Birla Institute of Technology and Science, Pilani

INTRODUCTION TO DEVOPS

Assignment – Part B (Post Mid Sem)

Group No - 3

Topic: DevOps Project

No.	Team Members	Roll No
1	Nitin Kumar	2020HS70003
2		
3		
4		
5		

Table of Contents

Post Mid Sem

Docker – Containerization

Docker

It is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels

Containerization

It is a form of virtualization where applications run in isolated user spaces, called containers, while using the same shared operating system (OS). A container is essentially a fully packaged and portable computing environment.

Everything an application needs to run—its binaries, libraries, configuration files, and dependencies—is encapsulated and isolated in its container. The container itself is abstracted away from the host OS, with only limited access to underlying resources—much like a lightweight virtual machine (VM). As a result, the containerized application can be run on various types of infrastructure—on bare metal, within VMs, and in the cloud—without needing to refactor it for each environment.

With containerization, there's less overhead during start-up and no need to set up a separate guest OS for each application since they all share the same OS kernel. Because of this high efficiency, containerization is commonly used for packaging up the many individual microservices that make up modern apps.

Docker Image

A Docker image is a file used to execute code in a Docker container. Docker images act as a set of instructions to build a Docker container, like a template. Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments.

Docker images have multiple layers, each one originates from the previous layer but is different from it. The layers speed up Docker builds while increasing reusability and decreasing disk use. Image layers are also read-only files. Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make changes.

Use Cases

A Docker image has everything needed to run a containerized application, including code, config files, environment variables, libraries and runtimes. When the image is deployed to a Docker environment, it can be executed as a Docker container. The docker run command creates a container from a specific image.
Docker images are a reusable asset -- deployable on any host. Developers can take the static image layers from one project and use them in another. This saves the user time, because they do not have to recreate an image from scratch.

Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

Microservices Dockerfile:

1. Cloud-Gateway:

   Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/cloud-gateway/Dockerfile

   ```
   FROM openjdk:11
   ```

```
EXPOSE 8080
ADD target/cloud-gateway-0.0.1-SNAPSHOT.jar cloud-gateway-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java","-jar","/cloud-gateway-0.0.1-SNAPSHOT.jar"]
```

2. Order-Service:
   Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/order-service/Dockerfile

```
FROM openjdk:11
EXPOSE 8081
ADD target/order-service-0.0.1-SNAPSHOT.jar order-service-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java","-jar","/order-service-0.0.1-SNAPSHOT.jar"]
```

3. Payment-Service:
   Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/payment-service/Dockerfile

```
FROM openjdk:11
EXPOSE 8082
ADD target/payment-service-0.0.1-SNAPSHOT.jar payment-service-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java","-jar","/payment-service-0.0.1-SNAPSHOT.jar"]
```

4. Product-Service:
   Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/product-service/Dockerfile

```
FROM openjdk:11
EXPOSE 8083
ADD target/product-service-0.0.1-SNAPSHOT.jar product-service-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java","-jar","/product-service-0.0.1-SNAPSHOT.jar"]
```

5. Service-Registry:
   Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/service-registry/Dockerfile

```
FROM openjdk:11
EXPOSE 8761
ADD target/service-registry-0.0.1-SNAPSHOT.jar service-registry-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java","-jar","/service-registry-0.0.1-SNAPSHOT.jar"]
```

Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Using Compose is basically a three-step process:

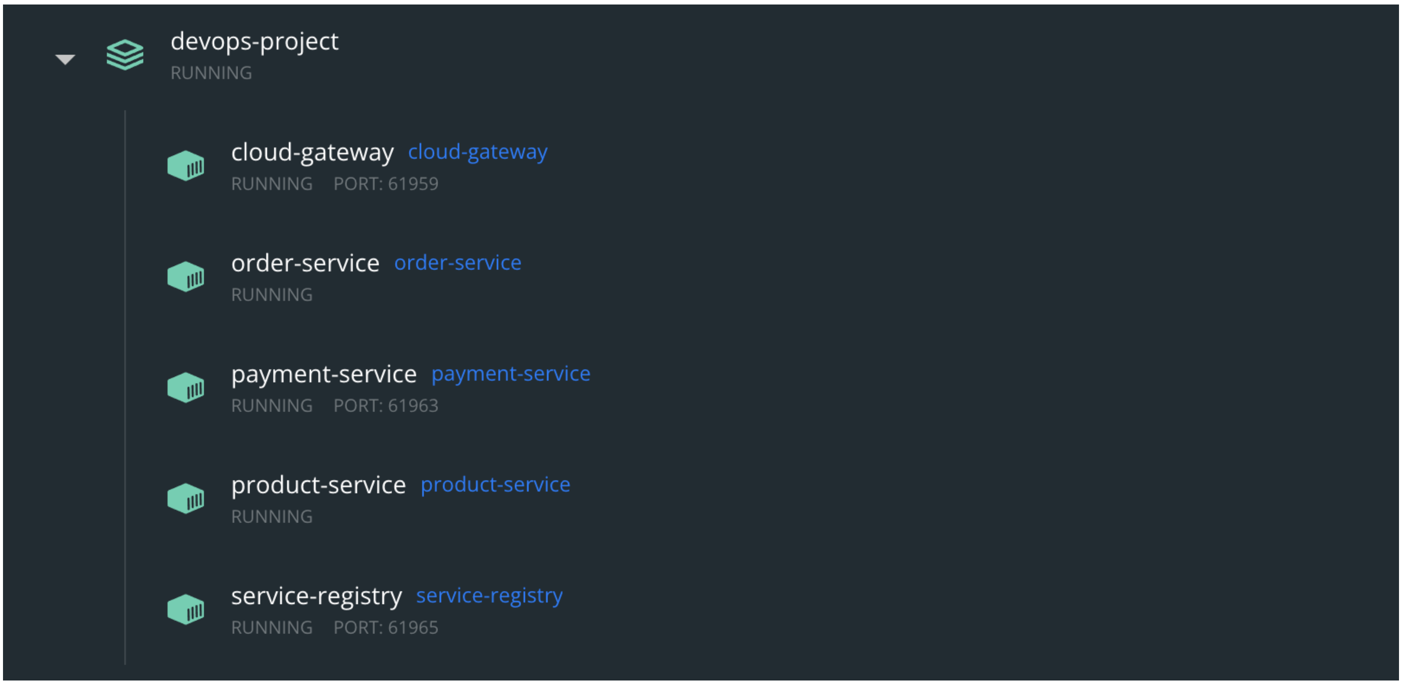
1. Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.

docker-compose.yml

Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/docker-compose.yml

```
version: '3'
services:
 cloud-gateway:
   container_name: cloud-gateway
   build:
     context: cloud-gateway
     dockerfile: Dockerfile
   image: "cloud-gateway"
   ports:
    - 8080
 order-service:
   container_name: order-service
   build:
     context: order-service
     dockerfile: Dockerfile
   image: "order-service"
   ports:
     - 8081
   depends_on:
     - cloud-gateway
 payment-service:
   container_name: payment-service
   build:
     context: payment-service
     dockerfile: Dockerfile
   image: "payment-service"
   ports:
     - 8082
   depends_on:
     - order-service
 product-service:
   container_name: product-service
   build:
     context: product-service
     dockerfile: Dockerfile
   image: "product-service"
   ports:
     - 8083
   depends_on:
     - payment-service
 service-registry:
   container_name: service-registry
   build:
     context: service-registry
     dockerfile: Dockerfile
   image: "service-registry"
   ports:
     - 8761
   depends_on:
     - product-service
```

Running containers using Docker-Compose



devops-project
RUNNING

cloud-gateway cloud-gateway
RUNNING PORT: 61959

order-service order-service
RUNNING

payment-service payment-service
RUNNING PORT: 61963

product-service product-service
RUNNING

service-registry service-registry
RUNNING PORT: 61965

Continuous Integration/Continuous Deployment

Building CI/CD

For Continuous Integration and Continuous Deploy we used Jenkins.

Jenkins:

It is an open source automation server. Jenkins helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat.

Jenkins Pipeline:

It is a suite of plugins which supports implementing and integrating *continuous delivery pipelines* into Jenkins.

Why Pipeline?

Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. By modelling a series of related tasks, users can take advantage of the many features of Pipeline:

- **Code**: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
- **Durable**: Pipelines can survive both planned and unplanned restarts of the Jenkins controller.
- **Pausable**: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
- **Versatile**: Pipelines support complex real-world CD requirements, including the ability to fork/join, loop, and perform work in parallel.
- **Extensible**: The Pipeline plugin supports custom extensions to its DSL and multiple options for integration with other plugins.

Pipeline-as-code:

Pipeline provides an extensible set of tools for modelling simple-to-complex delivery pipelines "as code" via the Pipeline domain-specific language (DSL) syntax.

The definition of a Jenkins Pipeline is written into a text file (called a Jenkinsfile) which in turn can be committed to a project’s source control repository.

This is the foundation of "Pipeline-as-code"; treating the CD pipeline a part of the application to be versioned and reviewed like any other code.

Creating a Jenkinsfile and committing it to source control provides a number of immediate benefits:

- Automatically creates a Pipeline build process for all branches and pull requests.
- Code review/iteration on the Pipeline (along with the remaining source code).
- Audit trail for the Pipeline.
- Single source of truth for the Pipeline, which can be viewed and edited by multiple members of the project.

While the syntax for defining a Pipeline, either in the web UI or with a Jenkinsfile is the same, it is generally considered best practice to define the Pipeline in a Jenkinsfile and check that in to source control.
Declarative versus Scripted Pipeline syntax

A Jenkinsfile can be written using two types of syntax - Declarative and Scripted.

Declarative Pipelines:

It is a more recent feature of Jenkins Pipeline which:

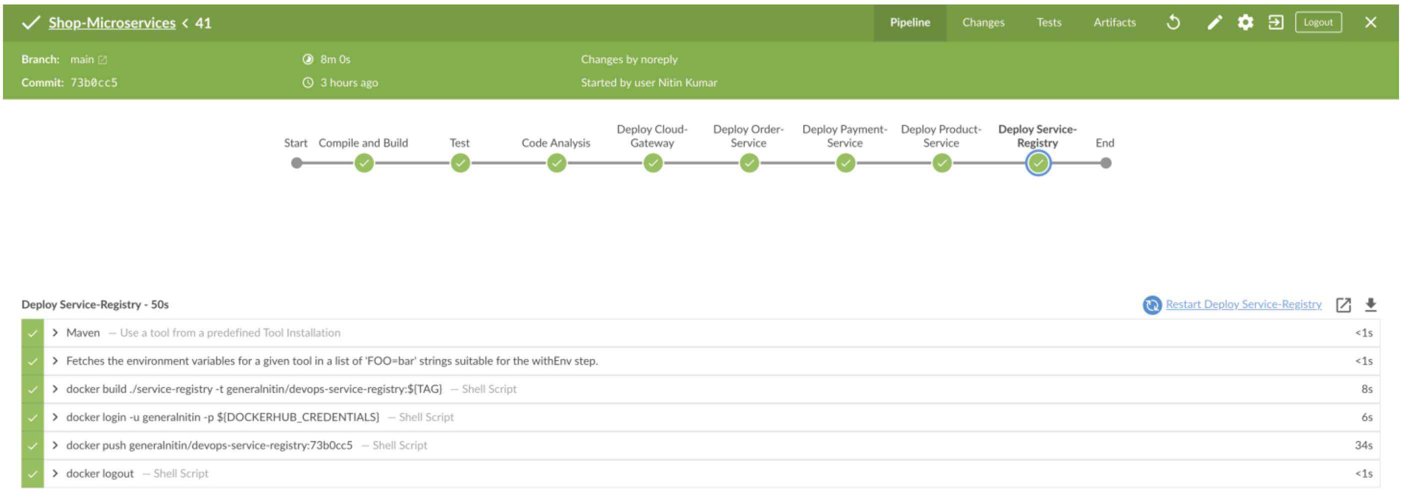
- provides richer syntactical features over Scripted Pipeline syntax, and
- is designed to make writing and reading Pipeline code easier.

Many of the individual syntactical components (or "steps") written into a Jenkinsfile, however, are common to both Declarative and Scripted Pipeline.

Link: https://github.com/2020HS-DevOps-Group3/Shop-Microservices/blob/main/Jenkinsfile

```
pipeline {
  agent any
  tools {
    maven 'Maven'
  }
  environment {
    DOCKERHUB_CREDENTIALS = credentials('generalnitin-dockerhub')
  }
  stages {
    stage("Compile and Build") {
      steps {
        echo 'Compile and Build the project...'
        sh "mvn clean install -DskipTests"
      }
    }
    stage("Test") {
      steps {
        echo 'Test the project...'
        sh "mvn test"
      }
    }
    stage("Code Analysis") {
      agent any
      steps {
        withSonarQubeEnv('SonarCloud') {
          echo 'Static code analysis with SonarQube...'
          sh 'mvn clean package -DskipTests sonar:sonar'
        }
      }
    }
    stage('Deploy Cloud-Gateway') {
      steps {
        sh 'docker build ./cloud-gateway -t generalnitin/devops-cloud-gateway:${GIT_COMMIT}'
        withCredentials([string(credentialsId: 'generalnitin-dockerhub', variable: 'docker_pwd')]) {
          sh "docker login -u generalnitin -p ${docker_pwd}"
        }
        sh "docker push generalnitin/devops-cloud-gateway:${GIT_COMMIT} "
      }
      post {
        always {
          sh 'docker logout'
        }
      }
    }
    stage('Deploy Order-Service') {
        steps {
            sh 'docker build ./order-service -t generalnitin/devops-order-service:${GIT_COMMIT}'
            withCredentials([string(credentialsId: 'generalnitin-dockerhub', variable: 'docker_pwd')]) {
                sh "docker login -u generalnitin -p ${docker_pwd}"
            }
            sh "docker push generalnitin/devops-order-service:${GIT_COMMIT}"
        }
        post {
            always {
                sh 'docker logout'
            }
        }
    }
    stage('Deploy Payment-Service') {
      steps {
        sh 'docker build ./payment-service -t generalnitin/devops-payment-service:${GIT_COMMIT}'
        withCredentials([string(credentialsId: 'generalnitin-dockerhub', variable: 'docker_pwd')]) {
          sh "docker login -u generalnitin -p ${docker_pwd}"
        }
        sh "docker push generalnitin/devops-payment-service:${GIT_COMMIT}"
      }
      post {
        always {
          sh 'docker logout'
        }
      }
    }
    stage('Deploy Product-Service') {
      steps {
        sh 'docker build ./product-service -t generalnitin/devops-product-service:${GIT_COMMIT}'
        withCredentials([string(credentialsId: 'generalnitin-dockerhub', variable: 'docker_pwd')]) {
          sh "docker login -u generalnitin -p ${docker_pwd}"
        }
        sh "docker push generalnitin/devops-product-service:${GIT_COMMIT}"
      }
      post {
        always {
          sh 'docker logout'
        }
      }
    }
    stage('Deploy Service-Registry') {
      steps {
        sh 'docker build ./service-registry -t generalnitin/devops-service-registry:${GIT_COMMIT}'
        withCredentials([string(credentialsId: 'generalnitin-dockerhub', variable: 'docker_pwd')]) {
          sh "docker login -u generalnitin -p ${docker_pwd}"
        }
        sh "docker push generalnitin/devops-service-registry:${GIT_COMMIT}"
      }
      post {
        always {
          sh 'docker logout'
        }
      }
    }
  }
}
```

Jenkins Build Output:



Branch: main

Commit: 73b0cc5

8m 0s

3 hours ago

Changes by noreply

Started by user Nitin Kumar

Deploy Service-Registry - 50s

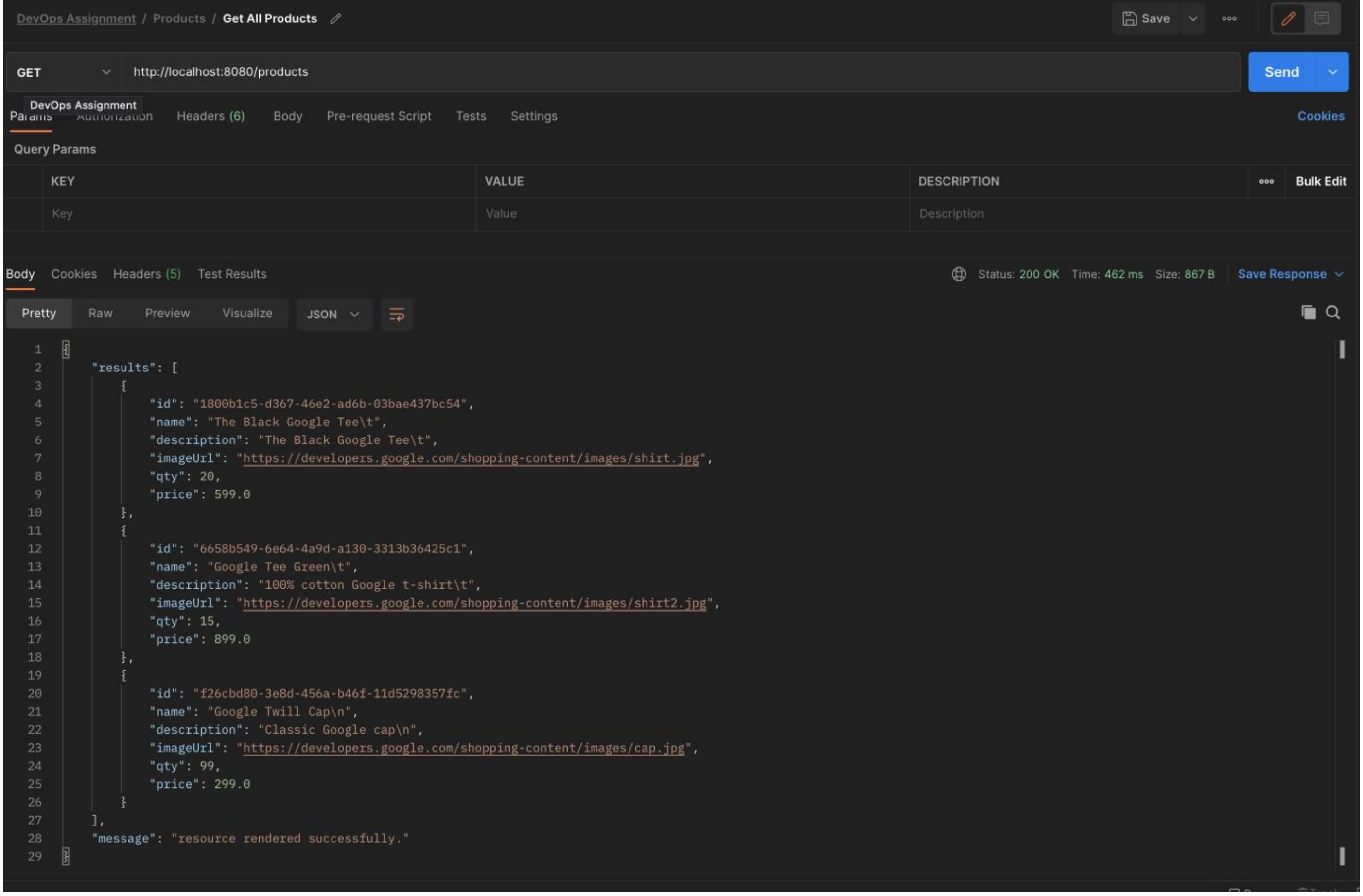
```
✓ > Maven — Use a tool from a predefined Tool Installation                                                   <1s
✓ > Fetches the environment variables for a given tool in a list of 'FOO=bar' strings suitable for the withEnv step.   <1s
✓ > docker build ./service-registry -t generalnitin/devops-service-registry:${TAG} — Shell Script             8s
✓ > docker login -u generalnitin -p ${DOCKERHUB_CREDENTIALS} — Shell Script                                   6s
✓ > docker push generalnitin/devops-service-registry:73b0cc5 — Shell Script                                   34s
✓ > docker logout — Shell Script                                                                              <1s
```

The Deployment stage could have been parallel, i.e. the deployment of service could have been concurrent but due to having a free tier account of Docker Hub multiple connections was not possible. Hence decided to do in series itself.

Calling REST APIs – Using Postman

List of products

Making a call through cloud-gateway using Postman



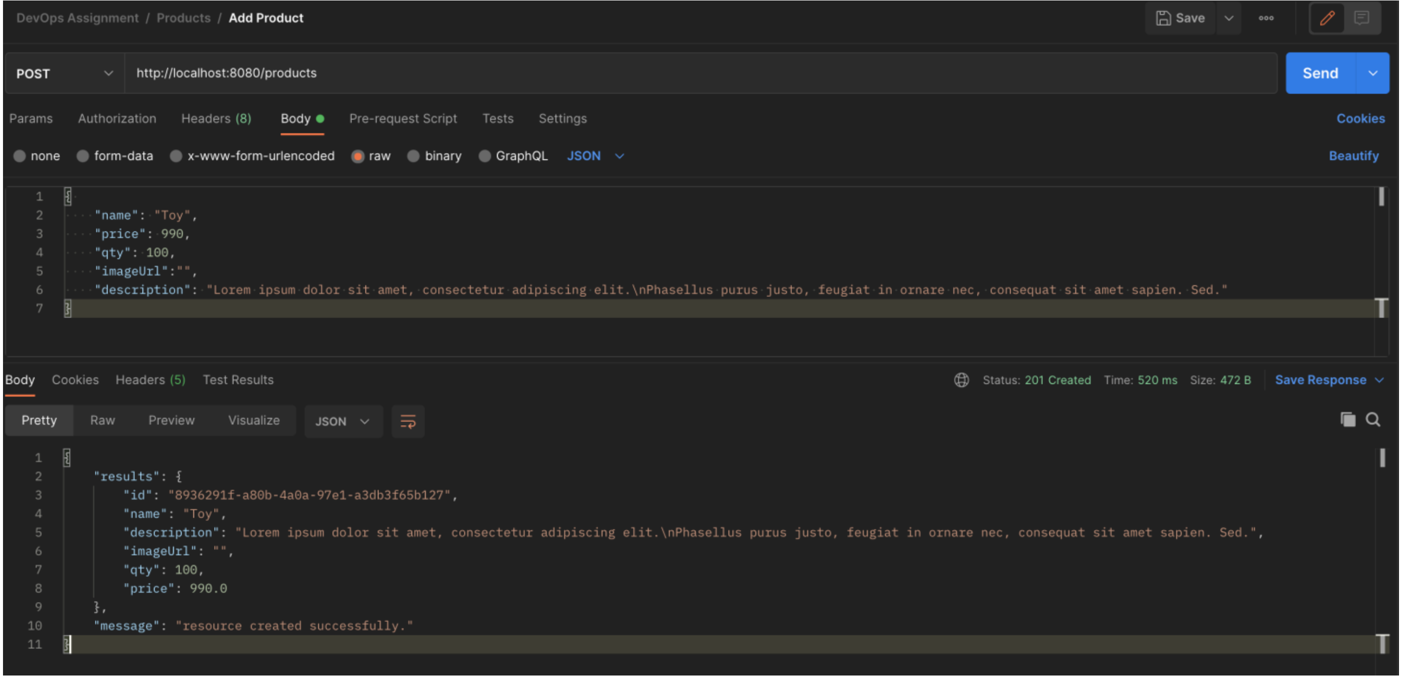
DevOps Assignment / Products / Get All Products

GET http://localhost:8080/products

Status: 200 OK Time: 462 ms Size: 867 B

```
{
    "results": [
        {
            "id": "1800b1c5-d367-46e2-ad6b-03bae437bc54",
            "name": "The Black Google Tee\t",
            "description": "The Black Google Tee\t",
            "imageUrl": "https://developers.google.com/shopping-content/images/shirt.jpg",
            "qty": 20,
            "price": 599.0
        },
        {
            "id": "6658b549-6e64-4a9d-a130-3313b36425c1",
            "name": "Google Tee Green\t",
            "description": "100% cotton Google t-shirt\t",
            "imageUrl": "https://developers.google.com/shopping-content/images/shirt2.jpg",
            "qty": 15,
            "price": 899.0
        },
        {
            "id": "f26cbd80-3e8d-456a-b46f-11d5298357fc",
            "name": "Google Twill Cap\n",
            "description": "Classic Google cap\n",
            "imageUrl": "https://developers.google.com/shopping-content/images/cap.jpg",
            "qty": 99,
            "price": 299.0
        }
    ],
    "message": "resource rendered successfully."
}
```

Adding a product



DevOps Assignment / Products / Add Product

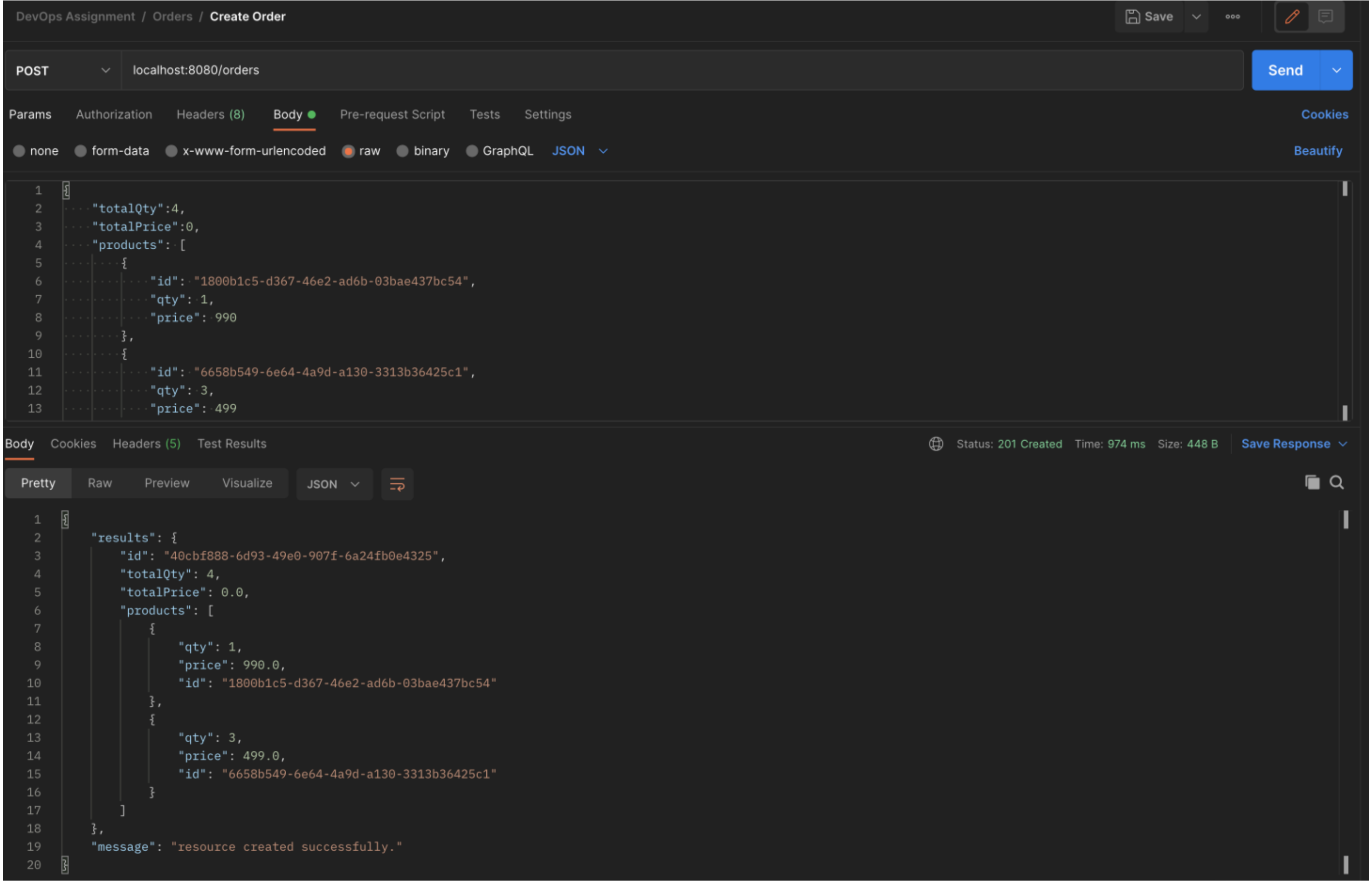
POST http://localhost:8080/products

```
{
    "name": "Toy",
    "price": 990,
    "qty": 100,
    "imageUrl":"",
    "description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\nPhasellus purus justo, feugiat in ornare nec, consequat sit amet sapien. Sed."
}
```

Status: 201 Created Time: 520 ms Size: 472 B

```
{
    "results": {
        "id": "8936291f-a80b-4a0a-97e1-a3db3f65b127",
        "name": "Toy",
        "description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\nPhasellus purus justo, feugiat in ornare nec, consequat sit amet sapien. Sed.",
        "imageUrl": "",
        "qty": 100,
        "price": 990.0
    },
    "message": "resource created successfully."
}
```

Creating an Order:



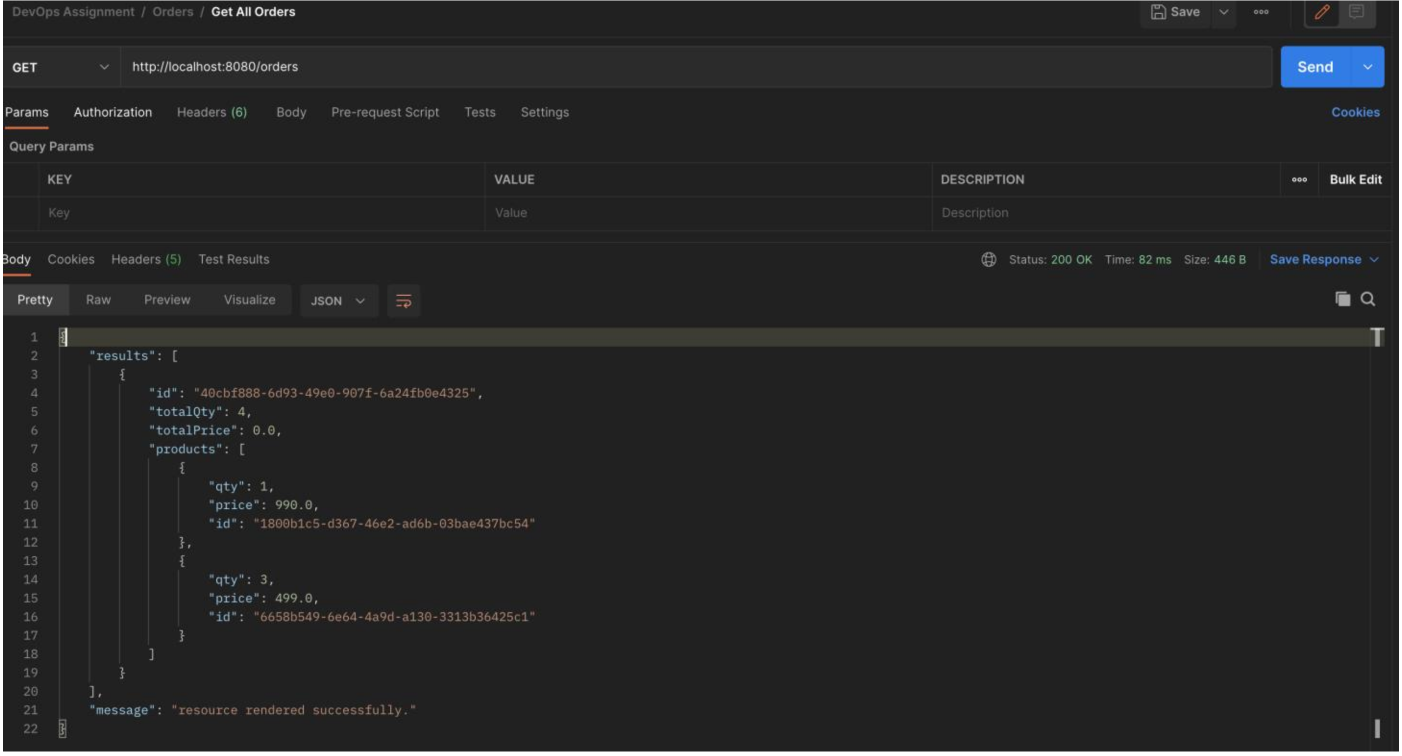
POST localhost:8080/orders

```
{
    "totalQty":4,
    "totalPrice":0,
    "products": [
        {
            "id": "1800b1c5-d367-46e2-ad6b-03bae437bc54",
            "qty": 1,
            "price": 990
        },
        {
            "id": "6658b549-6e64-4a9d-a130-3313b36425c1",
            "qty": 3,
            "price": 499
```

Status: 201 Created Time: 974 ms Size: 448 B

```
{
    "results": {
        "id": "40cbf888-6d93-49e0-907f-6a24fb0e4325",
        "totalQty": 4,
        "totalPrice": 0.0,
        "products": [
            {
                "qty": 1,
                "price": 990.0,
                "id": "1800b1c5-d367-46e2-ad6b-03bae437bc54"
            },
            {
                "qty": 3,
                "price": 499.0,
                "id": "6658b549-6e64-4a9d-a130-3313b36425c1"
            }
        ]
    },
    "message": "resource created successfully."
}
```

Fetching All Orders:



GET http://localhost:8080/orders

Status: 200 OK Time: 82 ms Size: 446 B

```
{
    "results": [
        {
            "id": "40cbf888-6d93-49e0-907f-6a24fb0e4325",
            "totalQty": 4,
            "totalPrice": 0.0,
            "products": [
                {
                    "qty": 1,
                    "price": 990.0,
                    "id": "1800b1c5-d367-46e2-ad6b-03bae437bc54"
                },
                {
                    "qty": 3,
                    "price": 499.0,
                    "id": "6658b549-6e64-4a9d-a130-3313b36425c1"
                }
            ]
        }
    ],
    "message": "resource rendered successfully."
}
```

DevOps Maturity in our project.

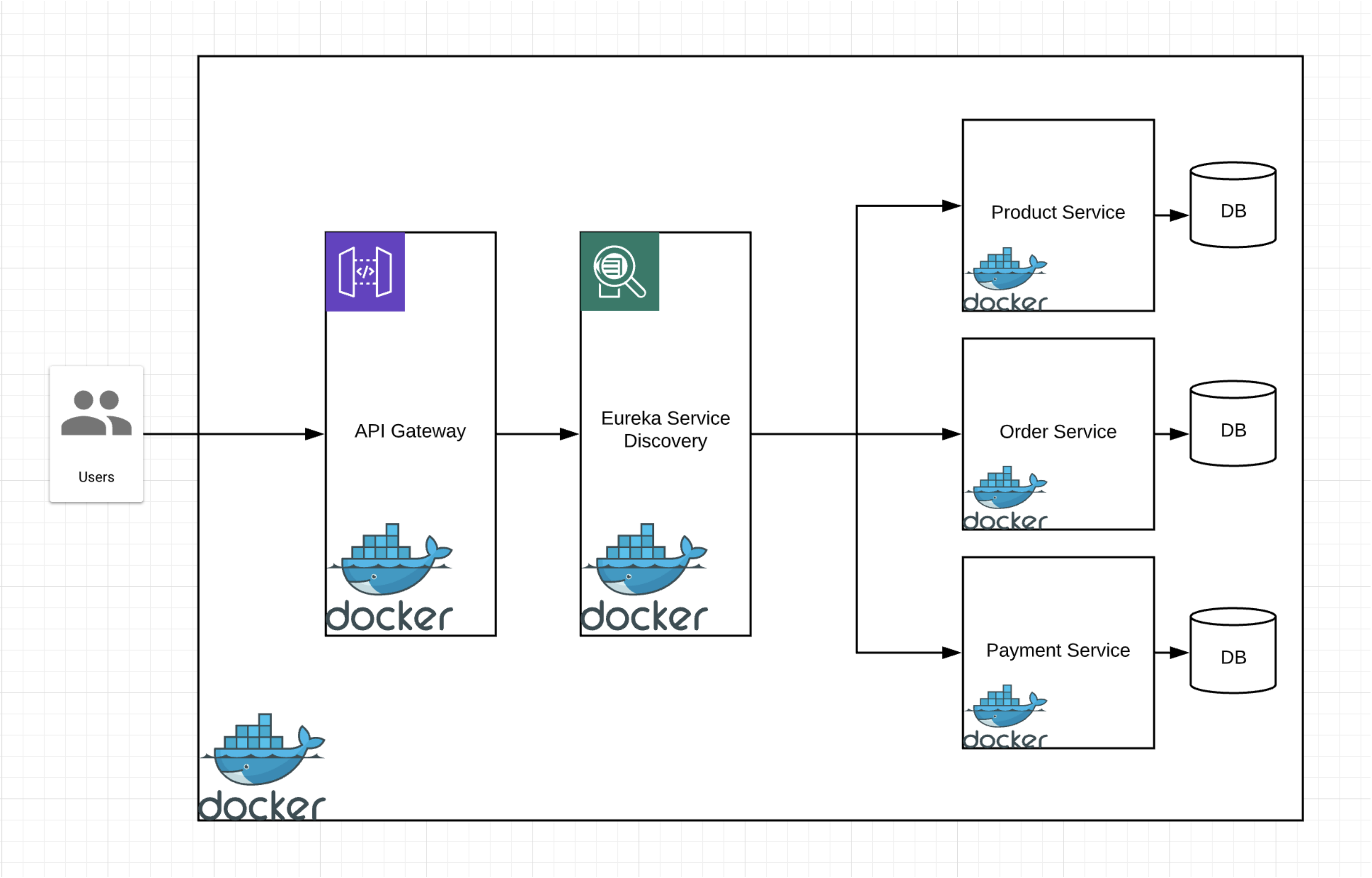
Describe Project:

The project is Shopping webapp with the frontend in Angular and the backend in Java Spring Boot.
At the backend we are using microservices architectures and there are 3 core services namely Product, Order and Payment along with 2 backing services, like a API Gateway and Eureka discovery client.

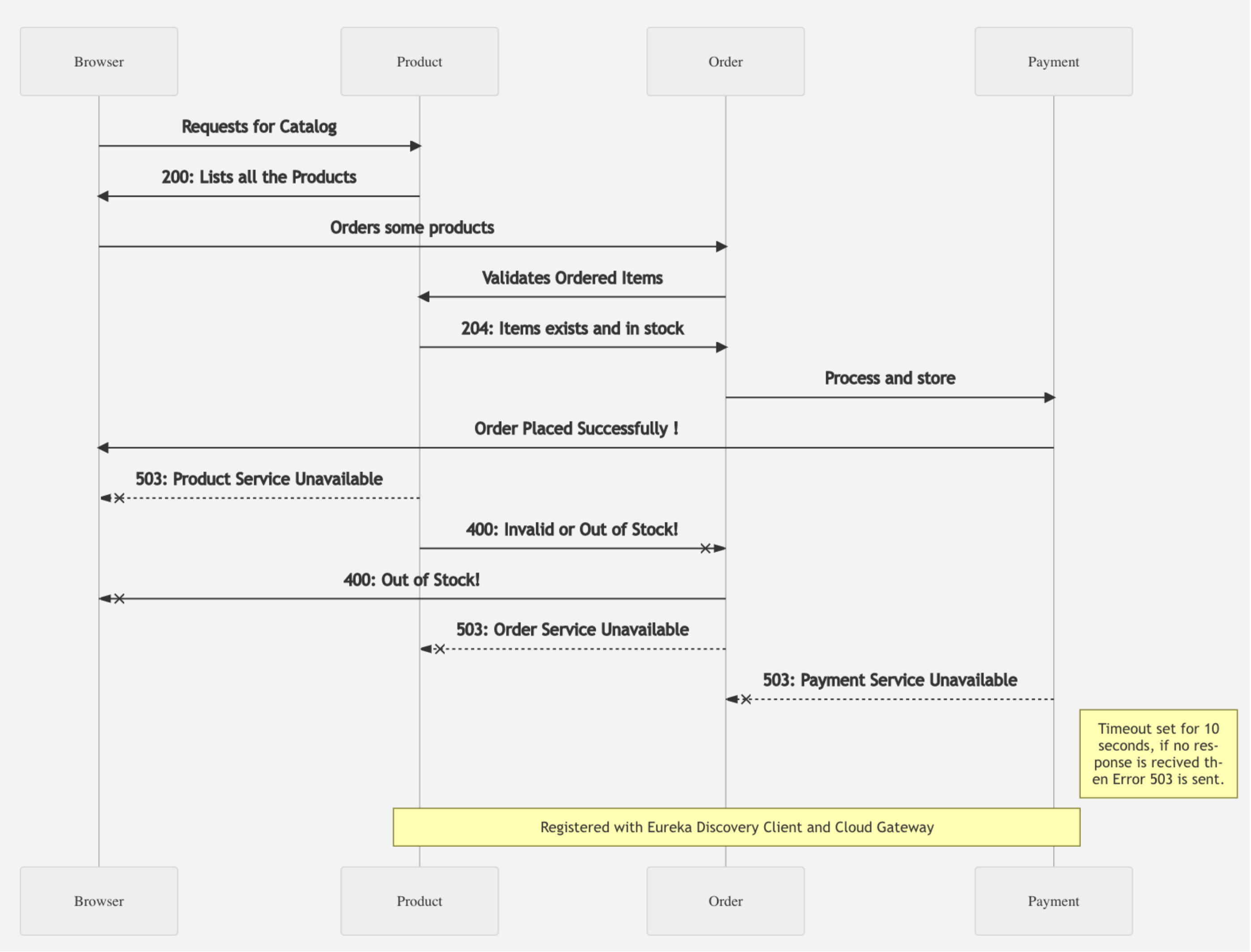
Project Type:

Microservice Webapp

Architecture:



Sequence Diagram:


Browser
Product
Order
Payment
Requests for Catalog
200: Lists all the Products
Orders some products
Validates Ordered Items
204: Items exists and in stock
Process and store
Order Placed Successfully !
503: Product Service Unavailable
400: Invalid or Out of Stock!
400: Out of Stock!
503: Order Service Unavailable
503: Payment Service Unavailable
Timeout set for 10 seconds, if no response is recived then Error 503 is sent.
Registered with Eureka Discovery Client and Cloud Gateway
Browser
Product
Order
Payment

Share Learnings:

Nitin Kumar:

1. Implementing a Microservices Application using Java Spring Boot
2. Making UI with Angular and connecting with a Backend Service.
3. Resolving CORS issue in a standard way.
4. Working with various tools and technologies such as:
    1. GitHub Organisation feature
    2. GitHub Actions
    3. SonarQube configuration and integration
    4. Writing Unit Test using JUnits
    5. Selenium: Writing test cases, POM design pattern, execution using TestNG
    6. Jenkins: Use and Purpose, Pipelines, Declarative Pipelines
    7. Amazon EC2: Setup and Configuration
    8. Docker, Docker-Compose
    9. Deploying image using Jenkins and integration.

Challenges:

1. Working with microservices architecture.
2. Deployment of the microservices.
3. Learning various tools such as Sonar, Jenkins, Docker, and IAAS – Amazon

Problem faced and resolved:

1. CORS: When we make request using a frontend client like in our case: an Angular App. We get this error, as the backend and server are running on different ports (in local context).
   In order to solve it we read about it online and implemented a configuration file which allows the request from a whitelisted client.

2. Communication between various services: In order resolve this we are using Eureka Discovery Client, using which we call the services and in a service in unavailable the request gets timed out and the flow continues. Here we used circuit breaker pattern provided by Eureka.3

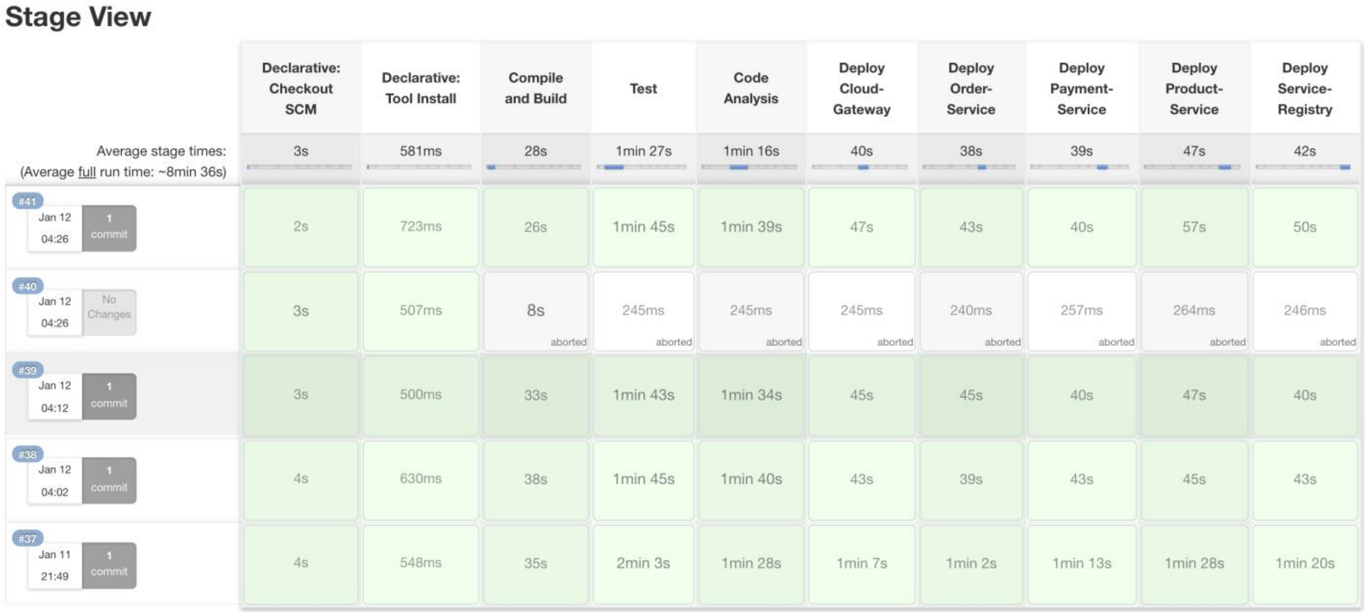
3. Dockerization: Dockerization was a little tricky, when working with various docker images. Upon reading we got to know about Docker-Compose and it is built for this purpose only.

DevOps:

How frequently you deploy:

With every single stable build we deploy the images of our services to docker hub.

Cycle time to build / Test / Deploy:



Stage View

	Declarative: Checkout SCM	Declarative: Tool Install	Compile and Build	Test	Code Analysis	Deploy Cloud-Gateway	Deploy Order-Service	Deploy Payment-Service	Deploy Product-Service	Deploy Service-Registry
Average stage times: (Average full run time: ~8min 36s)	3s	581ms	28s	1min 27s	1min 16s	40s	38s	39s	47s	42s
#41 Jan 12 04:26 1 commit	2s	723ms	26s	1min 45s	1min 39s	47s	43s	40s	57s	50s
#40 Jan 12 04:26 No Changes	3s	507ms	8s aborted	245ms aborted	245ms aborted	245ms aborted	240ms aborted	257ms aborted	264ms aborted	246ms aborted
#39 Jan 12 04:12 1 commit	3s	500ms	33s	1min 43s	1min 34s	45s	45s	40s	47s	40s
#38 Jan 12 04:02 1 commit	4s	630ms	38s	1min 45s	1min 40s	43s	39s	43s	45s	43s
#37 Jan 11 21:49 1 commit	4s	548ms	35s	2min 3s	1min 28s	1min 7s	1min 2s	1min 13s	1min 28s	1min 20s

Toolsets used for standardization:

1. GitHub
2. Maven
3. Selenium
4. Sonar
5. Jenkins
6. Docker