Continuous Improvement

Release 1.0.0

Adisakshya Chauhan

Dec 12, 2021

GETTING STARTED

CHAPTER

ONE

ABOUT

Many organizations moved away from their monolithic systems that served them well for many years to microservices which is the state-of-the-art software development technique that allows a complex application to be built as a suite of small services, developed around specific areas. Combined, they provide a cohesive set of functionalities and bring important business benefits.

This project embraces the DevOps method of working with microservices which is highly significant and well needed for these technologies to work at their best, highlighting key concepts to scale, synchronize and secure microservices architecture by implementing a notification-delivery system and also considering ways of optimizing the process of provisioning and managing infrastructure resources, automation servers, on-premise and services deployed in multiple environments.

CHAPTER

TWO

COMMUNITY GUIDE

2.1 How can I contribute?

2.1.1 Submit an issue

Submitting an issue, be it a bug report or a feature request or an optimization, is one of the best ways to contribute to this project. Checking if everything works in your system and if the latest commits work properly for you are both good ways to find bugs.

Please search existing issues to avoid creating duplicates.

2.1.2 Improve issues

Some issues are created with missing information (`needs more info`), are not reproducible, or are plain duplicates. Help us finding reproducible steps and closing duplicates.

2.1.3 Comment on issues

We are always looking for more opinions, leaving a comment in the issue tracker is a good opportunity to influence the future direction of Notable.

We also consider the number of ":+1:" an issue has when deciding if we are going to work on it in the Next milestone or not, so be sure to add your ":+1:" to the issues you're most interested in.

2.1.4 Submit a pull request

All versions of the app are open-source, you can fork the repository and modify it, and submit it as a PR or you might want to do that to experiment with radical new ideas.

2.2 Contributor Covenant Code of Conduct

Please note that this project is released with a Code of Conduct. By participating in this project you agree to abide by its terms.

2.2.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to make participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

2.2.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

2.2.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

2.2.4 Scope

This Code of Conduct applies within all project spaces, and it also applies when an individual is representing the project or its community in public spaces. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

2.2.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at adisakshya98@gmail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

2.2.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

CHAPTER

THREE

MICROSERVICES DOCUMENTATION

3.1 Reminder Service

This documentation is imported from official documentation of reminder-service

3.2 Event Service

This documentation is imported from official documentation of event-service

3.3 Notification Service

This documentation is imported from official documentation of notification-service

3.4 Notification Scheduler

This documentation is imported from official documentation of notification-scheduler

CHAPTER

FOUR

CHANGELOG

4.1 Reminder Service

This changelog is imported from official changelog of reminder-service

4.2 Event Service

This changelog is imported from official changelog of event-service

4.3 Notification Service

This changelog is imported from official changelog of notification-service

4.4 Notification Scheduler

This changelog is imported from official changelog of notification-scheduler

CHAPTER

FIVE

CI/CD PIPELINE

What is CI-CD?

Continuous Integration (CI) and Continuous Deployment/Delivery (CD) is a method to frequently deliver software to customers by introducing automation into the stages of software development. CI/CD introduces ongoing automation and continuous monitoring throughout the lifecycle of apps, from integration and testing phases to delivery and deployment. Taken together, these connected practices are often referred to as a "CI/CD pipeline" and are supported by development and operations teams working together.

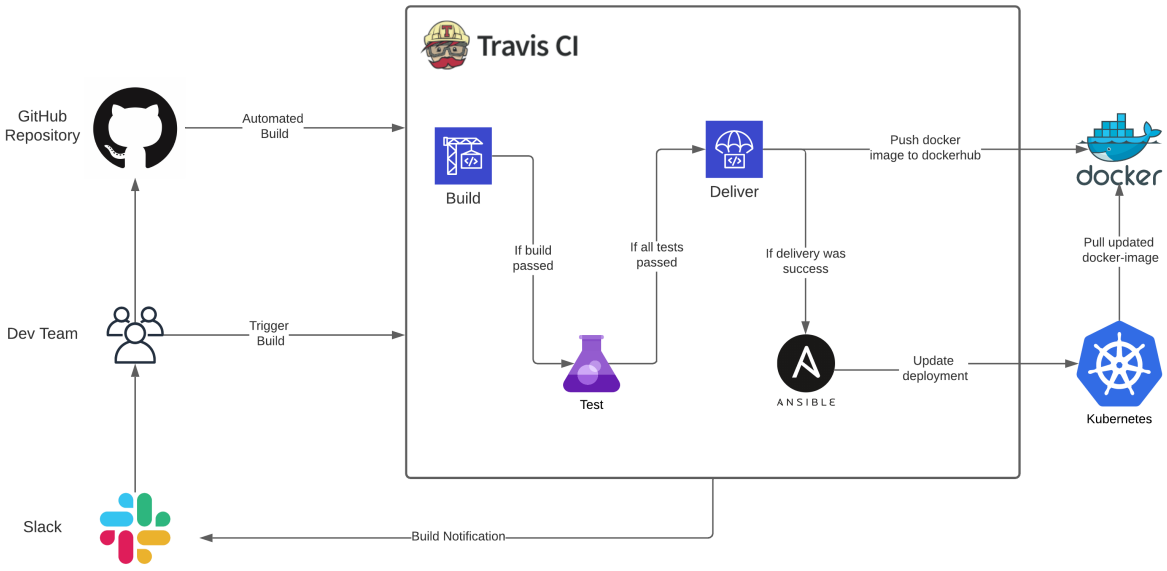
The goal of CI is to establish a consistent and automated method to build, package, and test software. CD picks up the CI artifacts and automate the delivery of application to a selected infrastructure environment.

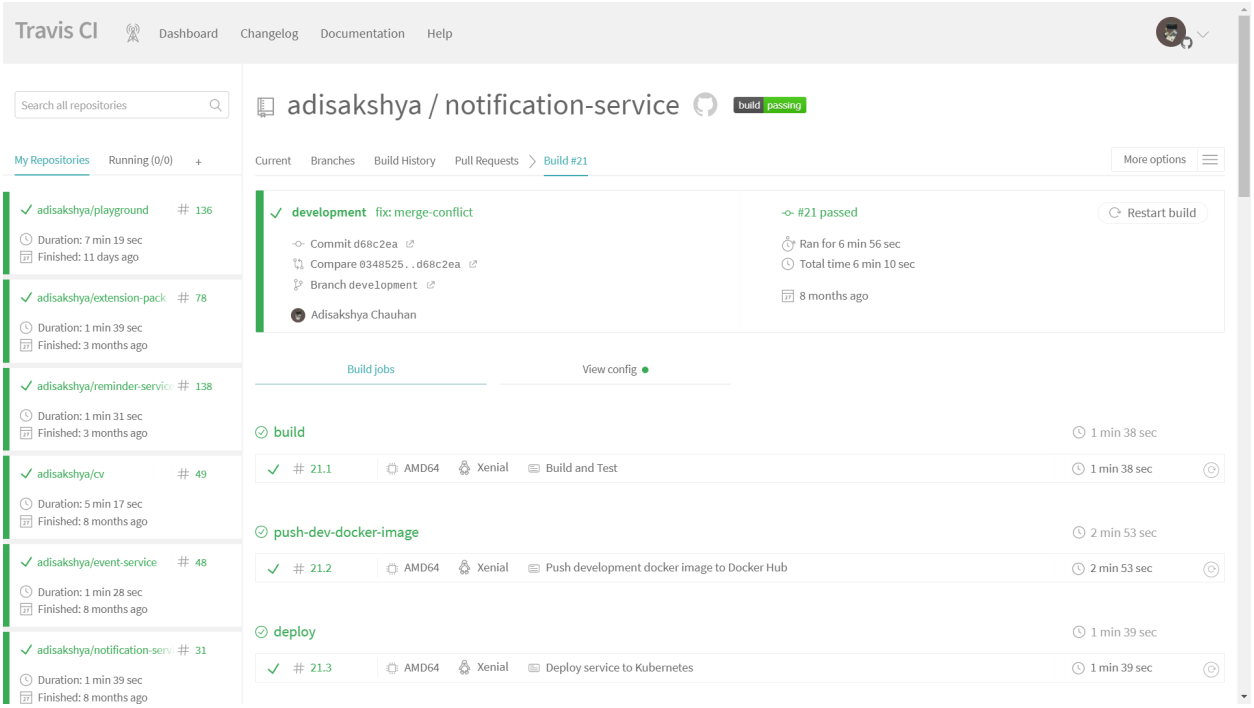
How CI/CD has been used with microservices?

The Notification Delivery system runs in 2 environments namely - 1. Development (or dev) Environment 2. Production Environment

At the source code level these environments can be distinguished using Git branches. Every microservice has 3 git branches namely development, production and master.

For every commit corresponding to a major-minor update, bug-fix, improvements, patches is built, tested and deployed using a CI/CD pipeline powered by Travis CI.





CI/CD stages are different for different branches and environments. Any commit/merge to the development-branch will build the source code and update the development docker-image on DockerHub, same applies for commits/merges on production-branch.

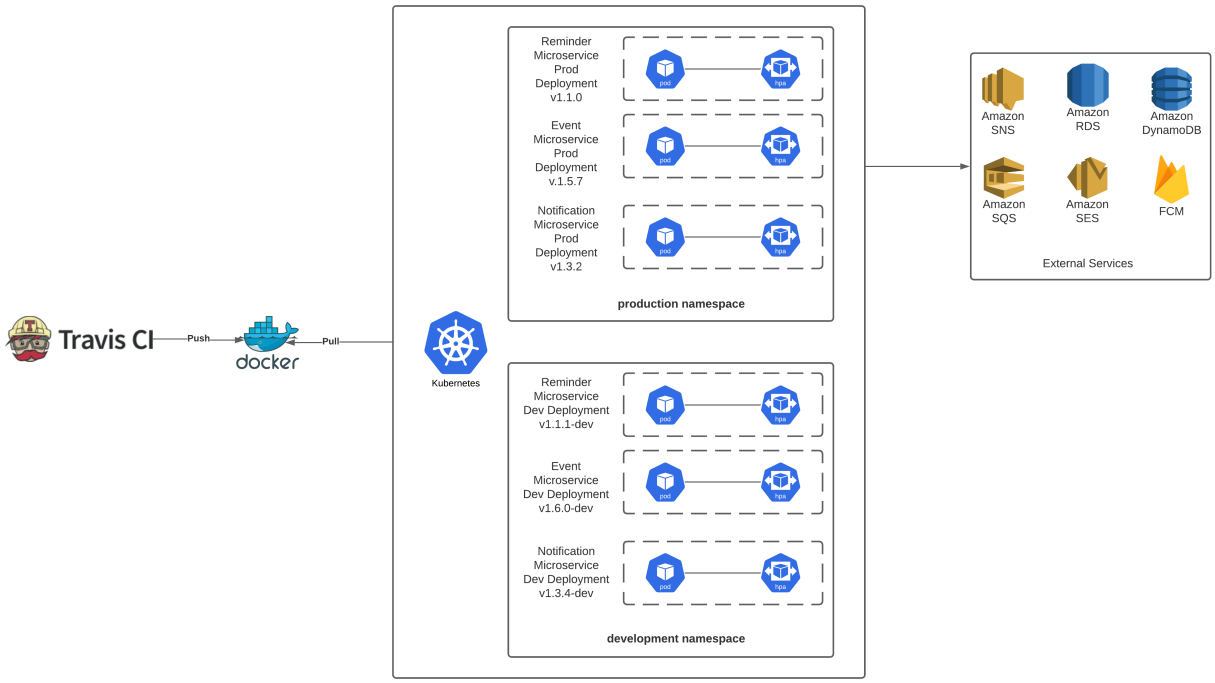
Any update in development environment will be reflected only in the development namespace and any update in production development environment will be reflected only in the production namespace. Updates may include but not limited to the following - 1. Scaling number of pods 2. Updating deployment with newer version of docker image 3. Updaing environment variables and secrets

After the build is complete Ansible updates the Kubernetes deployments by telling it to pull the updated docker-images for the microservices from DockerHub and create/update various resources as defined in the configuration files.

CHAPTER

SIX

DEPLOYMENT ARCHITECTURE



6.1 Deployment Environments

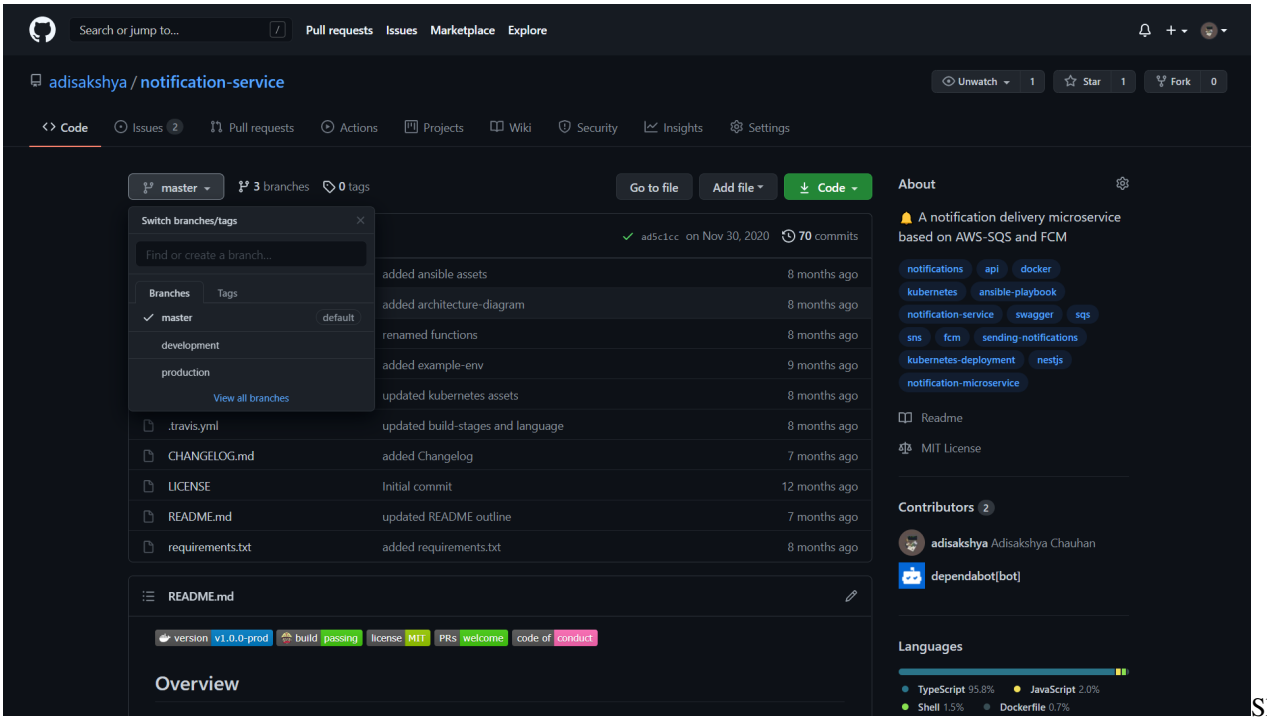
The notification-delivery system is deployed on Kubernetes in multiple environments and maintained on GitHub. The table below describes a list of possible tiers -

Local - Developer's desktop/workstation where changes are worked on and tried out. **Development (dev)** - Development servers acting as a sandbox where unit testing may be performed by the developer. The environment may contain development tools, different or additional versions of libraries and support software, etc., which are not present in the production environment. **Staging** - Mirror of production environment. **Production** - The environment that users directly interact with.

For this projects the microservices are deployed in 2 environments namely -

1. Development (or dev) Environment
2. Production Environment

At the source code level these environments are distinguished using Git branches. The `development` and `production` git branchs of the respective repositories features the source code for the Development Environment and the Production Environment respectively.



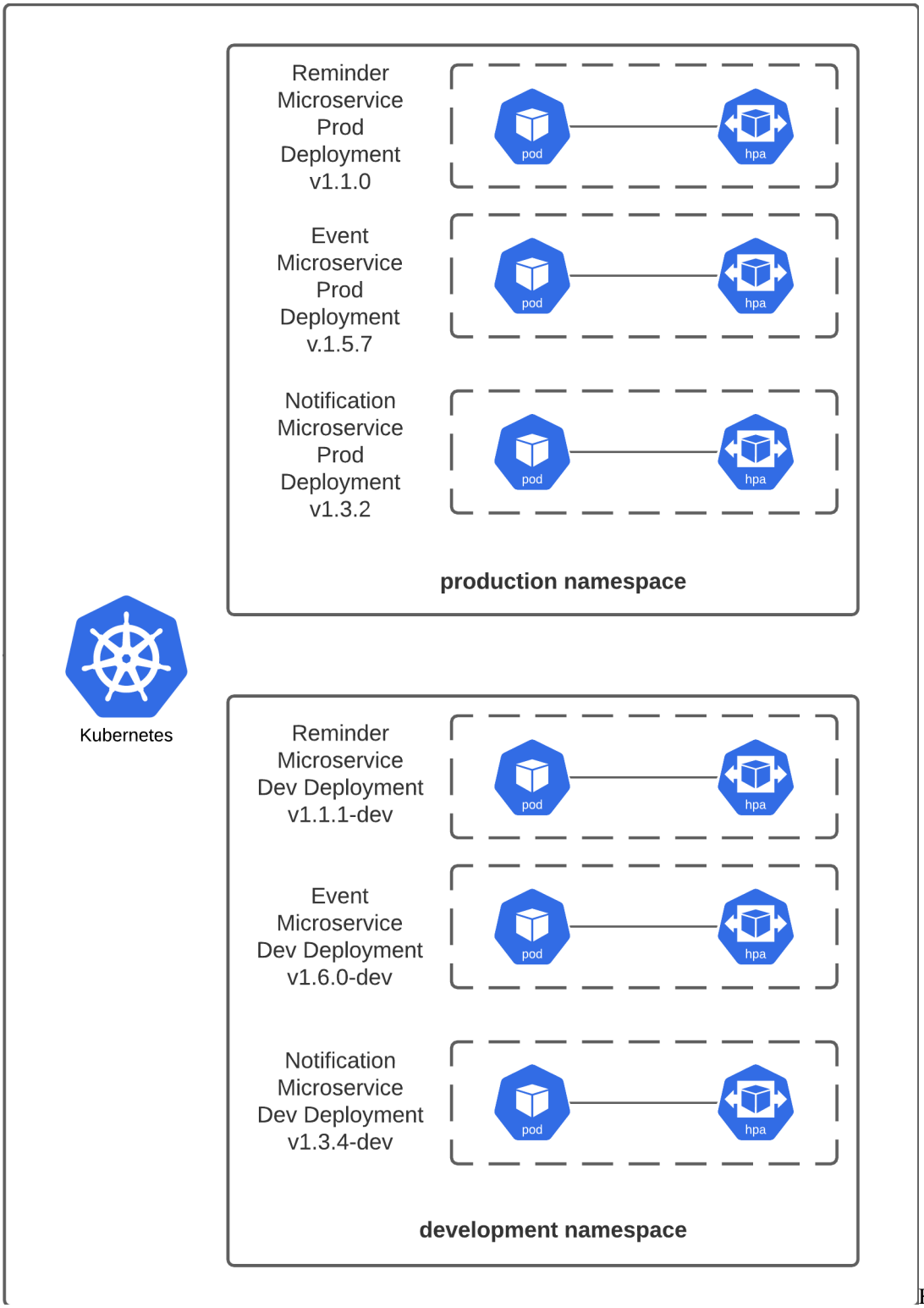
Showing all branches in the Notification Service git repository

The development branch is created from the master branch. An appropriate branch can be created from the development branch for any feature updates, patches etc and then it can be merged back to the development branch. Say when a feature currently in the development branch is complete to be shiped then the development branch can be merged to the master branch. The master branch reflects the code for the staging envionment through which all kinds of updates have to pass through. The production branch reflects the deployed code. A newer version of a microservice can be deployed by merging the master branch into the production branch. If you need to know what code is in production, you can check out the production branch.

6.2 Kubernetes Namespaces

> Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces and provide logical separation between the teams and their environments.

At the deployment level, i.e., on Kubernetes the deployment environments are distinguished using namespaces. Both development and production environments have their own namespaces, which logically separates them and allows modification in one without affecting the other environment. Using namespaces as environments makes it easier to manage workload for a project from a single Kubernetes cluster and this avoids the pain of maintaining multiple Kubernetes clusters for multiple environments.



Dashboard showing production and development namespaces

The `production` namespace wraps the production workload i.e., the source-code corresponding to the production-version/branch of each microservice is deployed in this namespace. The `development` namespace wraps the development workload i.e., the source-code corresponding to the development-version/branch of each microservice is deployed in this namespace.

Each environment has a unique identifier prefix asssigned to it, for production environment it is `prod` and for development environment it is `dev`. The unique prefix is used while naming the deployments, replica-sets, pods and services in the environment. For example the Notification Service is named as `prod-notification-service` and `dev-notification-service` when deployed to production environmentand development environment respectively. These prefix are also used while tagging the docker-images for the microservices to separate production and development packages.

Each environment has it own set of configurations, deployments and services running and independently interacting with various external services like AWS SNS, SQS, RDS, Firebase Cloud Messaging without interferring with the other environment.

The namespaces are created using the following YAML configuration -

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development

---

apiVersion: v1
kind: Namespace
metadata:
  name: production
  labels:
    name: production
```

6.3 Docker Images

To deploy any application to Kubernetes it is needed to be wraped as a container, this can be done by building docker-images. Each microservice is packaged as a docker image stored in a image repository provided by Dockerhub.

As each microservice runs in 2 environments namely `development` and `production` which are inturn two different namespaces in the Kubernetes Cluster, at the source code level - production and development branches are used to diffrentiate them. Each microservice is packed as a docker-image which has 2 tags - one that represents the production package and the second that represents the development package.

- Docker Image for Reminder Service - adisakshya/reminder-service
- Docker Image for Event Service - adisakshya/event-service
- Docker Image for Noification Service - adisakshya/notification-service

The tags for the docker images are named following the format - `{ VERSION }-{ ENVIRONMENT_PREFIX }`. Here `VERSION` is the version of the microservice that has been packed in that docker-image and `ENVIRONMENT_PREFIX` is the short name for a deployment environment. For example the name of docker image corresponding to the Event Service is `adisakshya/event-service` and it has the following tags -

- 1.0.0-dev
- 1.0.0-prod

The production environment uses the docker-image tagged with `prod` and the development environment uses the docker-image tagged with `dev` i.e., in case of event-service `adisakshya/event-service:1.0.0-prod` will be the

docker-image used for deploying the event-service to production environment and `adisakshya/event-service:1.0.0-dev` will be used to deploy it to the development environment.

6.4 Configurations and Environment Variables

There were a lot of reasons to use environment variables in this type of project the most important one's were to enable easier switching between multiple deployment environments and including configurations for external services.

There are multiple ways to define environment variables with Kubernetes -

Using configuration file

Environment variables can be included using the `env` field in the YAML configuration file. Environment variables that are set following this way override any environment variables specified in the container image.

Using secrets

It allows us to inject multiple values at once from a file. They're more suited to sensitive data like passwords, API keys, etc. For example, database passwords, private SSH keys, and certificates should all go into secrets. Reminder and Notification microservices uses Amazon RDS PostgreSQL Database and thus require appropriate credentials to use the database, the Notification microservice also requires Fiebase Cloud Messaging Configuration JSON to be able to function.

Following is an example of YAML configuration that can be used to set secrets -

```
apiVersion: v1
kind: Secret
metadata:
  name: prod-ses-secrets
  namespace: production
  resourceVersion: "1"
type: Opaque
data:
  # Name of reminder-email-template
  reminder_template: UHJvZHVjdGlvblJlbWluZGVyVGVtcGxhdGU=

---

apiVersion: v1
kind: Secret
metadata:
  name: dev-ses-secrets
  namespace: development
  resourceVersion: "1"
type: Opaque
data:
  # Name of reminder-email-template
  reminder_template: RGV2ZWxvcG1lbnRSZW1pbmRlclRlbXBsYXRl
```

Following is an example of YAML configuration that is used to create a deployment while utilizing secrets. Here `env` is the name of deployment environment, `env_prefix` is either 'dev' or 'prod', `resource_name` is the name of microservice for example - 'event-service', `docker_image_name` and `docker_image_tag` represent the name and tag of the docker image associated with the `resource_name`.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: '{{ env_prefix }}-{{ resource_name }}'
  namespace: '{{ env }}'
  labels:
    app: '{{ env_prefix }}-{{ resource_name }}'
spec:
  selector:
    matchLabels:
      app: '{{ env_prefix }}-{{ resource_name }}'
  replicas: 1
  template:
    metadata:
      labels:
        app: '{{ env_prefix }}-{{ resource_name }}'
    spec:
      containers:
        - name: '{{ env_prefix }}-{{ resource_name }}'
          image: '{{ docker_image_name }}:{{ docker_image_tag }}'
          imagePullPolicy: Always
          ports:
            - containerPort: 80
          env:
            - name: NODE_ENV
              value: '{{ env }}'
            - name: DB_HOST
              valueFrom:
                secretKeyRef:
                  name: '{{ env_prefix }}-reminder-db-secrets'
                  key: host
            - name: DB_NAME
              valueFrom:
                secretKeyRef:
                  name: '{{ env_prefix }}-reminder-db-secrets'
                  key: name
            - name: DB_PASS
              valueFrom:
                secretKeyRef:
                  name: '{{ env_prefix }}-reminder-db-secrets'
                  key: password
            - name: DB_USER
              valueFrom:
                secretKeyRef:
                  name: '{{ env_prefix }}-reminder-db-secrets'
                  key: user
            - name: EVENT_TOPIC_ARN
              valueFrom:
                secretKeyRef:
                  name: '{{ env_prefix }}-sns-topics-secrets'
                  key: event_topic
```