Requirements:

Develop a key-value store using Kubernetes (k8s), FastAPI, and Huey as a REDIS queue that can scale reliably across multiple pods/deployments. We're looking for your expertise in creating scalable, robust systems, with a keen eye on documentation, process, code style, minimalism, and clarity.

Approach:

At J.P Morgan Chase, as a Full Stack developer, I have worked with System Design, creating scalable systems, converting monolithic apps to micro services architecture and two different critical migrations. I have worked extensively in Java, Spring MVC Framework and Springbok - alongside Kafka, Redis Cache and Kubernetes. Having worked with Python only for Data Mining and Machine Learning use cases - this task is itself a challenge for me, to recreate the entire system - in Python. I do have basic experience in Docker and Kubernetes but have never really deployed Python apps by myself in Kubernetes so this overall experience was fun!

This document assumes that the reader does not have any background in creating scalable, robust systems especially with Python. I want the reader to understand my thought process behind designing any backend system and the fact that programming language is not as critical as the different concepts involved in system design. With every framework, comes it's set of pros and cons, and I have tried to address everything here as much as I can :)

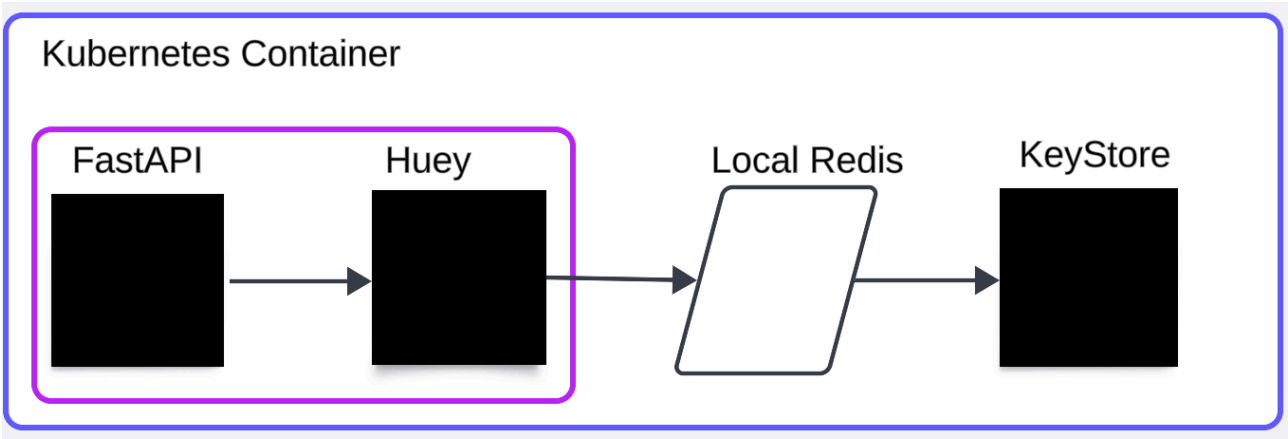
Steps	Description
A.	Breaking down the requirements
B.	System Design (Naive)
C.	Evolving the design and how?
D.	System Design (A more detailed overview)
E.	Explanation of Code Structure & Design
F.	Deployment Overview
G.	Deployment Steps - Huey Application
H.	Creating a repository on docker.io
I.	Deployment Steps - Redis Service and RedisJSON service
J.	Challenges faced!

A. Breaking down the requirements

We break down the given requirements into its individual components to understand them and make better design decisions.

Components	What is it
Key Value Store	A simple python dictionary or as complicated as a Redis Cache/ RedisJSON
Kubernetes	Orchestration platform to automate deployments, scaling and management of apps
FastAPI	Python equivalent of SpringBoot controllers (with much more capabilities)
Redis Queue	Supports running background jobs (like sending emails); has data structure support like Lists/Stacks/Queues; Message Acknowledgement feature; Supports Asynchronous tasks
Huey as Redis Queue	To create tasks that can be sent to Redis queue; Stores tasks metadata like task state, result, etc

B. System Design Diagram (Naive)



This diagram explains the basic data flow of the system. We have FastAPI controller that takes the request from the user, creates a Huey task from it and connects to the Redis Server. It then accesses the key Store and completes its operation. The entire application is deployed in the Kubernetes container to help with automated deployment and scaling of the app. We still are not much aware on the details for each components, hence we have kept them in black box (quite literally!)

Is this design good?

Yes, for a single user system. It uses Huey and Redis, so it is already faster than most applications. But will it suffice when we have multiple users and app deployed across multiple components? No, we need to think upon the scalability and robustness of our application in a multithreaded production environment. What if there are multiple users

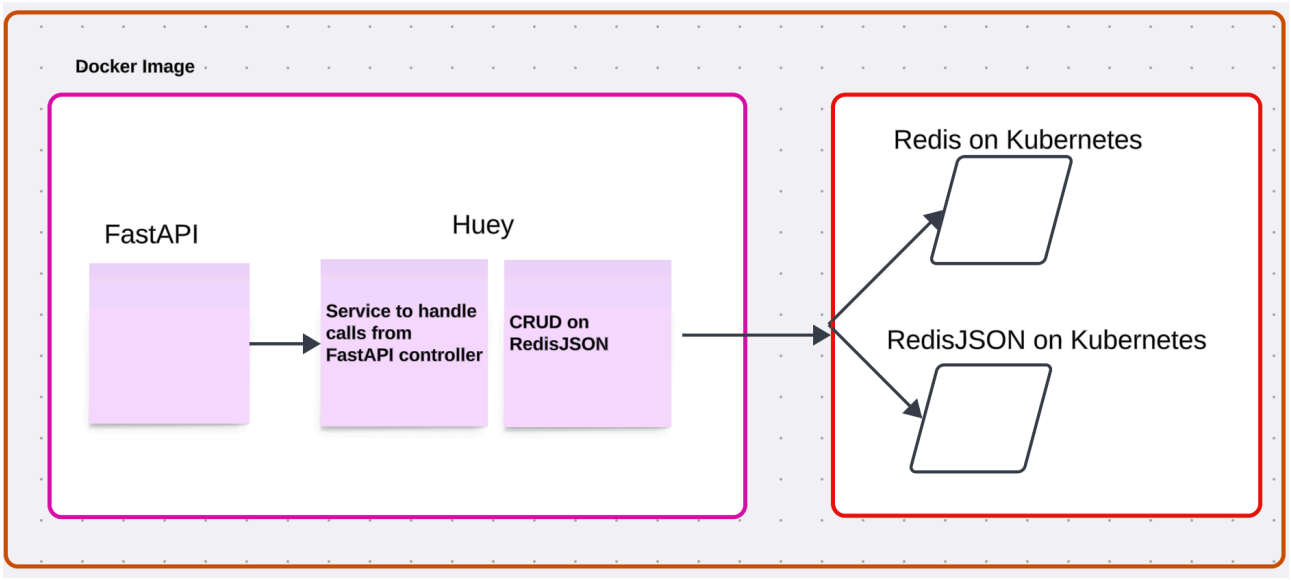
trying to access our key store? What if there is no concurrency control? Will our application be fast enough? And most importantly, will it return right data from key store every time?

C. Evolving the design and how?

Original Component	Changed to	Why?	How?
Key Value Store	RedisJSON	Redis is an in-memory data store, making it incredibly fast for read and write operations. RedisJSON builds on top of Redis; Scalability; Indexing & Querying; Publisher-subscriber messaging model; data structure flexibility	Create and use Redis service, RedisJSON service and expose them from a Kubernetes container
Redis Server on local env	Redis Server on Kubernetes	Scalability; High availability; Resource efficiency	Use Docker Desktop
Huey	Huey code uses async and await, connects to Redis Server on Kubernetes	Easier management and monitoring;	Use Huey python library
FastAPI	To include better control for Huey tasks	As an entry point to the application, it needs to be robust in terms of controls and exception handling in case of failure.	Better structured code design similar to Spring MVC framework

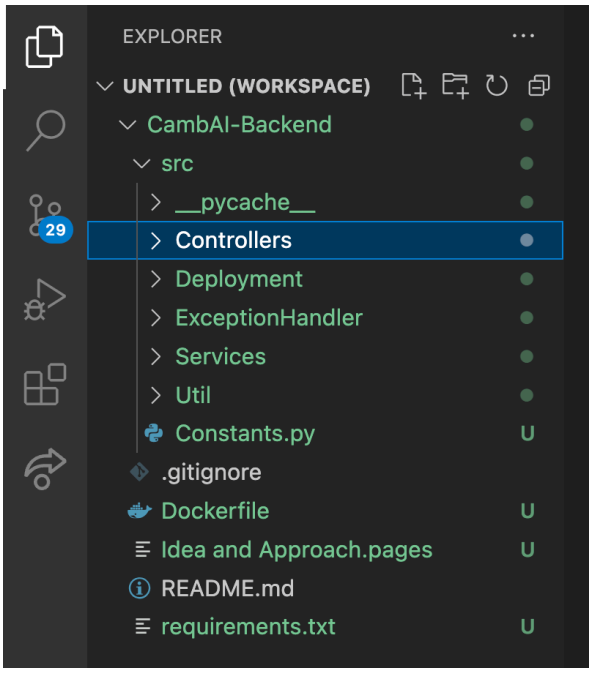
D. System Design (A more detailed overview)

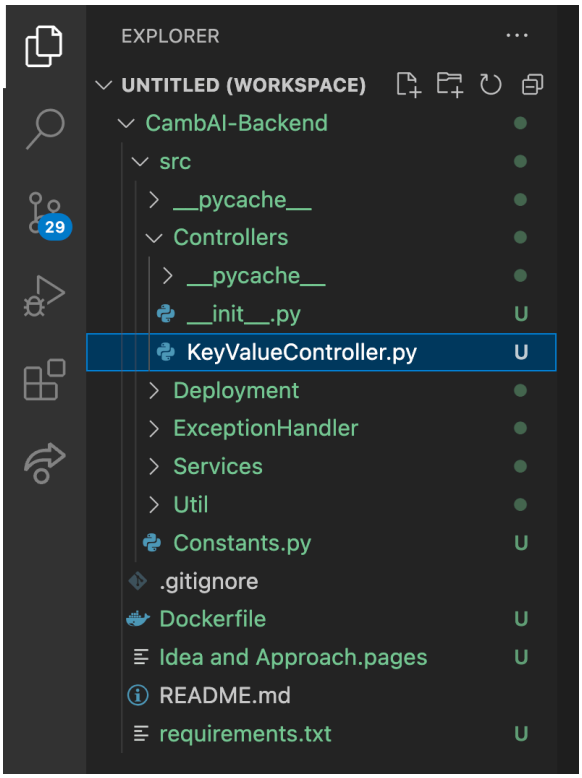
It is generally a good practice to keep the RedisService+RedisJSON deployed on a separate container and the code (Huey application) deployed in a separate one. The main reason is reason is that in a scenario where application on one container fails, it does not impact the functionality of the rest of the application. We can scale our application both horizontally and vertically very easily.



E. Explanation of code structure and design

1. Controllers



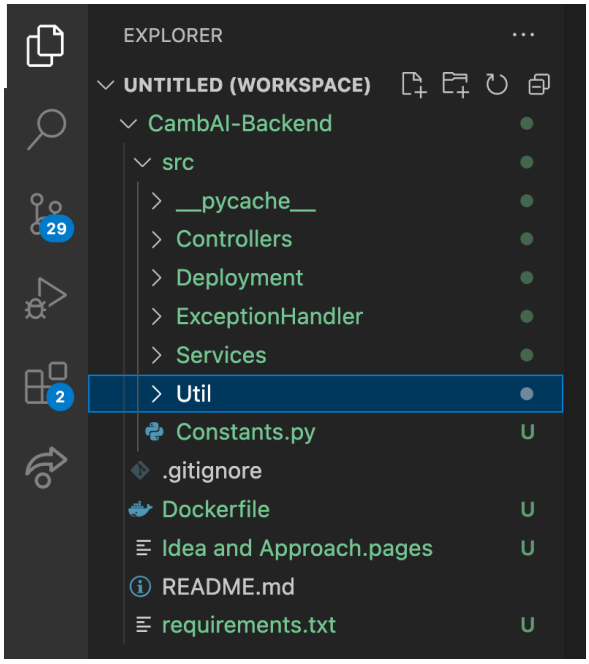


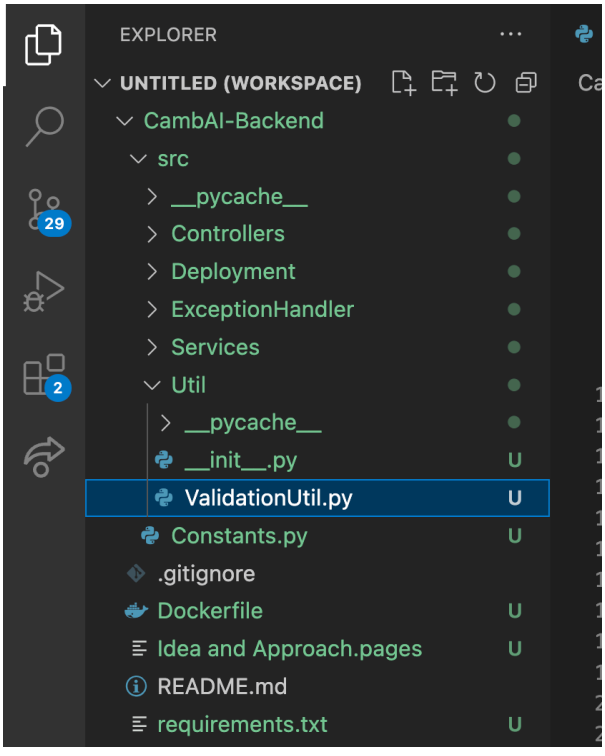
Purpose:
The controllers folder has the implementation for FastAPI, similar to the SpringBoot Controller. It will accept requests from the user for various operations on KeyStore. It has an init.py file to be recognised as an internal module/Package by Python.

Design:
The controller is designed in a way to handle all API calls to Key Store. It does not connect to the RedisJSON directly so encapsulates most of the implementation details from the user. We do not give away the key/values by printing them in the loggers. The controller also validates all requests coming to it, making sure that there are no malicious keys/values sent to the KeyStore. We have taken care of putting a check in place wherever needed, if a key is already existing or not. In case of a scenario where a call fails, we are handling it gracefully by using a custom exception class. The API caller will not be exposed to all details of the exception, thereby protecting our inner service layer and implementation details from hackers. In production scenario when we deploy our code across multiple containers, we have to be careful about any concurrency issues arising from deployment across multiple instances. Hence, we make use of asuncion library, to include async and wait across all Huey tasks. We are using JSON as an appropriate serialisation format.

Future Scope:
We can implement authentication and authorisation for user depending on their roles and only then allow them to perform any kind of operations on the KeyStore.

2. Utils



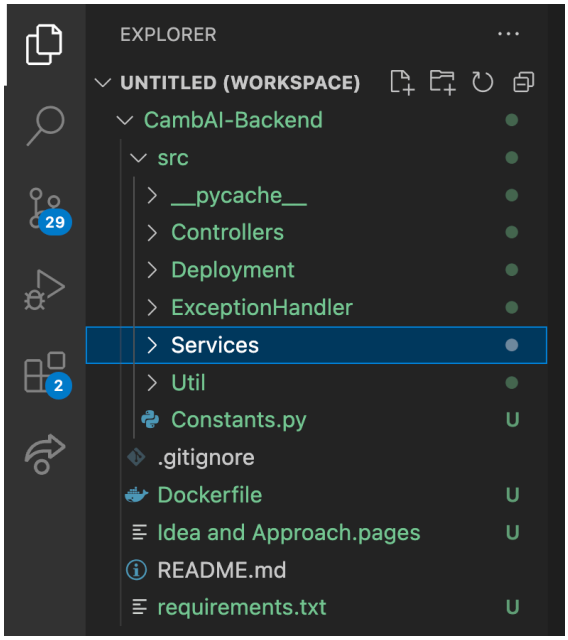


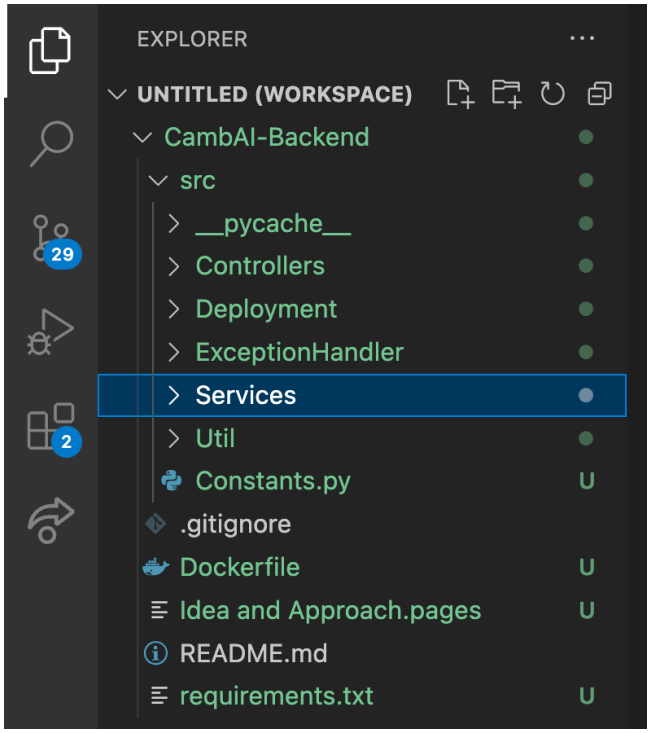
Purpose:
The Util folder is to include all types of common utilities that can be used across the application. It has an init.py file to be recognised as an internal module/Package by Python.

Design:
We have two methods that validate the keys and values passed to the key store, to avoid any malicious input. This validation check is common and can be extended across different parts of the application, for different use cases.

Future Scope:
We can implement more utilities and put them in this folder, example, FileUtil class that has all methods for CRUD operations on a file. We can have all SQL utils in this folder, set of all functions implemented that can be reused for any SQL query (examples include functions that calculate date/time difference between two dates and provide output as number of days). Such SQL utils can be useful for Cron jobs.

3. Services



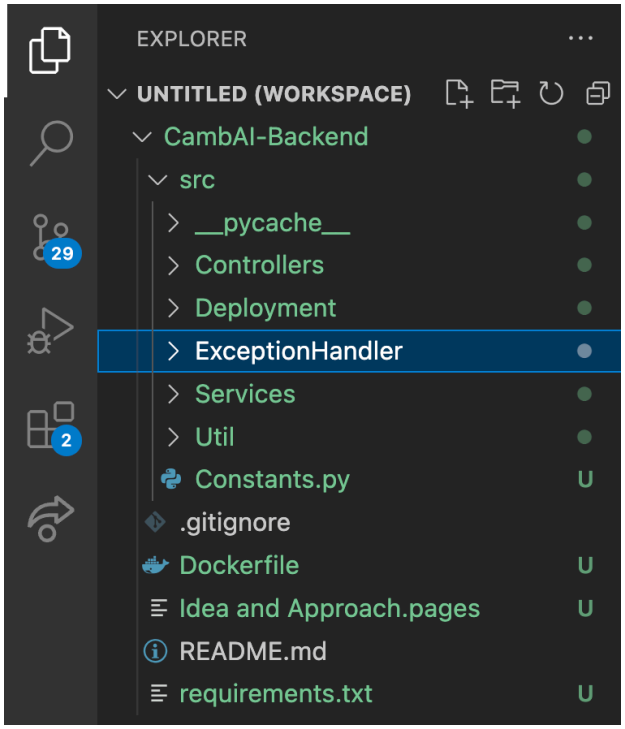


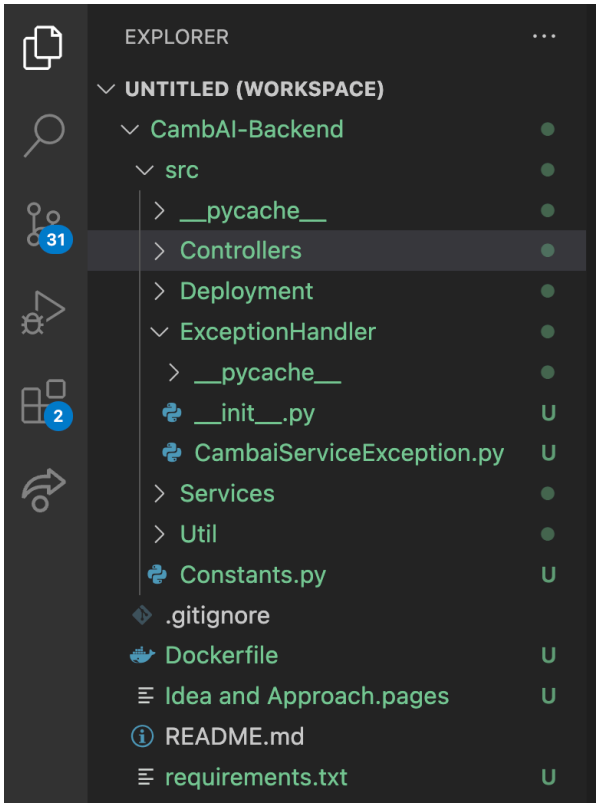
Purpose:
The Services folder is to include all types of services that application needs. It's purpose is different from the controller and handles communication between Controllers and Storage Components (Redis in our case). It has an init.py file to be recognised as an internal module/ Package by Python.

Design:
We have Huey application code in our service. We create different Huey tasks and put them on Redis queue. We have connections we establish to external components, Redis service, RedisJSON service. We are using RedisHuey here to create a Huey instance. We also have elaborate logging in this class as we have to get specific application exception details, not like what we have in controllers. We also have methods that directly set, get and delete keys/ objects in our key-value store. For now, we only have the following -
Get all keys, Set a key-value, Get value for a key, Delete a key.
We are using asuncion to create synchronisation needed within tasks to create/access key-value store.

Future Scope:
We can extend and include all functionalities that RedisJSON provides on key-store, in our service layer. Redis JSON, it has multiple benefits and lot of cool functions that can be needed on a key-value store - Easy search, retrieval and update. We can also include message state/ack, etc that might be needed for loggers; Rollback or Messages retries; data replication, consistency checks.

4. Exception Handling

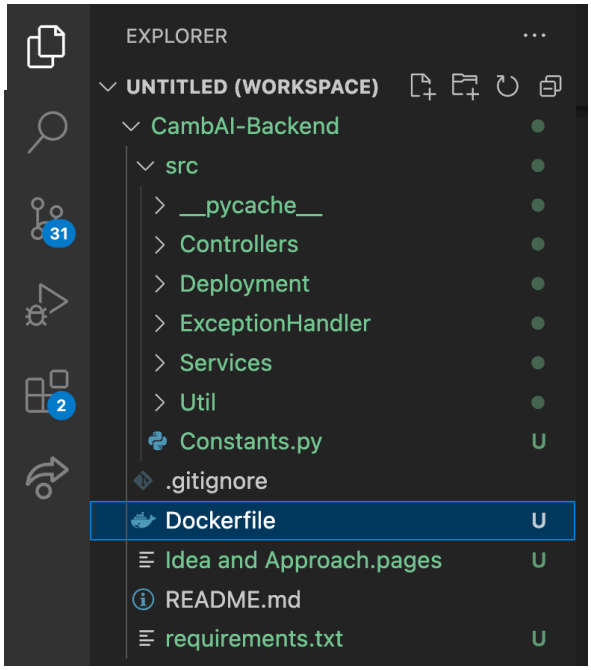


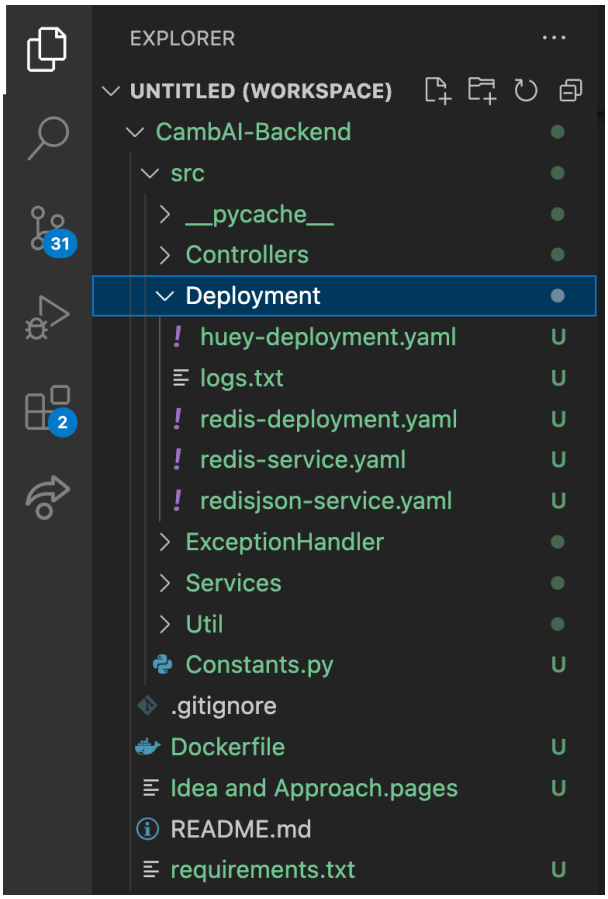


Purpose:
Exception handling, we have created a custom exception to handle all kinds of exception gracefully.

Future Scope:
We can include all kinds of graceful exceptions , eg. Error code 800 - to show to user that mean our application is down currently, without having to give away all implementation details.

F. Deployment Overview





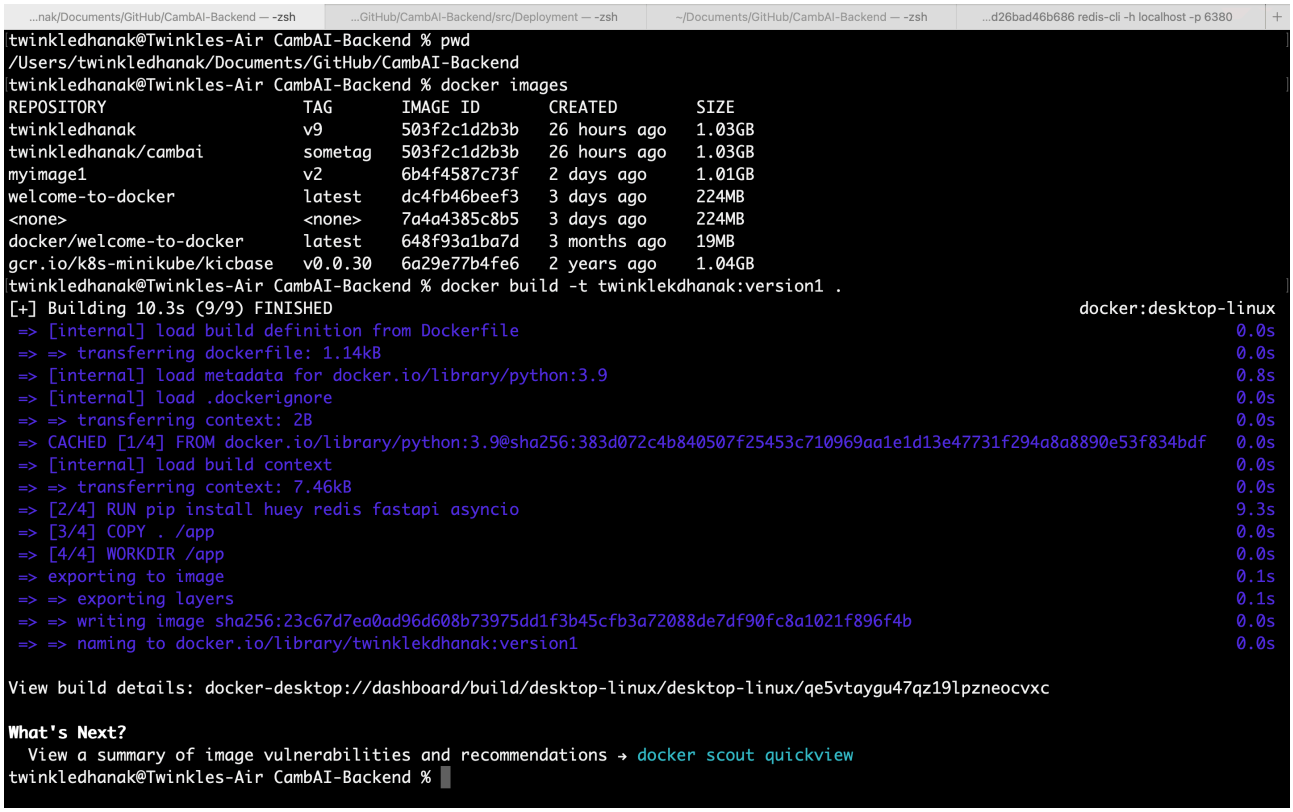
We have multiple components that we have to deploy onto Kubernetes,
1. Huey Application deployment
2. Redis Service on Kubernetes
3. RedisJSON service on Kubernetes

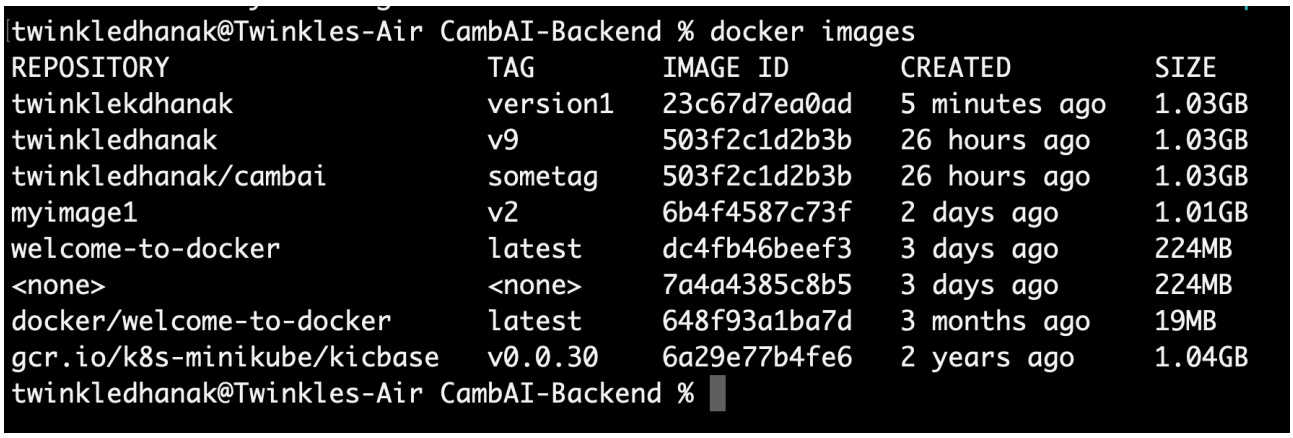
The Dockerfile we have is for the entire application, it helps to build an image for our application. We specify the code, working directory and command to run our application in Kubernetes. We have to provide all dependencies will be needed to run our application, which can be specified in requirements.txt or directly in our Dockerfile.

G. Deployment Steps - Huey Application

In order to deploy our application onto Kubernetes, We have used Docker Desktop. We containerize our Huey application and create a Kubernetes deployment YAML file (huey-deployment.yaml) to deploy the Huey application in Minikube. Include specifications for the Huey container, such as image, environment variables (to configure Huey to use Redis), ports, and resources. We start by building our docker image.

docker build -t twinkledhanak:version1 .

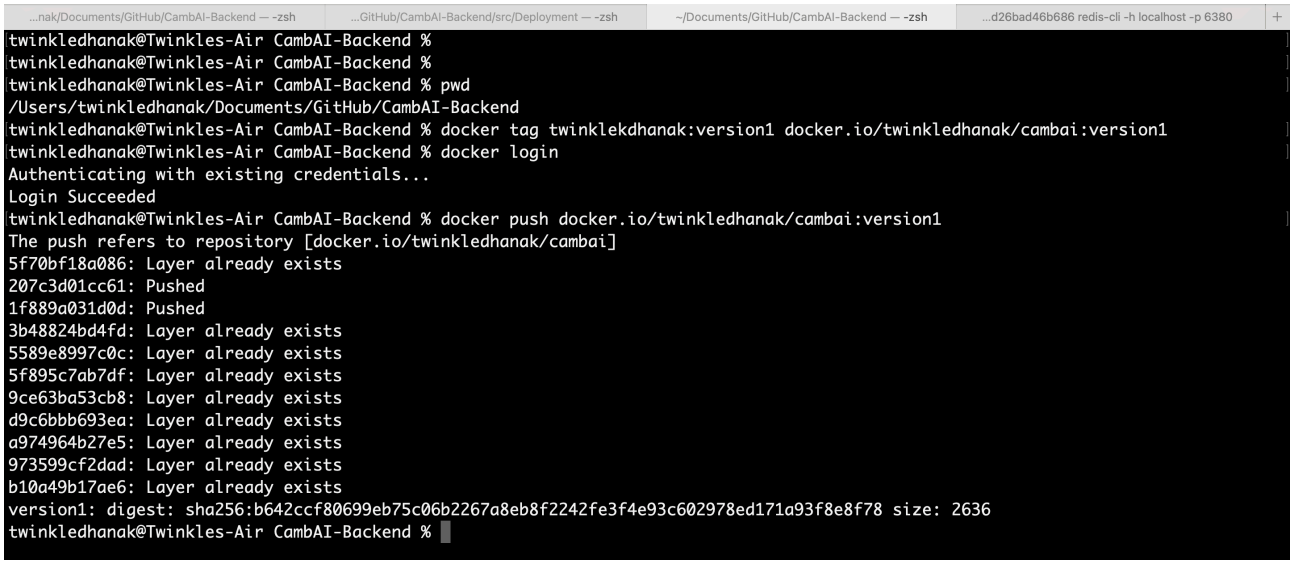


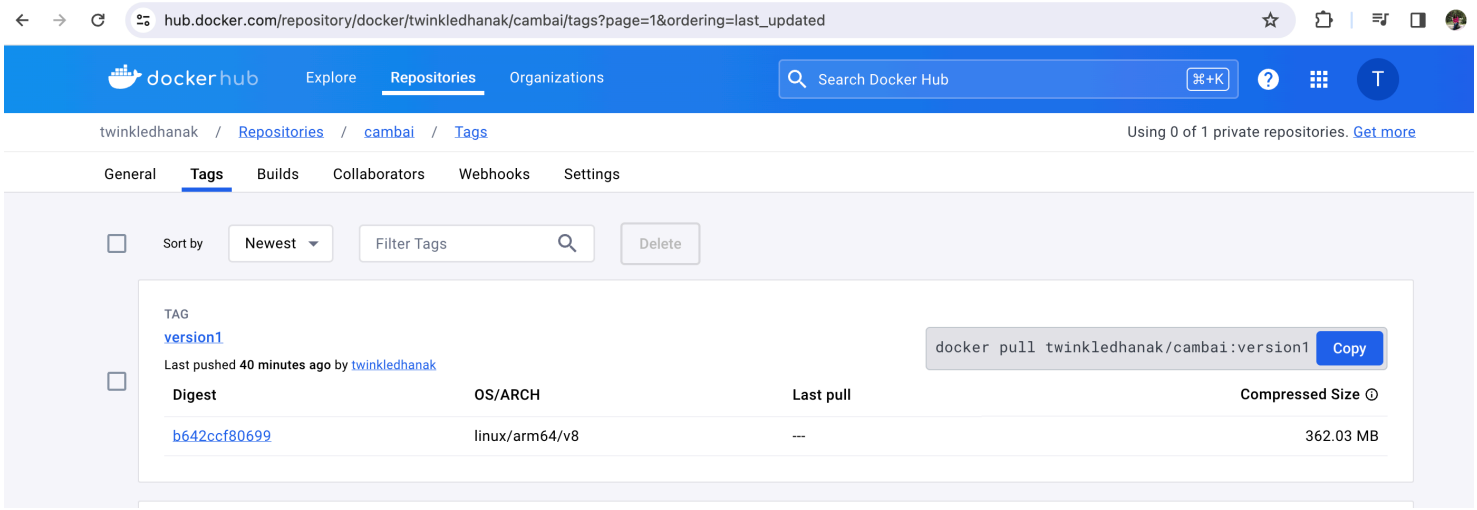


twinkledhanak:version1 is the docker image we have created

H. Creating an image on docker.io registry

docker tag local_image:tag docker.io/your_username/repository_name:tag
docker login
docker push docker.io/your_username/repository_name:tag



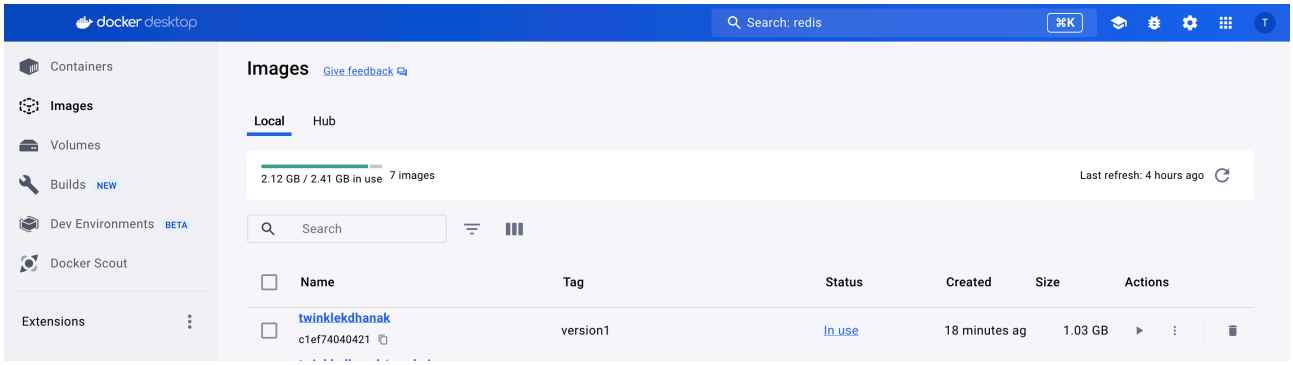


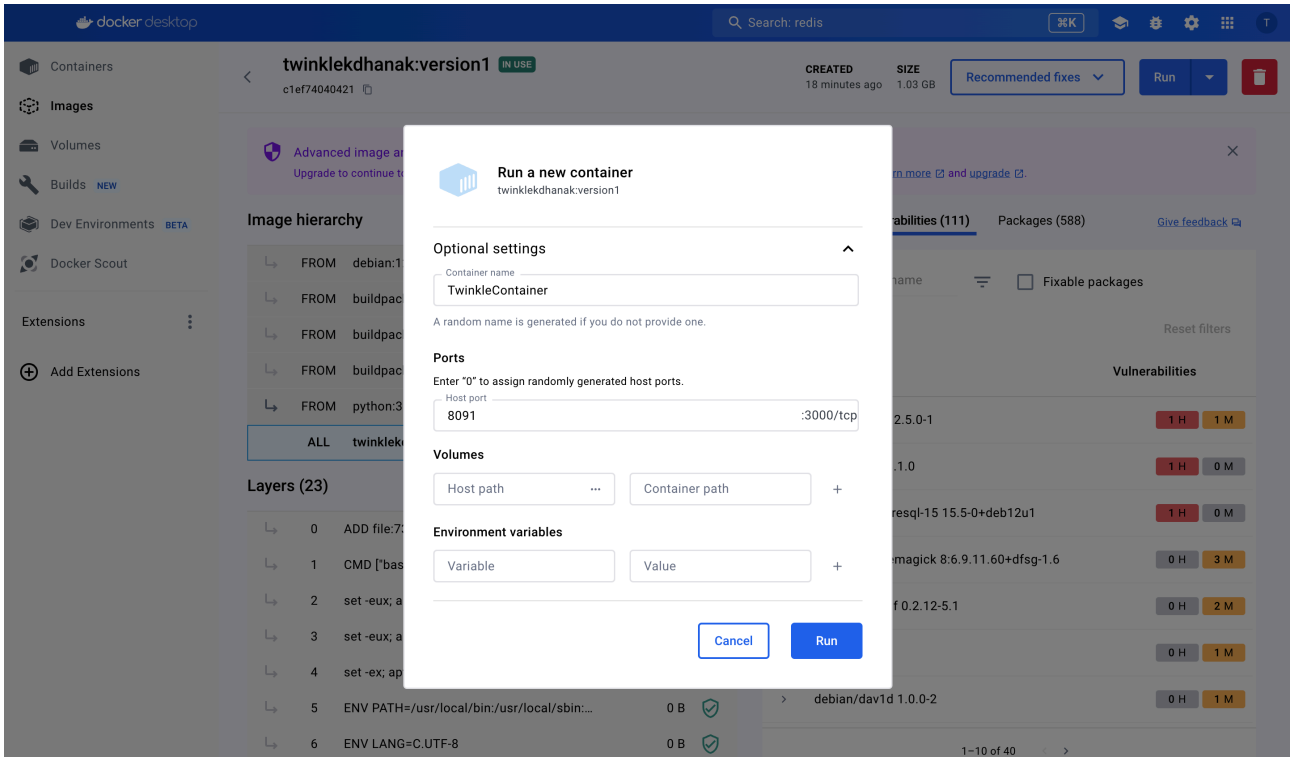
I. Deployment Steps for Redis Service and RedisJSON service

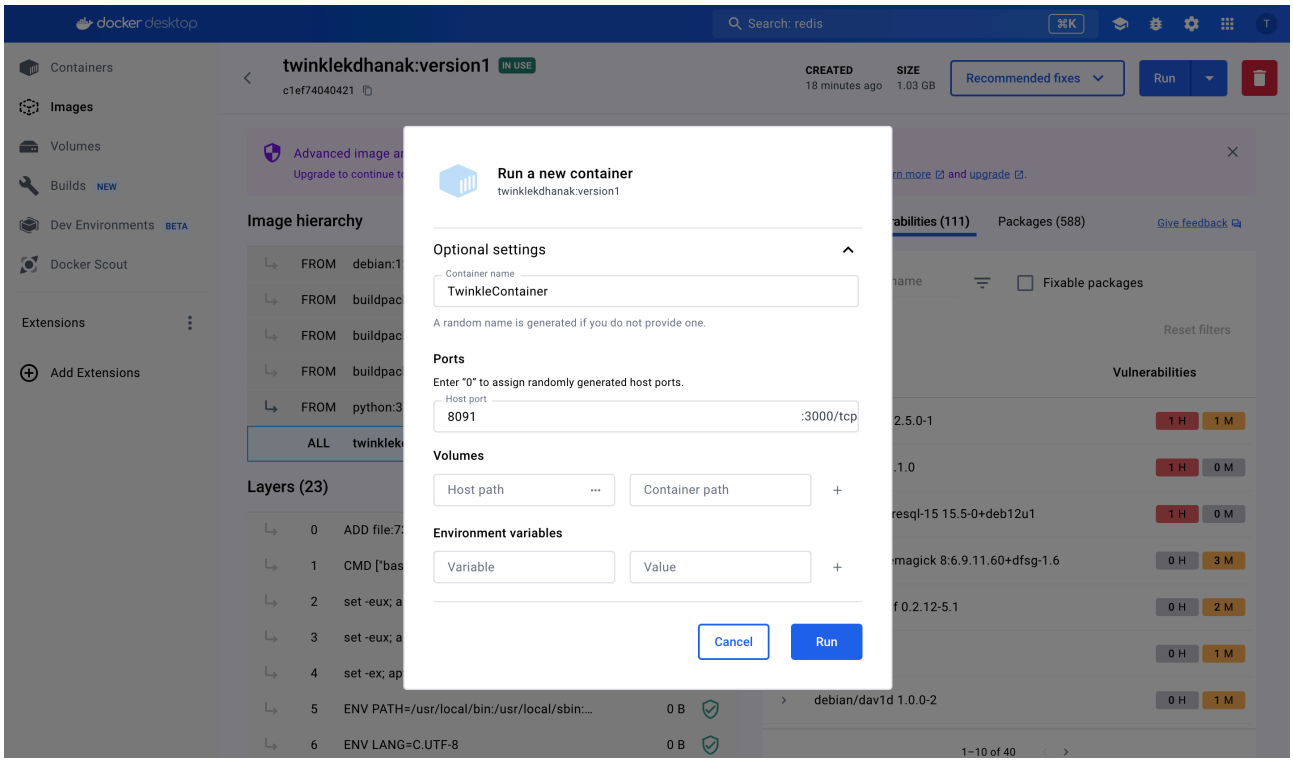
minikube start
kubectl config use-context minikube
kubectl apply -f redis-deployment.yaml
kubectl apply -f redis-service.yaml
kubectl apply -f redisjson-service.yaml
kubectl apply -f huey-deployment.yaml

```
...nak/Documents/GitHub/CambAI-Backend — -zsh   ...GitHub/CambAI-Backend/src/Deployment — -zsh   ~/Documents/GitHub/CambAI-Backend — -zsh   ...d26bad46b686
twinkledhanak@Twinkles-Air Deployment % pwd
/Users/twinkledhanak/Documents/GitHub/CambAI-Backend/src/Deployment
twinkledhanak@Twinkles-Air Deployment % minikube start
😄  minikube v1.25.2 on Darwin 14.3.1 (arm64)
🎉  minikube 1.32.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.32.0
💡  To disable this notice, run: 'minikube config set WantUpdateNotification false'

✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🏃  Updating the running docker "minikube" container ...
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass

❗  /usr/local/bin/kubectl is version 1.29.1, which may have incompatibilites with Kubernetes 1.23.3.
    ▪ Want kubectl v1.23.3? Try 'minikube kubectl -- get pods -A'
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
twinkledhanak@Twinkles-Air Deployment % kubectl config use-context minikube
Switched to context "minikube".
twinkledhanak@Twinkles-Air Deployment %
```

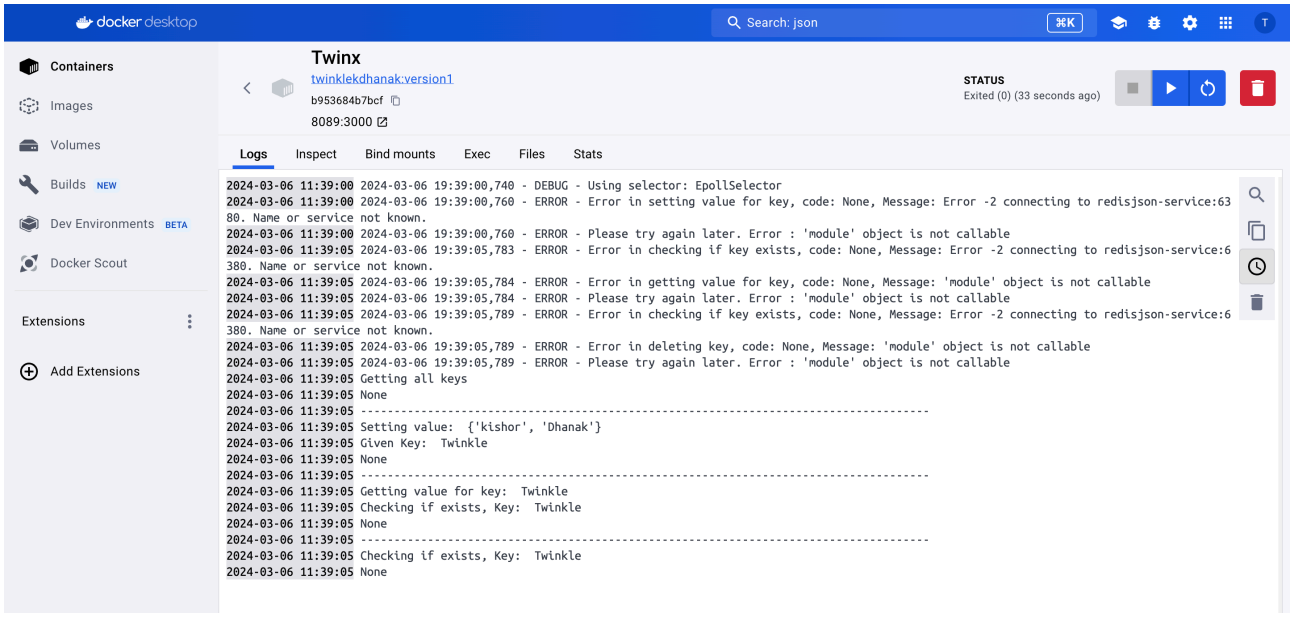
Docker Desktop




docker desktop
Search: redis
Containers
Images
Volumes
Builds NEW
Dev Environments BETA
Docker Scout
Extensions
Add Extensions
twinklekdhanak:version1 IN USE
c1ef74040421
CREATED 18 minutes ago
SIZE 1.03 GB
Recommended fixes
Run
Image hierarchy
Layers (23)
Run a new container
twinklekdhanak:version1
Optional settings
Container name
TwinkleContainer
A random name is generated if you do not provide one.
Ports
Enter "0" to assign randomly generated host ports.
Host port
8091
:3000/tcp
Volumes
Host path
Container path
Environment variables
Variable
Value
Cancel
Run


docker desktop
Search: redis
Containers
Images
Volumes
Builds NEW
Dev Environments BETA
Docker Scout
Extensions
Add Extensions
twinklekdhanak:version1 IN USE
c1ef74040421
CREATED 18 minutes ago
SIZE 1.03 GB
Recommended fixes
Run
Image hierarchy
Layers (23)
Run a new container
twinklekdhanak:version1
Optional settings
Container name
TwinkleContainer
A random name is generated if you do not provide one.
Ports
Enter "0" to assign randomly generated host ports.
Host port
8091
:3000/tcp
Volumes
Host path
Container path
Environment variables
Variable
Value
Cancel
Run

Post Deployment:



J. Challenges Faced

Challenges Faced	Resolution
Various import issues on Container for internal modules and code files	The method to resolve paths for files in containers is different and uses library OS. Deployed the application with code change for using OS to figure out paths for internal modules and files
When deploying Redis on Kubernetes, Error response from daemon: pull access denied for twinkledhanak, repository does not exist or may require 'docker login': denied: requested access to the resource is denied" image=“twinkledhanak:v9	Built a docker image locally with a different tag. Created a new repository on docker.io and pushed this image on the registry with this new tag.
Error -2 connecting to redis-service:6380. Name or service not known.	Still not resolved :)
Status is ImagePullBackOff for pods and services	Resolved. Used correct service and image name in the application.