DevOps Project

Completed and Submitted By:

Priyanka Agrawal

ABSTRACT

The project described in this documentation is the final task allotted to me during my internship duration and is all about how DevOps is playing a major role in today's IT industry. DevOps, in itself, is a very vast technology which provides the best practices for integrating the development, testing, operational and deployment parts in such a way so as to shorten the entire SDLC lifecycle and ensure continuous delivery of software with high quality to clients. My project thus includes a part of DevOps. At first, I was provided with a sufficient time to get myself trained with the major requirements of DevOps which included Linux System Administration, Containerization through Docker, Container Orchestration through Kubernetes and Package Management on Kubernetes through Helm. I was also provided with some individual practical tasks over these topics. Then I was given the final project which included the implementation of all these technologies together with a vision to assess and enhance my knowledge and problem-solving techniques in real time production environment.

I was provided with the task to deploy an application using DevOps keeping the production environment into consideration.

The web application which was provided to me for deployment was developed on the framework of Ruby on Rails with which I also had to integrate the database provision. The application provided to me was Redmine which is an open source and free, web-based flexible project management and issue tracking tool. The application has an already integrated SQLite database which serves the purpose of data storage and management. Apart from this, the application supports the integration of any other database as per the requirement. Therefore, I was provided with the requirement of using any database other than the inbuilt one.

I, thus, had to perform the deployment and container orchestration of the given application and its database over Kubernetes, by using the troubleshooting capabilities of both, Docker and Kubernetes. I was also given a custom configuration file for application routing which was to be mounted to a read-only filesystem on a running container. Apart from the containerization and orchestration part, I also had to develop a customized chart in Helm whose sole purpose would be the monitoring of the deployed application.

1. Technology Used

1.1 Overview

“Devops is not a goal, but a never-ending process of continual improvement.” – Jez Humble

DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes. What does DevOps mean for teams? DevOps enables formerly siloed roles—development, IT operations, quality engineering and security—to coordinate and collaborate to produce better, more reliable products. By adopting a DevOps culture along with DevOps practices and tools, teams gain the ability to better respond to customer needs, increase confidence in the applications they build and achieve business goals faster. The word “DevOps” is a mashup of “development’ and “operations”.

DevOps describes approaches to speeding up the processes by which an idea (like a new software feature, a request for enhancement, or a bug fix) goes from development to deployment in a production environment where it can provide value to the user. These approaches require that development teams and operations teams communicate frequently and approach their work with empathy for their teammates. Scalability and flexible provisioning are also necessary. With DevOps, those that need power the most, get it—through self-service and automation.

The DevOps approach goes hand-in-hand with Linux containers, which give your team the underlying technology needed for a cloud-native development style. Containers support a unified environment for development, delivery, integration, and automation. And Kubernetes is the modern way to automate Linux container operations. Kubernetes helps you easily and efficiently manage clusters running Linux containers across public, private, or hybrid clouds.

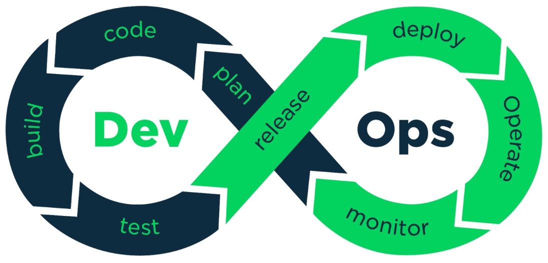


Fig 1: DevOps Overview

Let’s have a look at the containerization and its orchestration through Docker, Kubernetes and Helm one by one:

1.2 Docker

1.2.1 Overview

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

1.2.2 The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

1.2.3 Docker Engine

Docker Engine is a client-server application with these major components:

- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).

The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI. The daemon creates and manages Docker *objects*, such as images, containers, networks, and volumes.

1.2.4 Docker architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

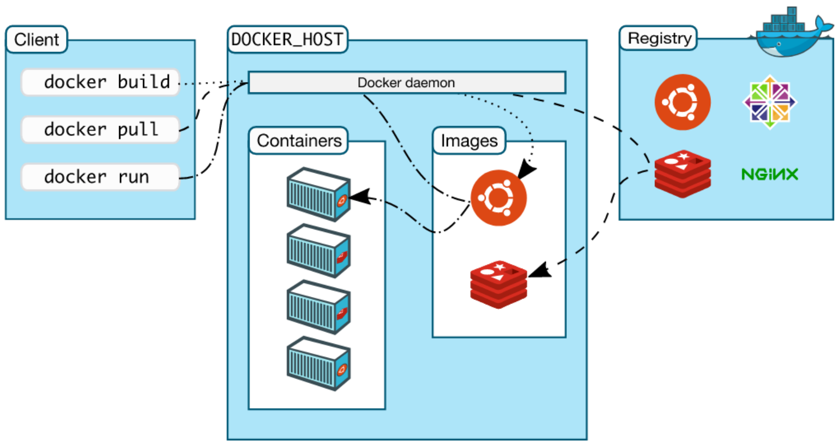


Fig 2: Docker Architecture

1.2.5 What can I use Docker for?

- Fast, consistent delivery of your applications
- Responsive deployment and scaling
- Running more workloads on the same hardware

1.3 Kubernetes

1.3.1 Overview

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. Google open-sourced the Kubernetes project in 2014.

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- Automatic bin packing
- Self-healing
- Secret and configuration management

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, and monitoring. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

1.3.2 Kubernetes Architecture

This architecture of Kubernetes provides a flexible, loosely-coupled mechanism for service discovery. Like most distributed computing platforms, a Kubernetes cluster consists of at least one master and multiple compute nodes. The master is responsible for exposing the application program interface (API), scheduling the deployments and managing the overall cluster. Each node runs a container runtime, such as Docker or rkt, along with an agent that communicates with the master. The node also runs additional components for logging, monitoring, service discovery and optional add-ons. Nodes are the workhorses of a Kubernetes cluster. They expose compute, networking and storage resources to applications. Nodes can be virtual machines (VMs) running in a cloud or bare metal servers running within the data center.

As seen in the following diagram, Kubernetes follows client-server architecture. Wherein, we have master installed on one machine and the node on separate Linux machines.

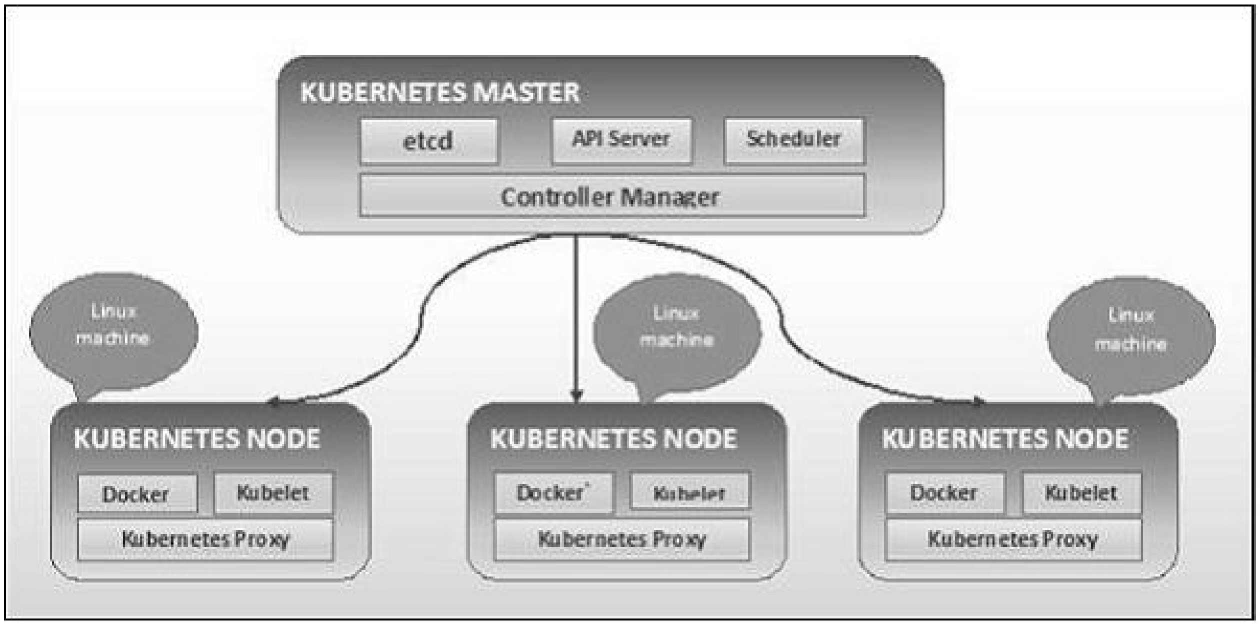


Fig 3: Kubernetes Architecture

Kubernetes - Master Machine Components

Following are the components of Kubernetes Master Machine.

etcd

It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes. It is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key value Store which is accessible to all.

API Server

Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

Controller Manager

This component is responsible for most of the collectors that regulates the state of cluster and performs a task. In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server. It works toward getting the shared state of cluster and then make changes to bring the current status of the server to the desired state. The key controllers are replication controller, endpoint controller, namespace controller, and service account controller. The controller manager runs different kind of controllers to handle nodes, endpoints, etc.

Scheduler

This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

Kubernetes - Node Components

Following are the key components of Node server which are necessary to communicate with Kubernetes master.

Docker

The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

Kubelet Service

This is a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and wright values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

Kubernetes Proxy Service

This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers' health checkup, etc.

Kubernetes - Master and Node Structure

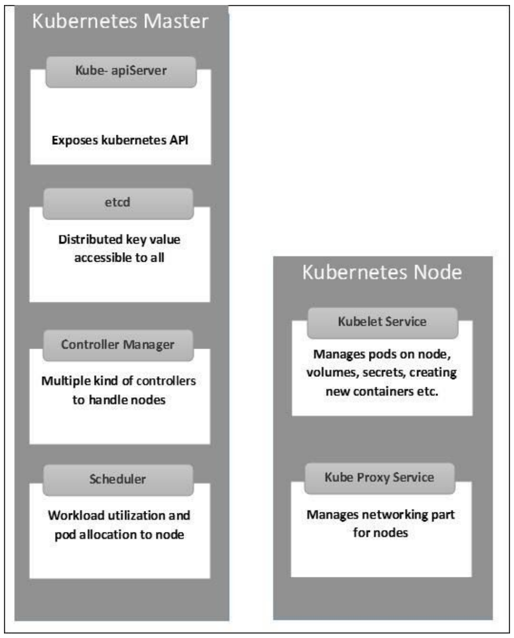

Kubernetes Master
Kube- apiServer
Exposes kubernetes API
etcd
Distributed key value accessible to all
Controller Manager
Multiple kind of controllers to handle nodes
Scheduler
Workload utilization and pod allocation to node
Kubernetes Node
Kubelet Service
Manages pods on node, volumes, secrets, creating new containers etc.
Kube Proxy Service
Manages networking part for nodes

Fig 4: Kubernetes master and node structure

1.4 Helm

Deploying applications to Kubernetes – the powerful and popular container-orchestration system – can be complex. Setting up a single application can involve creating multiple interdependent Kubernetes resources – such as pods, services, deployments, and replicasets – each requiring you to write a detailed YAML manifest file.

Helm is a package manager for Kubernetes that allows developers and operators to more easily package, configure, and deploy applications and services onto Kubernetes clusters.

Helm is now an official Kubernetes project and is part of the Cloud Native Computing Foundation, a non-profit that supports open source projects in and around the Kubernetes ecosystem.

Helm can:

- Install software.
- Automatically install software dependencies.
- Upgrade software.
- Configure software deployments.
- Fetch software packages from repositories.

Helm provides this functionality through the following components:

- A command line tool, helm, which provides the user interface to all Helm functionality.
- A companion server component, tiller, that runs on your Kubernetes cluster, listens for commands from helm, and handles the configuration and deployment of software releases on the cluster.
- The Helm packaging format, called *charts*.
- An official curated charts repository with prepackaged charts for popular open-source software projects.

Charts

Helm packages are called *charts*, and they consist of a few YAML configuration files and some templates that are rendered into Kubernetes manifest files. Here is the basic directory structure of a chart:

Example chart directory

```
package-name/

charts/

templates/

Chart.yaml

LICENSE

README.md

requirements.yaml

values.yaml
```

These directories and files have the following functions:

- charts/: Manually managed chart dependencies can be placed in this directory, though it is typically better to use requirements.yaml to dynamically link dependencies.
- templates/: This directory contains template files that are combined with configuration values (from values.yaml and the command line) and rendered into Kubernetes manifests. The templates use the Go programming language's template format.
- Chart.yaml: A YAML file with metadata about the chart, such as chart name and version, maintainer information, a relevant website, and search keywords.
- LICENSE: A plaintext license for the chart.
- README.md: A readme file with information for users of the chart.
- requirements.yaml: A YAML file that lists the chart's dependencies.
- values.yaml: A YAML file of default configuration values for the chart.

The helm command can install a chart from a local directory, or from a .tar.gz packaged version of this directory structure. These packaged charts can also be automatically downloaded and installed from chart repositories or repos.

2. Project Description

2.1 Problem Statement

The problem statement provided to me included a web application developed on Ruby On Rails framework which needed to be deployed along with the database over kubernetes by doing all the configurations, keeping production environment in consideration. The approach to solve the given task was solely dependent on me. All the source code and configuration files were needed to be pushed to the git repository.

This has to be done either by using minikube or a single node cluster deployed via kubeadm. I also had to create persistent volumes using hostPath.

There had to be two different namespaces, each for the application and the database respectively. Use of configmaps and secrets should be done to configure environment variables and any sensitive information and should be mounted as volumes too. Liveness and Readiness probes should be used to increase efficiency. A network policy should also be created which would only allow inbound traffic from the application's namespace to that of the database one.

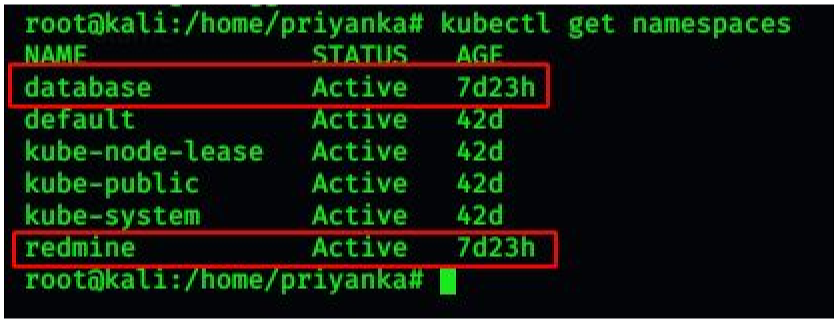
A custom configuration file was provided to me which was to be mounted as a configmap mand replace the existing configuration file onto the running container of the application.

Lastly, I had to deploy a monitoring stack for the application to be monitored by using Prometheus and Grafana. This was to be done via Helm v3. This task included the deployment of monitoring stack through the creation of a custom chart by the combination of existing helm charts of Prometheus and Grafana. The configuration of the charts had to be done in such a way that the deployed monitoring stack would only monitor the application and not the entire cluster, node or kube system.

2.2 Approach towards the Solution

The prerequisite to solve the given problem statement includes the installation of Docker and minikube along with kubectl.

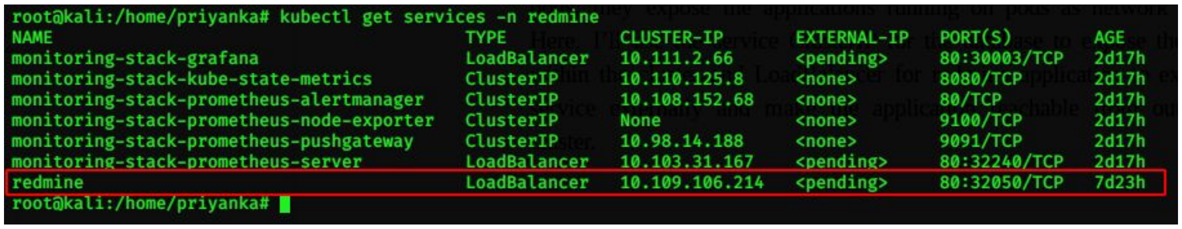
❑ First of all, according to the requirement, two namespaces namely redmine and database are created to separate the frontend and backend environment so that two different teams could work over two virtually created clusters without each other's intervention. Two yaml files for each namespace are created and deployed. kubectl command line can be used to achieve this.



```
root@kali:/home/priyanka# kubectl get namespaces
NAME              STATUS   AGE
database          Active   7d23h
default           Active   42d
kube-node-lease   Active   42d
kube-public       Active   42d
kube-system       Active   42d
redmine           Active   7d23h
root@kali:/home/priyanka# 
```

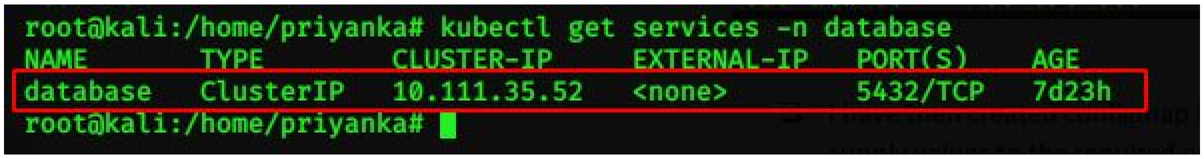
Fig 5: Namespaces

❑ Next, respective services are created, each for the redmine application and the database. Services are abstractions that define a policy to access a set of pods, they expose the applications running on pods as network services. Here, I'll use the service ClusterIP for the database to expose the service within the cluster and LoadBalancer for redmine application to expose the service externally and make the application reachable from outside the cluster.



```
root@kali:/home/priyanka# kubectl get services -n redmine
NAME                                        TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
monitoring-stack-grafana                    LoadBalancer   10.111.2.66      <pending>     80:30003/TCP     2d17h
monitoring-stack-kube-state-metrics         ClusterIP      10.110.125.8     <none>        8080/TCP         2d17h
monitoring-stack-prometheus-alertmanager    ClusterIP      10.108.152.68    <none>        80/TCP           2d17h
monitoring-stack-prometheus-node-exporter   ClusterIP      None             <none>        9100/TCP         2d17h
monitoring-stack-prometheus-pushgateway     ClusterIP      10.98.14.188     <none>        9091/TCP         2d17h
monitoring-stack-prometheus-server          LoadBalancer   10.103.31.167    <pending>     80:32240/TCP     2d17h
redmine                                     LoadBalancer   10.109.106.214   <pending>     80:32050/TCP     7d23h
root@kali:/home/priyanka# 
```

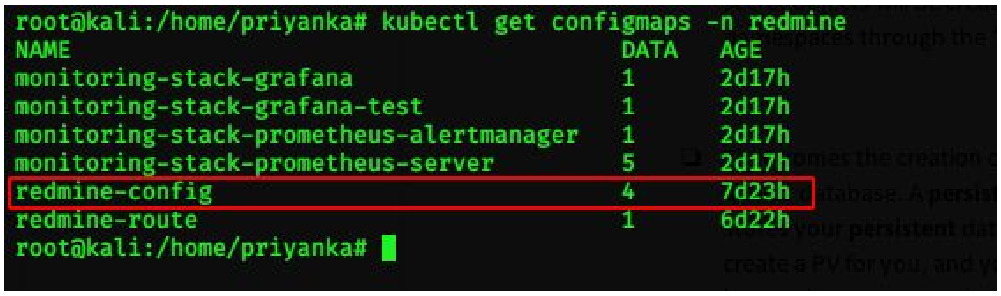
Fig 6: Redmine-service



```
root@kali:/home/priyanka# kubectl get services -n database
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
database   ClusterIP   10.111.35.52   <none>        5432/TCP   7d23h
root@kali:/home/priyanka# 
```

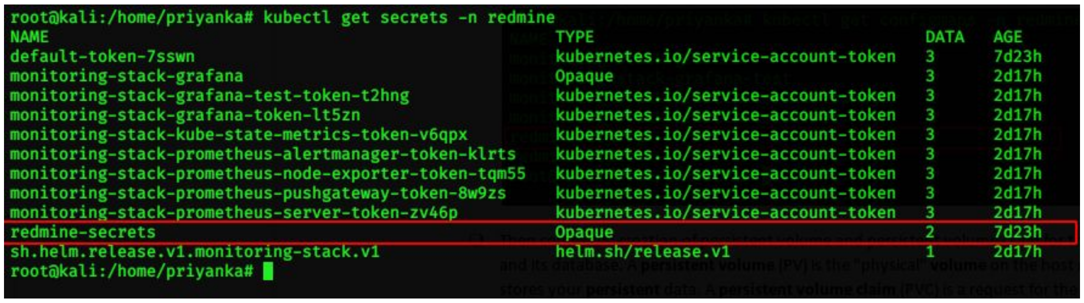
Fig 7: Database-service

- I have then created configmap for the application and the database respectively in order to supply values to the required environment variables for both of them. Secrets will be used to supply sensitive information such as the username and password. Separate configmap and secret objects will be created through the yaml file and deployed in their respective namespaces through the 'kubectl create' command.



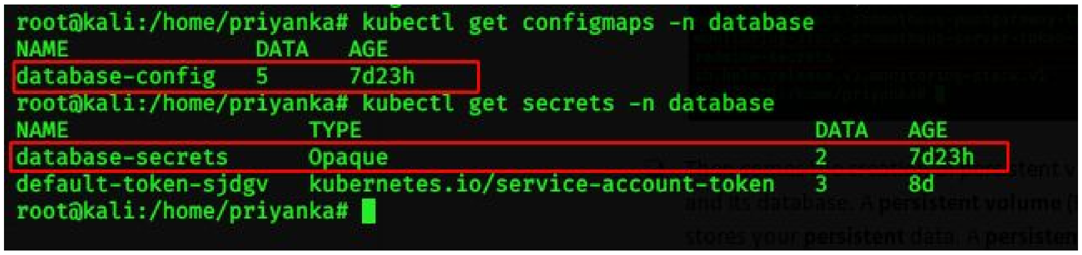
```
root@kali:/home/priyanka# kubectl get configmaps -n redmine
NAME                                        DATA   AGE
monitoring-stack-grafana                    1      2d17h
monitoring-stack-grafana-test               1      2d17h
monitoring-stack-prometheus-alertmanager    1      2d17h
monitoring-stack-prometheus-server          5      2d17h
redmine-config                              4      7d23h
redmine-route                               1      6d22h
root@kali:/home/priyanka#
```

Fig 8: Redmine-configmap



```
root@kali:/home/priyanka# kubectl get secrets -n redmine
NAME                                                 TYPE                                  DATA   AGE
default-token-7sswn                                  kubernetes.io/service-account-token   3      7d23h
monitoring-stack-grafana                             Opaque                                3      2d17h
monitoring-stack-grafana-test-token-t2hng            kubernetes.io/service-account-token   3      2d17h
monitoring-stack-grafana-token-lt5zn                 kubernetes.io/service-account-token   3      2d17h
monitoring-stack-kube-state-metrics-token-v6qpx      kubernetes.io/service-account-token   3      2d17h
monitoring-stack-prometheus-alertmanager-token-klrts kubernetes.io/service-account-token   3      2d17h
monitoring-stack-prometheus-node-exporter-token-tqm55 kubernetes.io/service-account-token  3      2d17h
monitoring-stack-prometheus-pushgateway-token-8w9zs  kubernetes.io/service-account-token   3      2d17h
monitoring-stack-prometheus-server-token-zv46p       kubernetes.io/service-account-token   3      2d17h
redmine-secrets                                      Opaque                                2      7d23h
sh.helm.release.v1.monitoring-stack.v1               helm.sh/release.v1                    1      2d17h
root@kali:/home/priyanka#
```

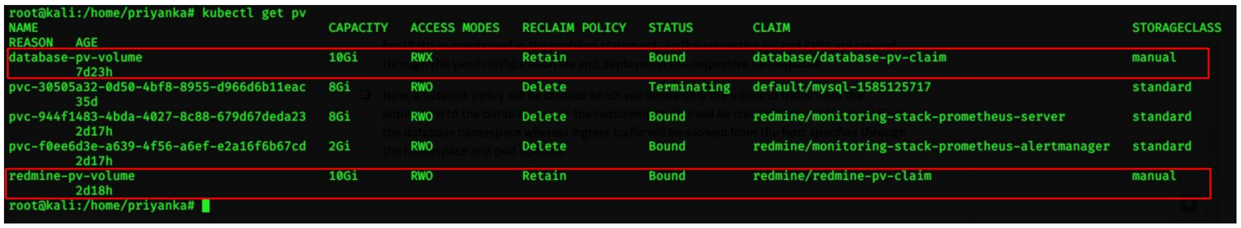
Fig 9: Redmine-secret



```
root@kali:/home/priyanka# kubectl get configmaps -n database
NAME              DATA   AGE
database-config   5      7d23h
root@kali:/home/priyanka# kubectl get secrets -n database
NAME                  TYPE                                  DATA   AGE
database-secrets      Opaque                                2      7d23h
default-token-sjdgv   kubernetes.io/service-account-token   3      8d
root@kali:/home/priyanka#
```

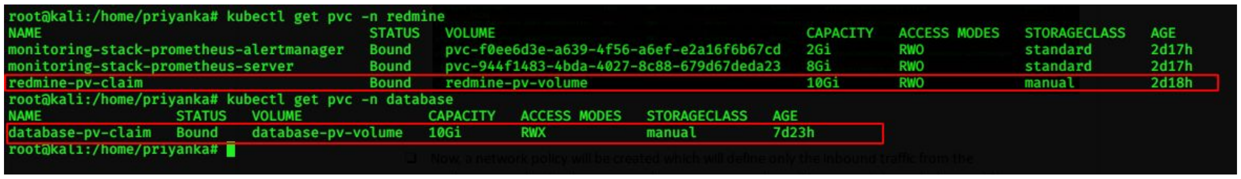
Fig 10: Database configmap and secret

- ❑ Then comes the creation of persistent volume and persistent volume claim for the application and its database. A persistent volume (PV) is the "physical" volume on the host machine that stores your persistent data. A persistent volume claim (PVC) is a request for the platform to create a PV for you, and you attach PVs to your pods via a PVC. I'll create them using the hostPath as mentioned in the problem statement. So the respective PV and PVCs are created through the yaml configuration file and deployed in the respective namespaces.



```
root@kali:/home/priyanka# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS        CLAIM                                           STORAGECLASS
REASON   AGE
database-pv-volume                         10Gi       RWX            Retain           Bound         database/database-pv-claim                      manual
         7d23h
pvc-30505a32-0d50-4bf8-8955-d966d6b11eac   8Gi        RWO            Delete           Terminating   default/mysql-1585125717                        standard
         35d
pvc-944f1483-4bda-4027-8c88-679d67deda23   8Gi        RWO            Delete           Bound         redmine/monitoring-stack-prometheus-server       standard
         2d17h
pvc-f0ee6d3e-a639-4f56-a6ef-e2a16f6b67cd   2Gi        RWO            Delete           Bound         redmine/monitoring-stack-prometheus-alertmanager standard
         2d17h
redmine-pv-volume                          10Gi       RWO            Retain           Bound         redmine/redmine-pv-claim                        manual
         2d18h
root@kali:/home/priyanka#
```

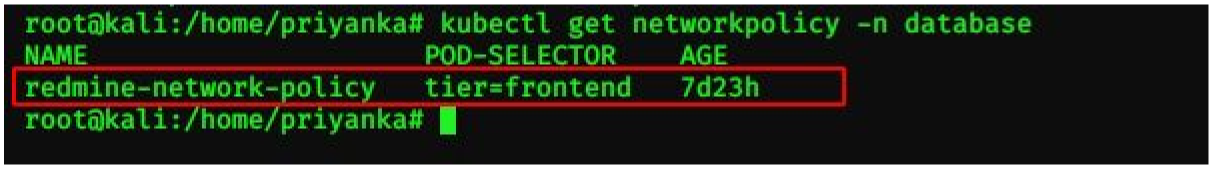
Fig 11: Persistent Volumes



```
root@kali:/home/priyanka# kubectl get pvc -n redmine
NAME                                       STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
monitoring-stack-prometheus-alertmanager   Bound    pvc-f0ee6d3e-a639-4f56-a6ef-e2a16f6b67cd   2Gi        RWO            standard       2d17h
monitoring-stack-prometheus-server         Bound    pvc-944f1483-4bda-4027-8c88-679d67deda23   8Gi        RWO            standard       2d17h
redmine-pv-claim                           Bound    redmine-pv-volume                          10Gi       RWO            manual         2d18h
root@kali:/home/priyanka# kubectl get pvc -n database
NAME                  STATUS   VOLUME               CAPACITY   ACCESS MODES   STORAGECLASS   AGE
database-pv-claim     Bound    database-pv-volume   10Gi       RWX            manual         7d23h
root@kali:/home/priyanka#
```

Fig 12: Persistent Volume Claims

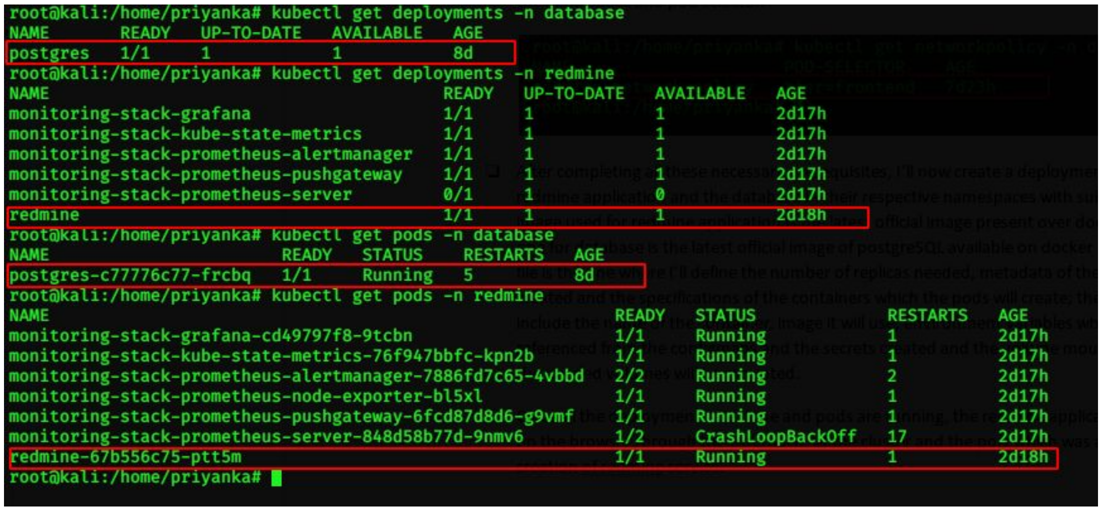
- ❑ Now, a network policy will be created which will define only the inbound traffic from the application to the database as per the requirement. This will be created through the yaml file in the database namespace wherein ingress traffic will be allowed from the host specified through the namespace and pod selector.



```
root@kali:/home/priyanka# kubectl get networkpolicy -n database
NAME                     POD-SELECTOR    AGE
redmine-network-policy   tier=frontend   7d23h
root@kali:/home/priyanka#
```

Fig 13: Network Policy

❏ After completing all these necessary prerequisites, I'll now create a deployment file each for the redmine application and the database in their respective namespaces with suitable labels. The image used for redmine application is the latest official image present over docker hub and the one for database is the latest official image of postgreSQL available on docker hub. Deployment file is the one where I'll define the number of replicas needed, metadata of the pods to be created and the specifications of the containers which the pods will create; these specifications include the name of the container, image it will use, environment variables which will be referenced from the configmaps and the secrets created and the volume mounts section where the needed volumes will be mounted.



```
root@kali:/home/priyanka# kubectl get deployments -n database
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
postgres   1/1     1            1           8d
root@kali:/home/priyanka# kubectl get deployments -n redmine
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
monitoring-stack-grafana                1/1     1            1           2d17h
monitoring-stack-kube-state-metrics     1/1     1            1           2d17h
monitoring-stack-prometheus-alertmanager 1/1    1            1           2d17h
monitoring-stack-prometheus-pushgateway 1/1     1            1           2d17h
monitoring-stack-prometheus-server      0/1     1            0           2d17h
redmine                                 1/1     1            1           2d18h
root@kali:/home/priyanka# kubectl get pods -n database
NAME                       READY   STATUS    RESTARTS   AGE
postgres-c77776c77-frcbq   1/1     Running   5          8d
root@kali:/home/priyanka# kubectl get pods -n redmine
NAME                                                     READY   STATUS             RESTARTS   AGE
monitoring-stack-grafana-cd49797f8-9tcbn                 1/1     Running            1          2d17h
monitoring-stack-kube-state-metrics-76f947bbfc-kpn2b     1/1     Running            1          2d17h
monitoring-stack-prometheus-alertmanager-7886fd7c65-4vbbd 2/2    Running            2          2d17h
monitoring-stack-prometheus-node-exporter-bl5xl          1/1     Running            1          2d17h
monitoring-stack-prometheus-pushgateway-6fcd87d8d6-g9vmf 1/1     Running            1          2d17h
monitoring-stack-prometheus-server-848d58b77d-9nmv6      1/2     CrashLoopBackOff   17         2d17h
redmine-67b556c75-ptt5m                                  1/1     Running            1          2d18h
root@kali:/home/priyanka#
```

Fig 14: Redmine and Database deployments and pods

The pods can further be described using the command:

kubectl describe <pod name> --namespace <namespace>

❏ Once all the deployments are done and pods are running, the redmine application can be run on the browser through the external IP of the cluster and the port which was assigned after the creation of the redmine service.

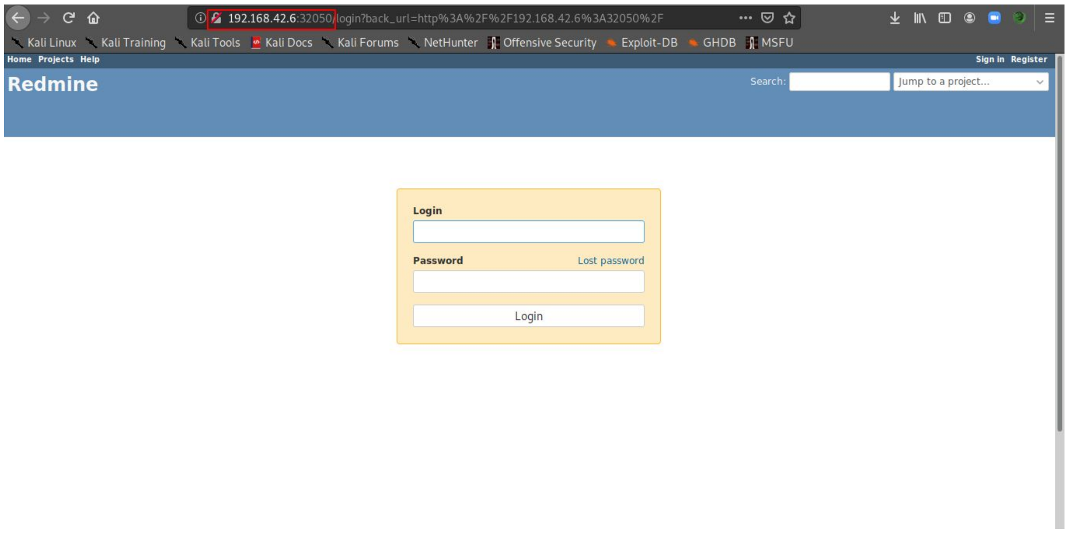
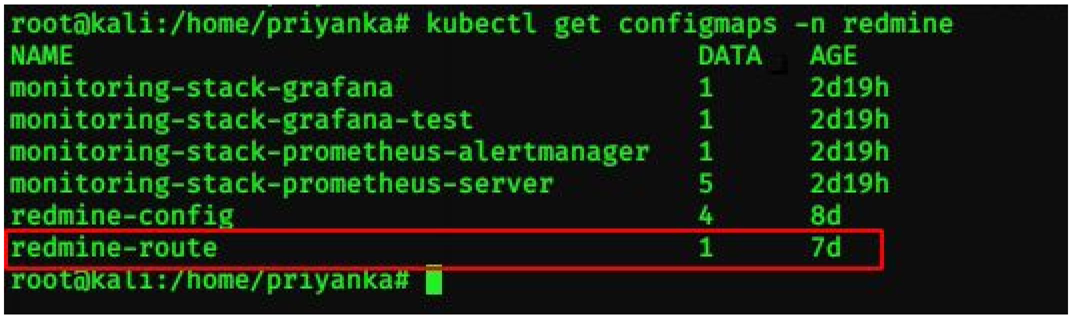


Fig 15: Accessing the application

- Now since the required containers are up and running, my next task was to mount a given config map file at the location /usr/src/redmine/config/routes.rb in the redmine container.

  This could not be done by simply mounting the file to the required location since the specified location is owned by user redmine and is a read-only file system for any other user. By default, the container runs as a root user and the root user is permitted to write or modify the redmine user's directory.

  So, to achieve this task, I made use of security context property in the pod's specification section. A security context defines privilege and access control settings for a Pod or Container. So, first of all I deployed the given configmap 'redmine-route' in the redmine namespace and then edited the redmine deployment file to include the configuration of securityContext field in the Pod specification.

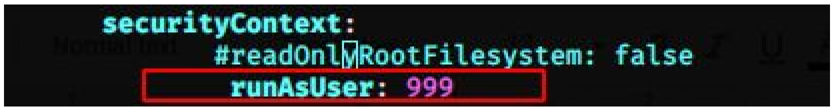


```
root@kali:/home/priyanka# kubectl get configmaps -n redmine
NAME                                        DATA   AGE
monitoring-stack-grafana                    1      2d19h
monitoring-stack-grafana-test               1      2d19h
monitoring-stack-prometheus-alertmanager    1      2d19h
monitoring-stack-prometheus-server          5      2d19h
redmine-config                              4      8d
redmine-route                               1      7d
root@kali:/home/priyanka#
```

Fig 16: Deployment of the given configmap

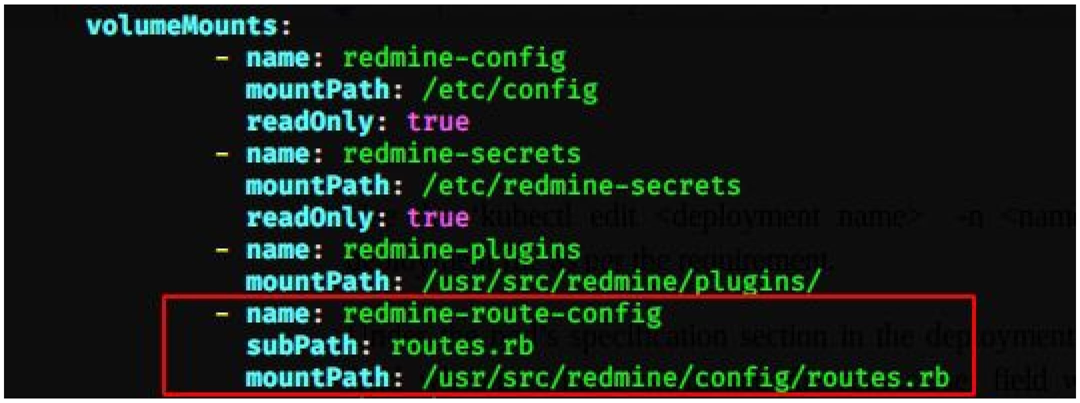
Use the ‘kubectl edit <deployment name> -n <namespace>’ command to edit the deployment file as per the requirement.

Under the pod’s specification section in the deployment file, I added a securityContext field wherein I have mentioned the runAsUser field with the value 999 which itself demonstrates that for any container in the specified Pod, all the processes will run with userID 999 which is the user ID of redmine user. Thus, the container processes now run as a redmine user and the file mount at the specified location is therefore successful. Now, I just need to mount the configmap into the app’s container to the specified location i.e. /usr/src/redmine/config/routes.rb



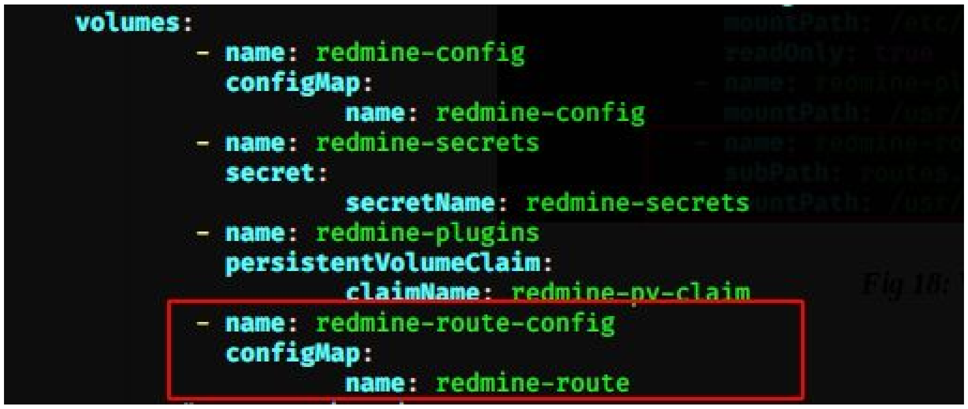
```
securityContext:
       #readOnlyRootFilesystem: false
        runAsUser: 999
```

Fig 17: Security Context under Pod’s Specification



```
volumeMounts:
        - name: redmine-config
          mountPath: /etc/config
          readOnly: true
        - name: redmine-secrets
          mountPath: /etc/redmine-secrets
          readOnly: true
        - name: redmine-plugins
          mountPath: /usr/src/redmine/plugins/
        - name: redmine-route-config
          subPath: routes.rb
          mountPath: /usr/src/redmine/config/routes.rb
```

Fig 18: Volume Mount of the given configmap to specified location



```
volumes:
        - name: redmine-config
          configMap:
                  name: redmine-config
        - name: redmine-secrets
          secret:
                  secretName: redmine-secrets
        - name: redmine-plugins
          persistentVolumeClaim:
                  claimName: redmine-pv-claim
        - name: redmine-route-config
          configMap:
                  name: redmine-route
```

Fig 19: Specification of given ConfigMap in Volumes section

This mounts the given config map to the specified location.

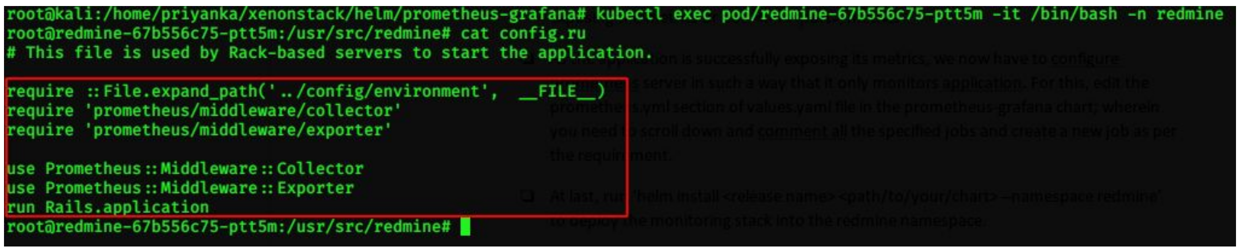
❑ Now comes the task of deploying the monitoring stack for the application. For this, Helmv3 is needed as per the requirement. As it is known that Helm already has individual charts of Prometheus and Grafana to get them deployed one by one respectively. The task involved the creation of a single chart which would install prometheus and grafana in one go and deploy the entire monitoring stack.

  ❑ I did it by installing the contents of both the charts individually into separate local directories.
  ❑ Use the command 'helm create <chart name> ' to create your own chart
  ❑ It will provide a directory with the chart's name
  ❑ I copied the contents of the prometheus chart into my new chart named 'prometheus-grafana'
  ❑ Since I have to work over Helm3 so I moved the contents of requirements.yaml and requirements.lock files into Chart.yaml and Chart.lock files respectively as per the changes made from Helm2 to Helm3
  ❑ Then, I included grafana as a dependency in the Chart.yaml and Chart.lock file and the contents of the grafana chart into the charts/ directory
  ❑ This will combine the charts of prometheus and grafana and will deploy the entire monitoring stack in one go.

❑ Lastly, there was another requirement of deploying the monitoring stack in such a way that it only monitors the application and not the entire cluster, node or kube system as it is meant for by its default configuration.

  ❑ For this, first of all configure the redmine application in such a way that it exposes its metrics to prometheus.

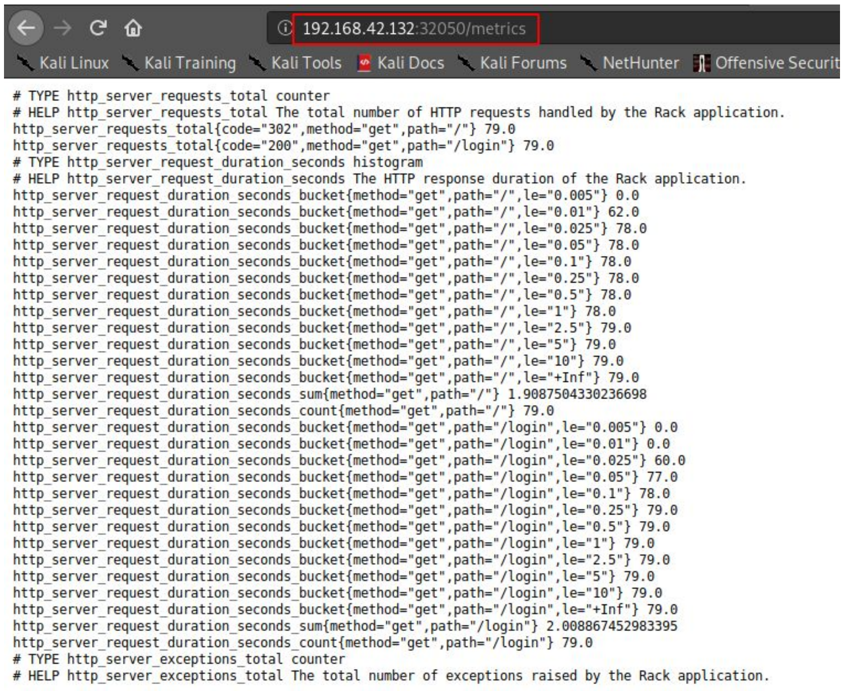
  ❑ Install the gem named prometheus-client and then change the config.ru file inside the redmine container as shown below:



```
root@kali:/home/priyanka/xenonstack/helm/prometheus-grafana# kubectl exec pod/redmine-67b556c75-ptt5m -it /bin/bash -n redmine
root@redmine-67b556c75-ptt5m:/usr/src/redmine# cat config.ru
# This file is used by Rack-based servers to start the application.

require ::File.expand_path('../config/environment',  __FILE__)
require 'prometheus/middleware/collector'
require 'prometheus/middleware/exporter'

use Prometheus::Middleware::Collector
use Prometheus::Middleware::Exporter
run Rails.application
root@redmine-67b556c75-ptt5m:/usr/src/redmine# 
```

Fig 20: Replacement of config.ru file in redmine container to expose metrics

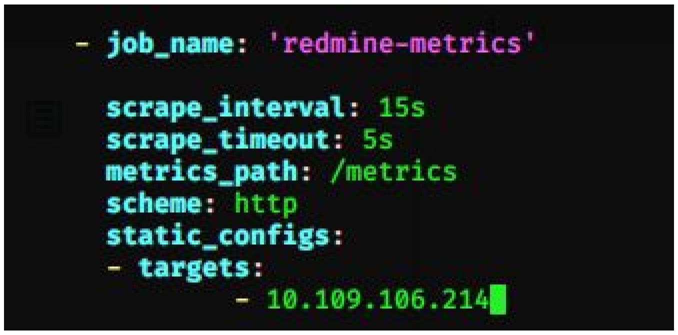
- ❑ Now try accessing the redmine application through the browser and verify whether it is exposing the metrics or not: <IP>:<port>/metrics



```
# TYPE http_server_requests_total counter
# HELP http_server_requests_total The total number of HTTP requests handled by the Rack application.
http_server_requests_total{code="302",method="get",path="/"} 79.0
http_server_requests_total{code="200",method="get",path="/login"} 79.0
# TYPE http_server_request_duration_seconds histogram
# HELP http_server_request_duration_seconds The HTTP response duration of the Rack application.
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.005"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.01"} 62.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.025"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.05"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.1"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.25"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.5"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="1"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="2.5"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="5"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="10"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="+Inf"} 79.0
http_server_request_duration_seconds_sum{method="get",path="/"} 1.9087504330236698
http_server_request_duration_seconds_count{method="get",path="/"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.005"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.01"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.025"} 60.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.05"} 77.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.1"} 78.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.25"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="0.5"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="1"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="2.5"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="5"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="10"} 79.0
http_server_request_duration_seconds_bucket{method="get",path="/login",le="+Inf"} 79.0
http_server_request_duration_seconds_sum{method="get",path="/login"} 2.008867452983395
http_server_request_duration_seconds_count{method="get",path="/login"} 79.0
# TYPE http_server_exceptions_total counter
# HELP http_server_exceptions_total The total number of exceptions raised by the Rack application.
```

Fig 21: Exposing application metrics

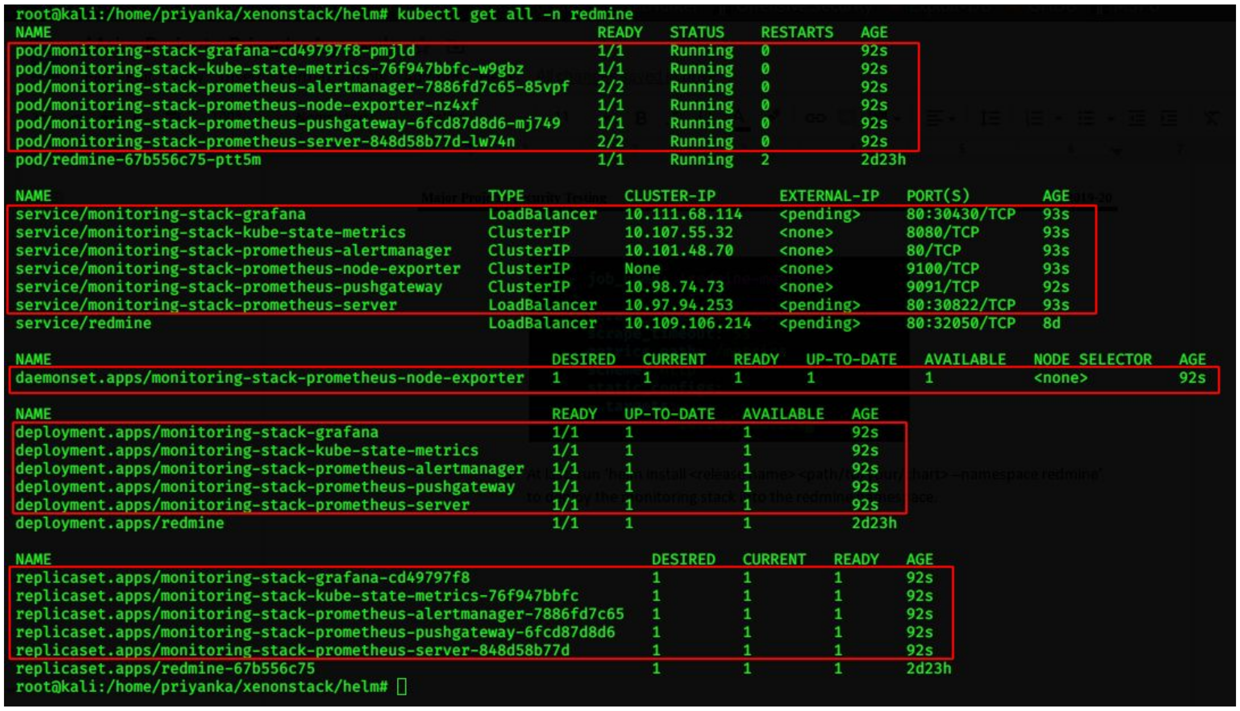
- ❑ As the application is successfully exposing its metrics, we now have to configure prometheus server in such a way that it only monitors application. For this, edit the prometheus.yml section of values.yaml file in the prometheus-grafana chart; wherein you need to scroll down and comment all the specified jobs and create a new job as per the requirement.



```
  - job_name: 'redmine-metrics'

    scrape_interval: 15s
    scrape_timeout: 5s
    metrics_path: /metrics
    scheme: http
    static_configs:
    - targets:
            - 10.109.106.214
```

Fig 22: Prometheus job for monitoring application metrics

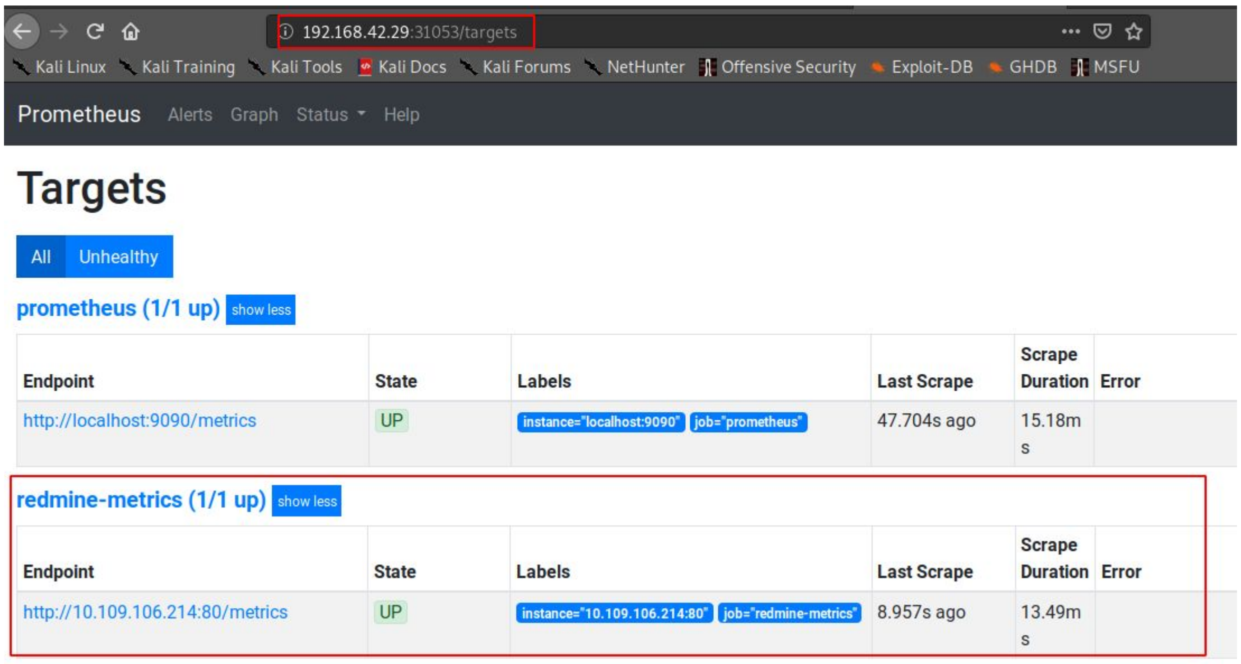
- ❑ At last, run ‘helm install <release name> <path/to/your/chart> --namespace redmine’ to deploy the monitoring stack into the redmine namespace.



```
root@kali:/home/priyanka/xenonstack/helm# kubectl get all -n redmine
NAME                                                        READY   STATUS    RESTARTS   AGE
pod/monitoring-stack-grafana-cd49797f8-pmjld                1/1     Running   0          92s
pod/monitoring-stack-kube-state-metrics-76f947bbfc-w9gbz    1/1     Running   0          92s
pod/monitoring-stack-prometheus-alertmanager-7886fd7c65-85vpf 2/2   Running   0          92s
pod/monitoring-stack-prometheus-node-exporter-nz4xf         1/1     Running   0          92s
pod/monitoring-stack-prometheus-pushgateway-6fcd87d8d6-mj749 1/1    Running   0          92s
pod/monitoring-stack-prometheus-server-848d58b77d-lw74n     2/2     Running   0          92s
pod/redmine-67b556c75-ptt5m                                 1/1     Running   2          2d23h

NAME                                             TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
service/monitoring-stack-grafana                 LoadBalancer   10.111.68.114    <pending>     80:30430/TCP   93s
service/monitoring-stack-kube-state-metrics      ClusterIP      10.107.55.32     <none>        8080/TCP       93s
service/monitoring-stack-prometheus-alertmanager ClusterIP      10.101.48.70     <none>        80/TCP         93s
service/monitoring-stack-prometheus-node-exporter ClusterIP     None             <none>        9100/TCP       93s
service/monitoring-stack-prometheus-pushgateway  ClusterIP      10.98.74.73      <none>        9091/TCP       92s
service/monitoring-stack-prometheus-server       LoadBalancer   10.97.94.253     <pending>     80:30822/TCP   93s
service/redmine                                  LoadBalancer   10.109.106.214   <pending>     80:32050/TCP   8d

NAME                                                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/monitoring-stack-prometheus-node-exporter    1         1         1       1            1           <none>          92s

NAME                                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/monitoring-stack-grafana                    1/1     1            1           92s
deployment.apps/monitoring-stack-kube-state-metrics         1/1     1            1           92s
deployment.apps/monitoring-stack-prometheus-alertmanager    1/1     1            1           92s
deployment.apps/monitoring-stack-prometheus-pushgateway     1/1     1            1           92s
deployment.apps/monitoring-stack-prometheus-server          1/1     1            1           92s
deployment.apps/redmine                                     1/1     1            1           2d23h

NAME                                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/monitoring-stack-grafana-cd49797f8                1         1         1       92s
replicaset.apps/monitoring-stack-kube-state-metrics-76f947bbfc    1         1         1       92s
replicaset.apps/monitoring-stack-prometheus-alertmanager-7886fd7c65 1       1         1       92s
replicaset.apps/monitoring-stack-prometheus-pushgateway-6fcd87d8d6 1        1         1       92s
replicaset.apps/monitoring-stack-prometheus-server-848d58b77d     1         1         1       92s
replicaset.apps/redmine-67b556c75                                 1         1         1       2d23h
root@kali:/home/priyanka/xenonstack/helm# 
```

Fig 23: Deployment of monitoring stack

- ❑ Since the monitoring stack is deployed successfully, access the prometheus server through the browser and verify application monitoring.



Targets

All Unhealthy

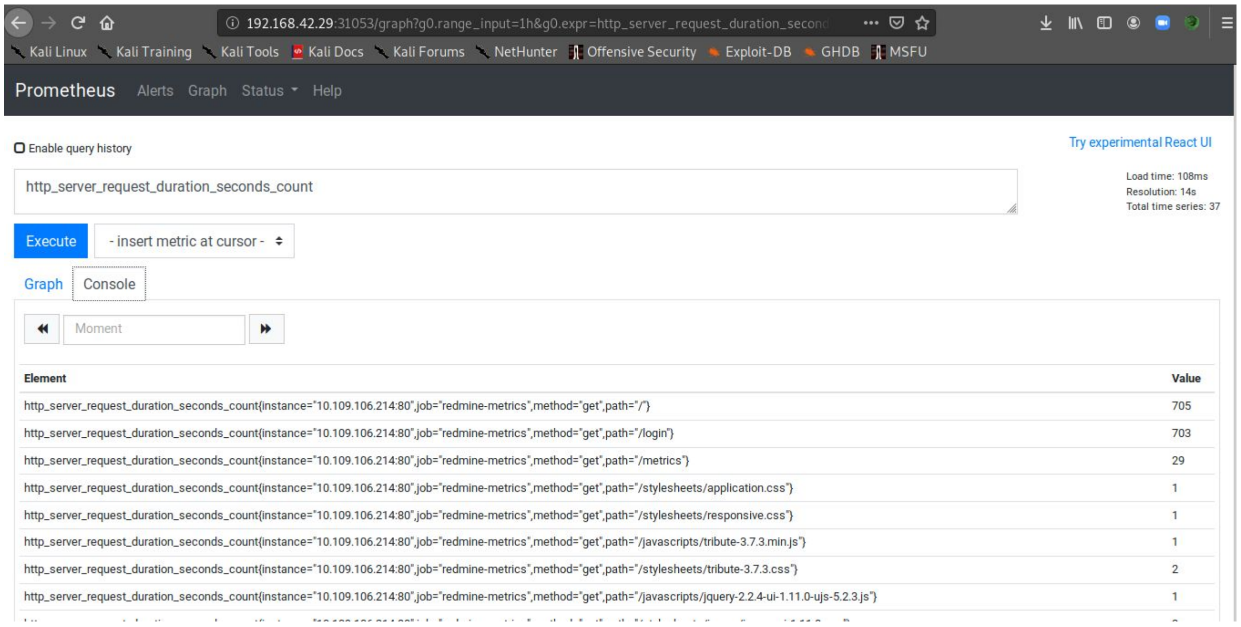
prometheus (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	47.704s ago	15.18ms	

redmine-metrics (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.109.106.214:80/metrics	UP	instance="10.109.106.214:80" job="redmine-metrics"	8.957s ago	13.49ms	

Fig 24: Accessing Prometheus



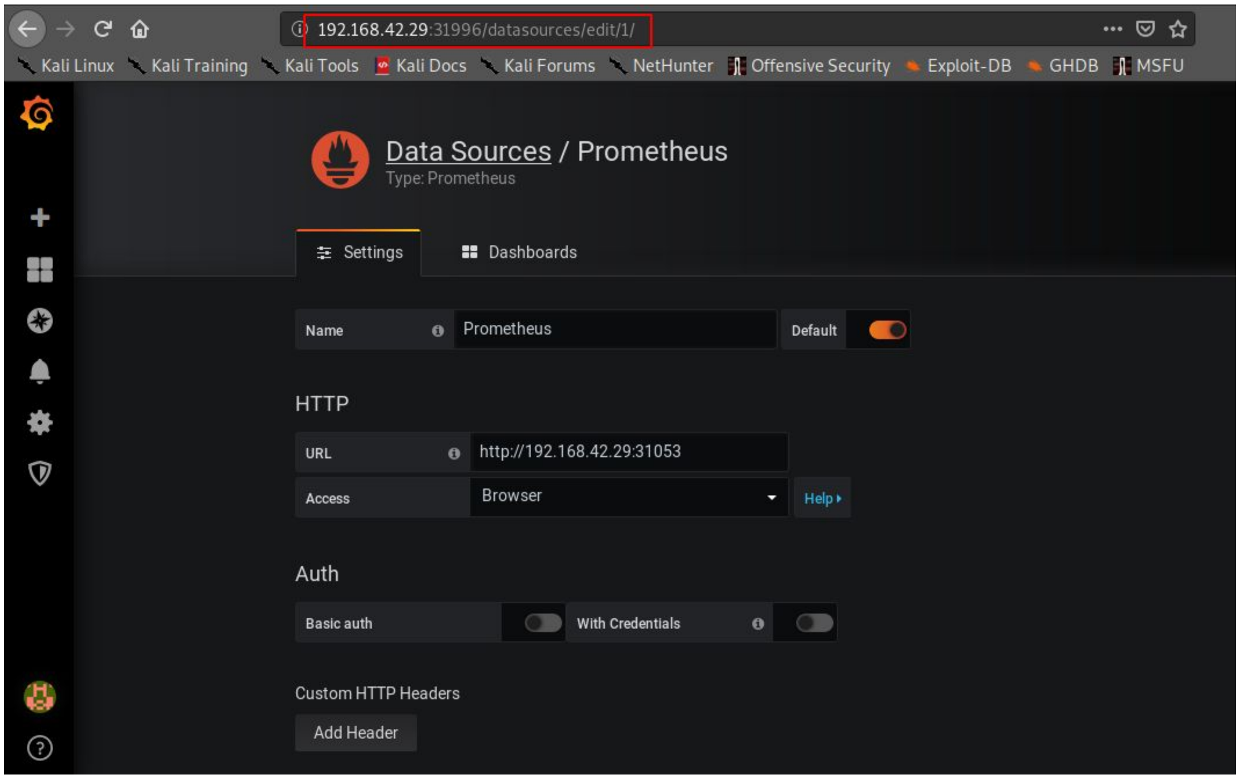
Load time: 108ms
Resolution: 14s
Total time series: 37

```
http_server_request_duration_seconds_count
```

Element	Value
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/"}	705
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/login"}	703
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/metrics"}	29
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/stylesheets/application.css"}	1
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/stylesheets/responsive.css"}	1
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/javascripts/tribute-3.7.3.min.js"}	1
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/stylesheets/tribute-3.7.3.css"}	2
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/javascripts/jquery-2.2.4-ui-1.11.0-ujs-5.2.3.js"}	1

Fig 25: Querying metrics in Prometheus and monitoring the redmine application

❑ At last, access grafana through the browser, log in, add prometheus as a data source and visualize the metrics graphically.

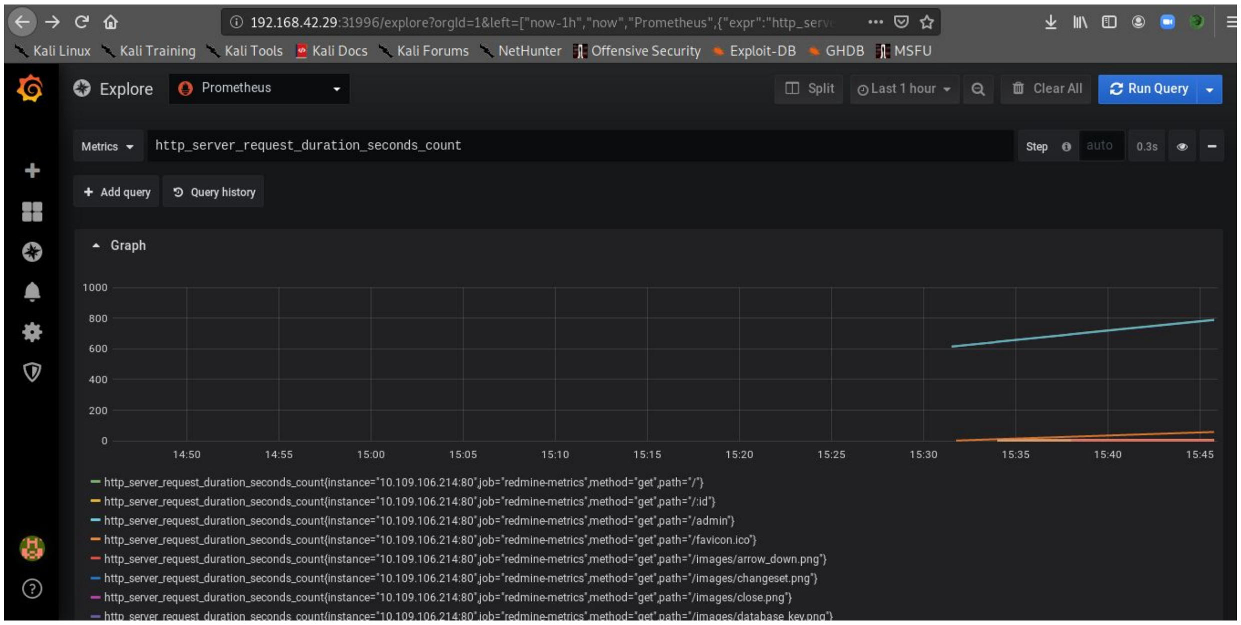


Name: Prometheus
Default: on

HTTP
URL: http://192.168.42.29:31053
Access: Browser

Auth
Basic auth: off
With Credentials: off

Fig 26: Accessing Grafana and setting Prometheus as a Data-Source



Metrics: `http_server_request_duration_seconds_count`

```
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/:id"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/admin"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/favicon.ico"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/images/arrow_down.png"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/images/changeset.png"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/images/close.png"}
http_server_request_duration_seconds_count{instance="10.109.106.214:80",job="redmine-metrics",method="get",path="/images/database_key.png"}
```

Fig 27: Metrics visualization through Grafana

❑ The task is thus completed successfully with the best approach possible.

3. Conclusion

3.1 Conclusion

The entire project was thus completed successfully with the best approach possible to fulfill all the given requirements.

As a conclusion, I would like to state that the primary strength of Kubernetes is its modularity and generality. Nearly every kind of application that you might want to deploy you can fit within Kubernetes, and no matter what kind of adjustments or tuning you need to make to your system, they're generally possible. Of course, this modularity and generality come at a cost, and that cost is a reasonable amount of complexity. Understanding how the APIs and components of Kubernetes work is critical to successfully unlocking the power of Kubernetes to make your application development, management, and deployment easier and more reliable.

Likewise, understanding how to link Kubernetes up with a wide variety of external systems and practices as varied as an on-premises database and a Continuous Delivery system is critical to efficiently making use of Kubernetes in the real world.

Throughout this project I have worked to provide concrete real-world experience on specific topics that will likely be encountered whether for a newcomer to Kubernetes or an experienced administrator.

3.2 Future scope

Looking at the rate of Docker and Kubernetes adoption, it seems that these technologies will take over much of the IT world either in their original form or as PaaS. Containerization makes development and deployment unified and predictable, so in the nearest future we will see companies leave their legacy applications and embark on a container journey as much for their financial benefit as for increased service delivery.

Speaking more broadly, I can think of two key trends:

Community-driven development

Being an Open Source orchestrator, Kubernetes has built a large community of people who work on Kubernetes stability, security, etc. and offer the result as standalone Kubernetes distros. It means that we will see a growing family of K8s-based orchestrators that introduce new features or fine-tune existing ones, adding to the overall performance of K8s.

Managed Kubernetes

More and more companies are shifting towards managed Docker/ Kubernetes services instead of running and maintaining clusters on their own. Accordingly, what we will witness (and are already witnessing) is a growing number of small and enterprise-scale PaaS that offer fully managed K8s clusters and maintain the underlying infrastructure.

We started using Docker and Kubernetes for our own purposes and soon realized that however powerful an orchestrator it might be, it is a complex and challenging product that requires much knowledge and experience to run and maintain it.

REFERENCES

The References taken for the development of this project and documentation are:

- https://kubernetes.io/docs
- https://docs.docker.com/
- https://www.tutorialspoint.com/kubernetes/kubernetes_architecture.htm
- https://www.digitalocean.com/community/tutorials/an-introduction-to-helm-the-package-manager-for-kubernetes
- https://prometheus.io/
- http://www.techplayon.com/what-is-prometheus/
- https://grafana.com/docs/grafana/
- https://www.thoughtworks.com/insights/blog/kubernetes-exciting-future-developers-and-infrastructure-engineering-0
- https://helm.sh/