also depends on the size of the organization , number of people that are working in the team . For an alternative we can also use this .

<https://nvie.com/posts/a-successful-git-branching-model/>

CD:-

Continuous delivery

- 1) In the CI pipeline we are testing our application in an integrated state continuously . In this stage we are making sure after every successful test the code artifacts will get stored in a central/remote repository .
- 2) For that we are expanding our GitLab yml file to build a container with every commit and push it to the remote docker repo with the formatted tags .

```
variables:
  DOCKER_DRIVER: "overlay2"
  REPOSITORY_URL: ""
  tag: "$BRANCH_$DATE_$HASH"
services:
  - docker:dind

build-app:|
  stage: build

  script:
    - mkdir -p $HOME/.docker
    - apk add --no-cache curl jq python py-pip
    - pip install awscli
    - ${aws ecr get-login --no-include-email --region ap-southeast-1}
    - docker info
    - docker build -t ${REPOSITORY_URL}-app:$tag .
    - docker push ${REPOSITORY_URL}-app:$tag
```

This Job will executed in CI/CD pipeline of Gitlab and gitlab will push it to the docker repo with the formatted tag of branch name , commit hash and date , to track the deployment artifacts

Continuous Deployments

- 1) Till now we have resolved the delivery and integration aspects of our application . We are now going to deploy it on kubernetes cluster with our pipeline . Either we can create a new Cluster on GKE or authenticate the existing one with gitlab .

```
stages:
- test
- deploy

test:

deploy to staging:
  stage: deploy
  script: make deploy
  environment:
    name: staging
    url: https://staging.example.com/

deploy to production:
  stage: deploy
  script: make deploy
  environment:
    name: production
    url: https://example.com/
```

-
- 2) To Counter the downtime we will use Rolling deployments of our container app in kubernetes with Readiness Probe config option , so the old pods will only get terminated when new pods are able to return a valid request according to the config option .
 - 3) Another thing we can do is to define the rolling upgrade strategy in deployment config .which specifies the maximum number of Pods that can be unavailable during the update process .