AEM Engineering Test

Requirements:

1. Create a webservice to accept a URI in the format: http://localhost:8080/romannumeral?query={integer}, where integer is in the range 1-255 and return the success response in the format: {“input” : “1”, “output” : “I”}.
2. Expand the range of numbers to 1-3999.
3. Include additional DevOps capabilities in your project to represent how you would prepare your project for ease of operation in a production environment (e.g. metrics, monitoring, logging, etc.).
4. Add tooling to build a runnable Docker container for your service if you are familiar with Docker.
5. Add support for range queries. In addition to the standard integer conversion query described above, you must also support a query of this format: http://localhost:8080/romannumeral?min={integer}&max={integer} and return the success response in the format: { “conversions”: [

    { “input” : “1”, “output” : “I” },
    { “input” : “2”, “output” : “II” },
    { “input” : “3”, “output” : “III” }] }

Tools used:

Spring Boot Framework
Actuator Endpoints
Swagger API documentation
Prometheus and Grafana for Metrics and Monitoring
Jaeger for Tracing
Elasticsearch-Fluentd-Kibana (EFK) for Logging

Setup:

I have developed all this code using a Macbook. I have installed the following softwares:

- Java version 17
- Docker software
- IntelliJ IDEA IDE
- Maven

Packaging layout:

-- src/main:

-- controller

exposes the REST APIs and calls the service layer.

-- service

contains the business logic to invoke each subpart and return a success/failure response to the controller.

-- entity

contains the logic to validate input and return roman output.

-- src/test:

-- controller

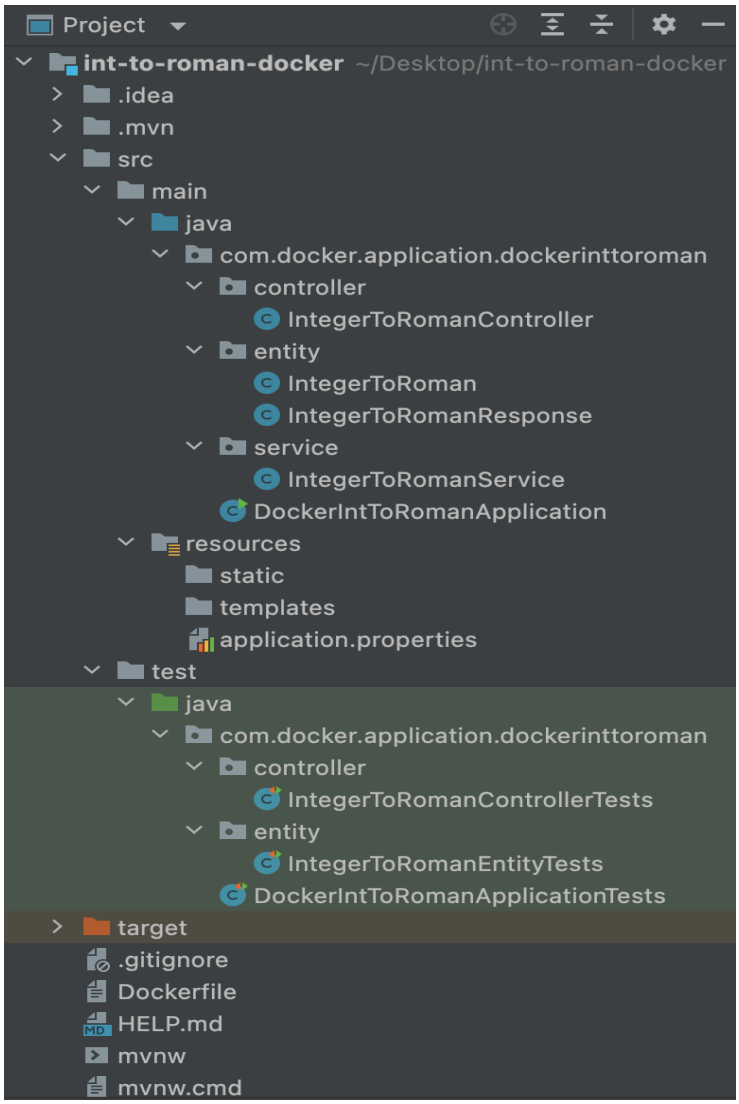
contains the JUNIT tests for testing REST endpoints.

-- entity

contains the JUNIT tests for validating the core functionality.

-- pom.xml contains the project dependencies.

-- Dockerfile contains the steps to build a docker image.



Code structure:

1. **src/main:**
   DockerIntToRomanApplication file contains the Spring Boot application's main method to run the webservice.

   1.1 controller/IntegerToRomanController:
   Exposes the REST APIs and calls the service layer. Also displays the response from the service layer as the final output.

   1.2 service/IntegerToRomanService:
   Contains the business logic to invoke each subpart and return a success/failure response to the controller. It has a convertToRoman method to validate input from the entity layer and return the conversion result to the controller.

   1.3 entity:
   Contains the actual logic to validate input and give roman output. IntegerToRoman file contains methods to validate input, parse the query input to integer, and convert to roman numeral. IntegerToRomanResponse file gives the appropriate success/failure response to the service layer.

2. **src/test:**
   DockerIntToRomanApplicationTests file contains Spring Boot application's test method contextLoads to run the JUnit test cases.

   2.1 controller/IntegerToRomanControllerTests:
   Contains the JUnit tests for testing REST endpoints. Details of test cases in the **"Tests"** section.

   2.2 entity/IntegerToRomanEntityTests:
   Contains the JUnit tests for validating the core functionality. Details of test cases in the "Tests" section.

Errors are returned in plain text format. More details in the "**Error handling**" section later in this report.

I have divided the assignment in 5 parts:
1. Integer to Roman Numeral conversion and Docker container
2. Metrics and Monitoring using Prometheus and Grafana
3. Tracing using Jaeger
4, Logging using Elasticsearch-Fluentd-Kibana (EFK) stack
5. Multithreaded Integer to Roman Numeral conversion and Docker container

Part 1: Integer to Roman Numeral conversion and Docker container

Steps to run:

1) git clone https://github.com/PurvaDeekshit/AEM-Engineering-Test.git
2) cd AEM-Engineering-Test/int-to-roman-docker
3) mvn clean install
4) docker build -f Dockerfile -t docker-int-to-roman .
5) Check image generated using the command docker images
6) docker run -p 8080:8080 docker-int-to-roman
7) check API documentation: http://localhost:8080/swagger-ui.html

Testing using browser/POSTMAN:

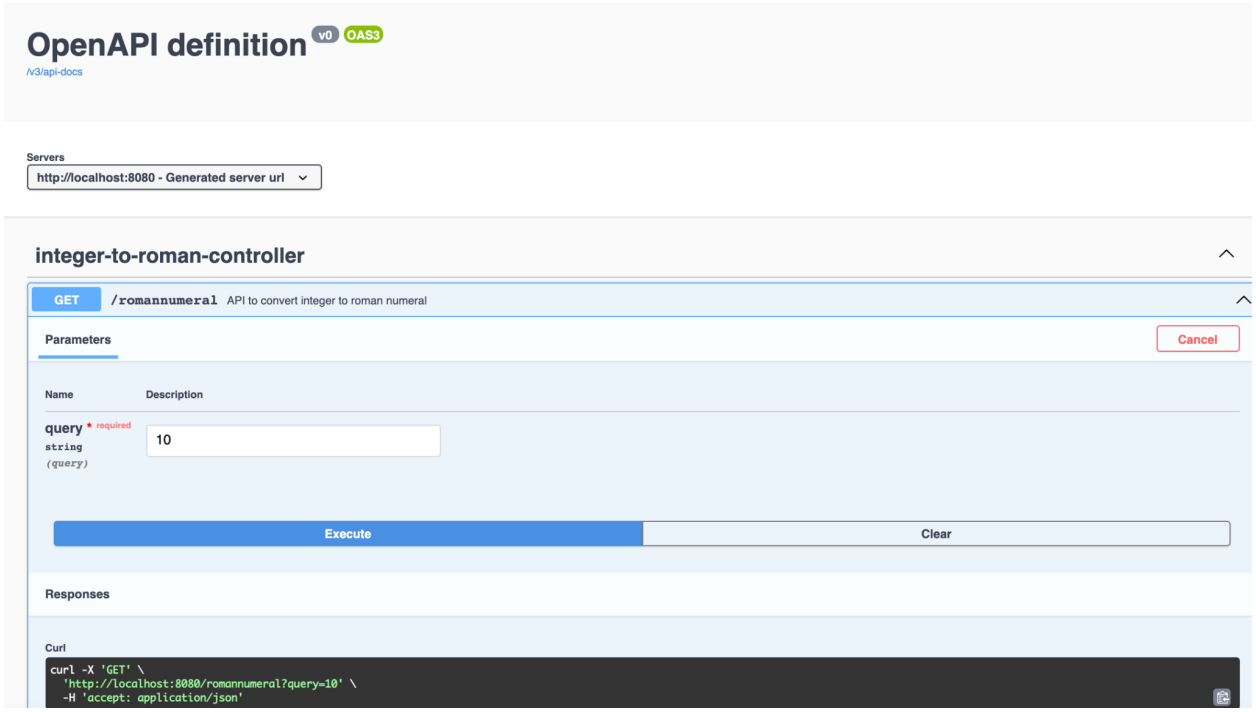
1) Test url : http://localhost:8080/romannumeral?query=10
   Output: {"input":10,"output":"X"}

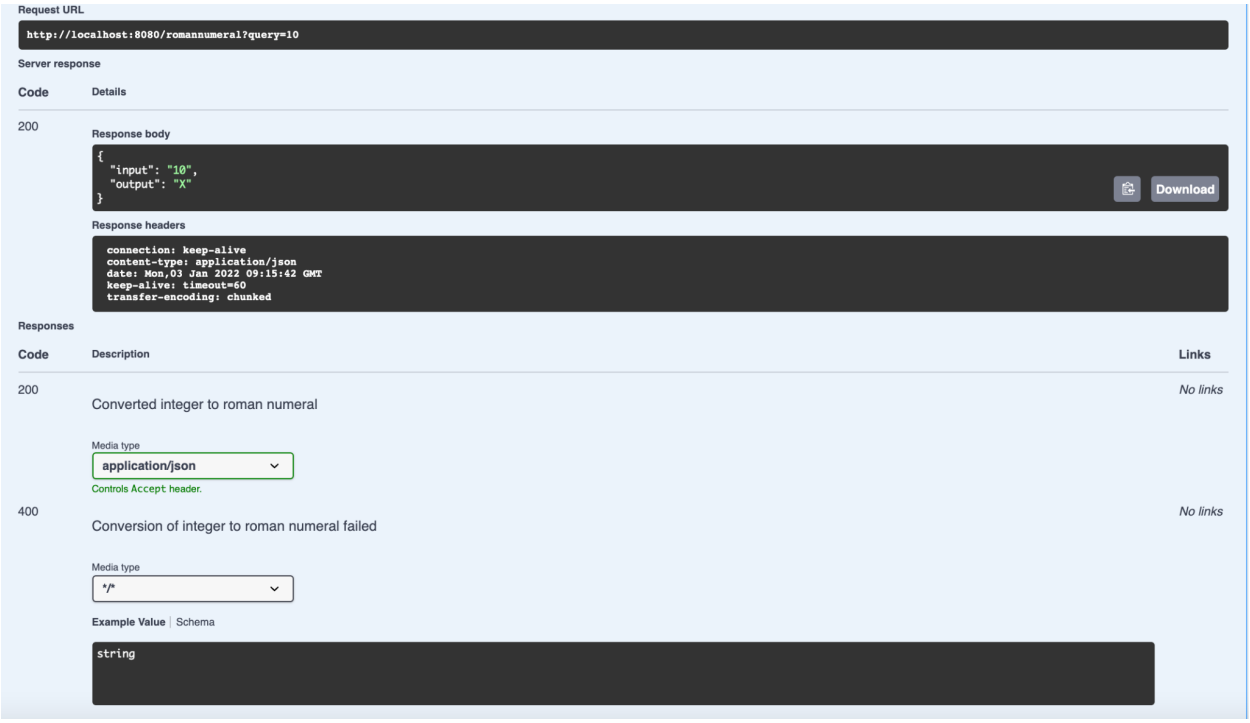
2) Test url : http://localhost:8080/romannumeral?query=0
   Output: Invalid Input Range

3) Test url : http://localhost:8080/romannumeral?query=abcd
   Output: Invalid input specified

4) Test url : http://localhost:8080/romannumeral?query=4000
   Output: Invalid Input Range

Sample output from Swagger OpenAPI documentation:





Part 2: Metrics and Monitoring using Prometheus and Grafana

1. Add the following dependencies in pom.xml file:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
  <scope>runtime</scope>
</dependency>
```

Steps to run:
1) git clone https://github.com/PurvaDeekshit/AEM-Engineering-Test.git
2) cd AEM-Engineering-Test/prometheus-grafana-integration
3) mvn clean install
4) docker-compose up

Test using browser:
1) Check the actuator endpoint:
http://localhost:8080/admin

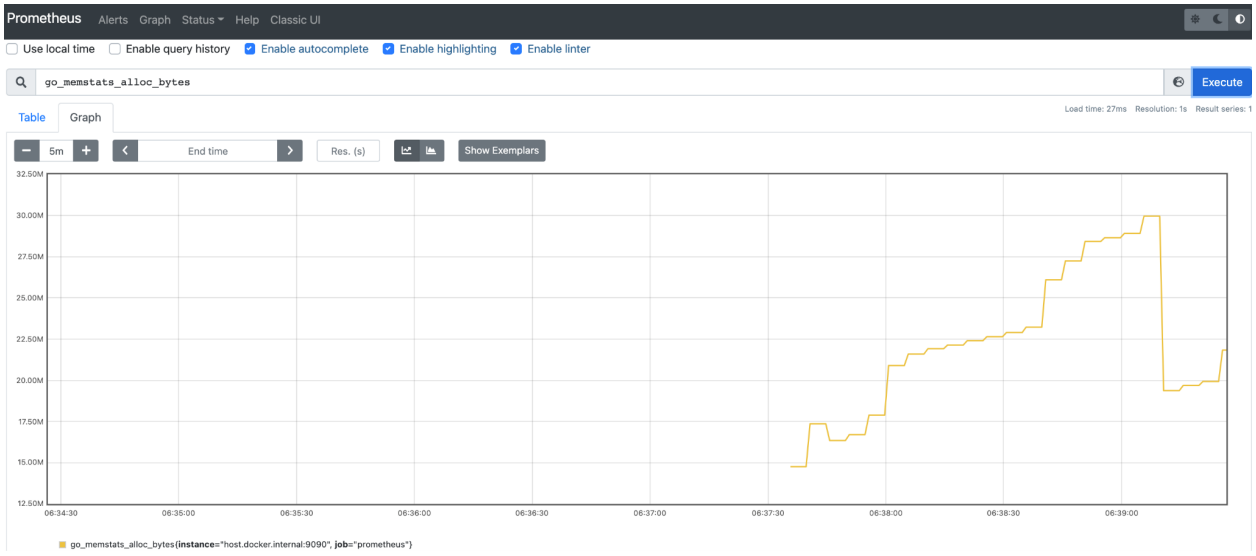
2) General actuator endpoints:
http://localhost:8080/admin/health

{"status":"UP","components":{"diskSpace":{"status":"UP","details":{"total":250685575168,"free":206251069440,"threshold":10485760,"exists":true}},"ping":{"status":"UP"}}}

3) Check the Prometheus actuator endpoint:
http://localhost:8080/admin/prometheus

4) Check Prometheus endpoint:
http://localhost:9090/graph?g0.expr=go_memstats_alloc_bytes&g0.tab=0&g0.stacked=0&g0.show_exemplars=0&g0.range_input=5m

Sample output from Prometheus:



5) Check Grafana endpoint:
http://localhost:3000/login

5.1) Give Username and Password as “admin”.

5.2) Click “Skip” on the “ Create New Password” page.

5.3) From the icon list on the left side of the “New Dashboard” page, click on Configuration -> Data sources -> Add data source -> Prometheus -> Select.

5.4) On the “Settings” page, in the HTTp section, paste the URL as: http://host.docker.internal:9090.

5.5) Click on “Save & test”.

5.6) From the icon list on the left side, click on Create -> Dashboard -> Add a new panel.

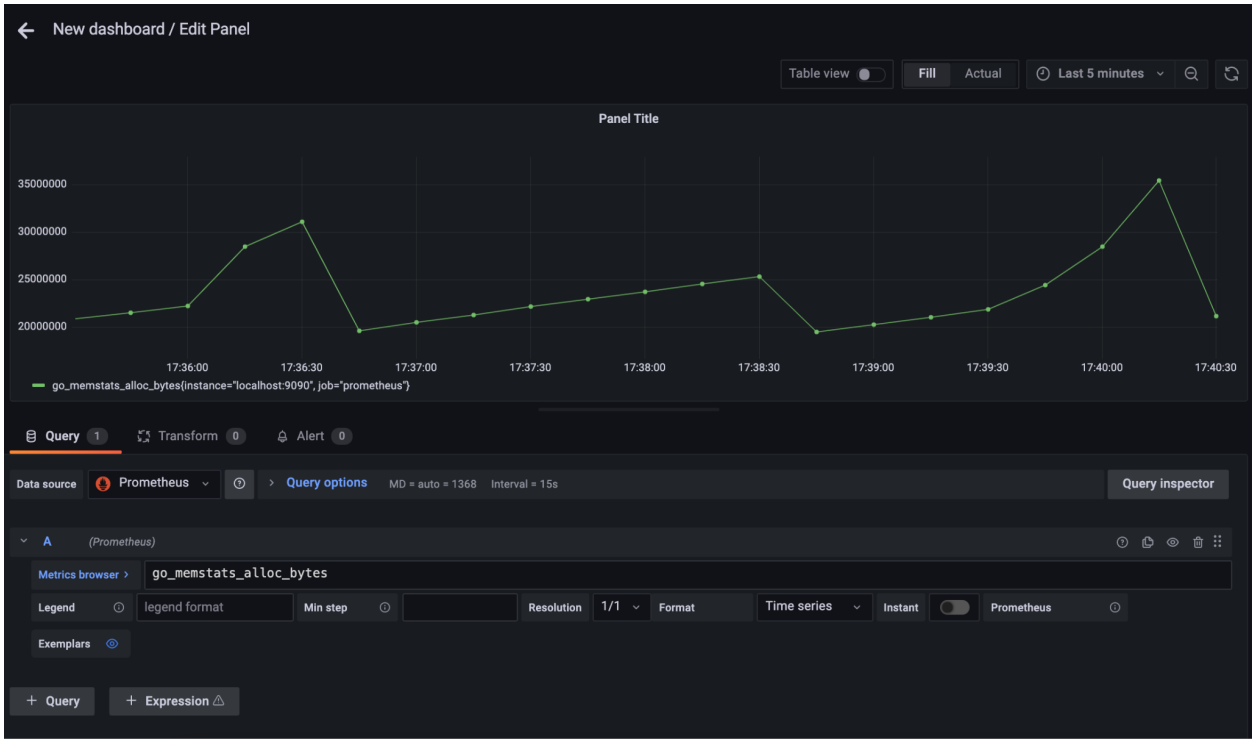
5.7) Add the required metrics for checking the logs on the web service by selecting the parameters from “Metrics browser”.

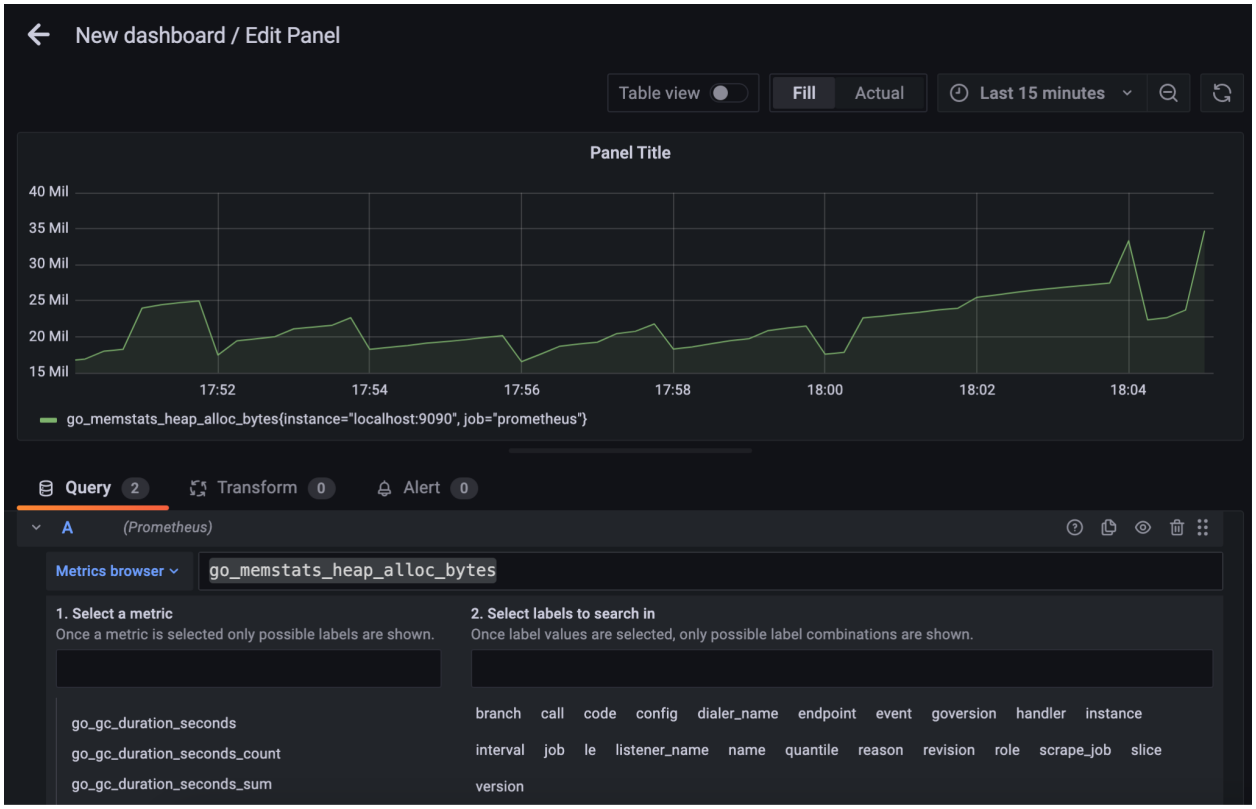
5.8) To add more parameters, click on “+Query”.

5.9) Click on “Apply” and the logs will be displayed on the dashboard, specifying the usage and requests that are generated on the web service.

6) docker-compose stop

Sample graphs from Grafana:





Part 3: Tracing using Jaeger

1. Add the following dependencies in pom.xml file:

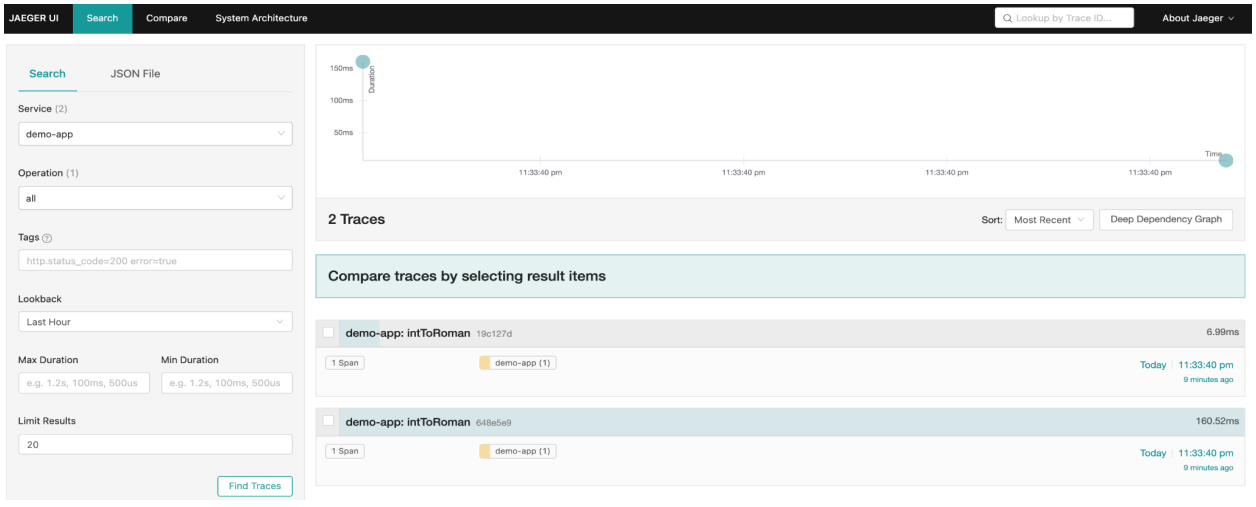
```
<dependency>
        <groupId>io.opentracing.contrib</groupId>
        <artifactId>opentracing-spring-jaeger-web-starter</artifactId>
        <version>3.3.1</version>
</dependency>
```

2. Create a JaegerConfig.java file, so that Jaeger can trace the incoming requests.

Steps to run:

1) mvn clean install
2) Run Jaeger in docker using the command:
docker run -d --name jaeger-ui -p 16686:16686 -p 6831:6831/udp jaegertracing/all-in-one
3) Run the application using the command:
java -jar target/docker-int-to-roman-0.0.1-SNAPSHOT.jar
4) Check the Jaeger URL: http://localhost:16686/
5) Under the "Service" section, click on "int-to-roman" and click on "Find Traces". Refresh the page if the service is not visible.
6) The tracing logs can be viewed in detail under the "Service & Operation" section.

Sample logs from Jaeger:



Part 4: Logging using Elasticsearch-Fluentd-Kibana (EFK) stack

Steps to run:

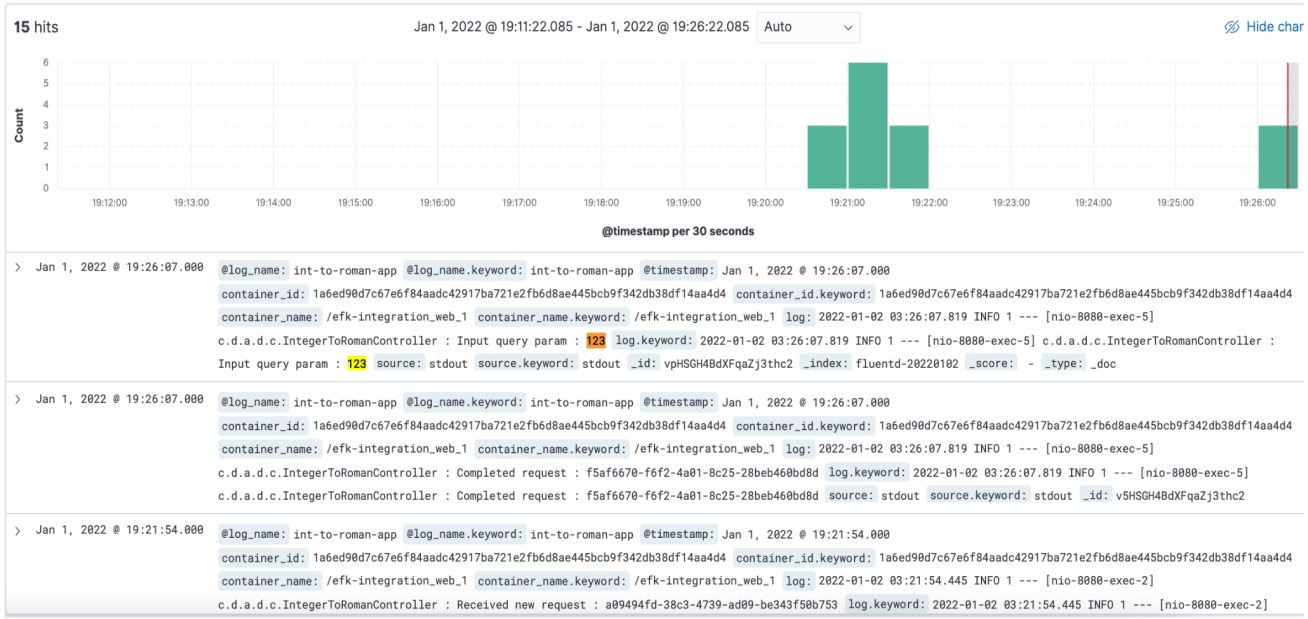
1) git clone https://github.com/PurvaDeekshit/AEM-Engineering-Test.git
2) cd AEM-Engineering-Test/efk-integration
3) mvn clean install
4) docker-compose up
5) After all docker containers are started successfully, test the application running on: http://localhost:8080/romannumeral?query=10
6) Check Elasticsearch running on: http://localhost:9200/

Expected output:

{
"name" : "a46aa09e30e3", "cluster_name" : "docker-cluster", "cluster_uuid" : "xnkMxG74RXqjcwnnAowfzg", "version" : { "number" : "7.13.1", "build_flavor" : "default", "build_type" : "docker", "build_hash" : "9a7758028e4ea59bcab41c12004603c5a7dd84a9", "build_date" : "2021-05-28T17:40:59.346932922Z", "build_snapshot" : false, "lucene_version" : "8.8.2", "minimum_wire_compatibility_version" : "6.8.0", "minimum_index_compatibility_version" : "6.0.0-beta1" }, "tagline" : "You Know, for Search"
}

7) Check Kibana running on: http://localhost:5601
8) On the Home page, search for "Index patterns" and click “Create index pattern” with “fluentd-*”.
9) Click “Next step” and select “Time field” as “@timestamp”. Click on “Create index pattern”.
10) On the top left corner, go to Analytics -> Discover section to check the logs generated for the webservice using: http://localhost:8080/romannumeral?query=123
11) docker-compose stop

Sample logs from Elasticsearch-Fluentd-Kibana (EFK):



Part 5: Multithreaded Integer to Roman Numeral conversion and Docker container

Steps to run:

1) git clone https://github.com/PurvaDeekshit/AEM-Engineering-Test.git
2) cd AEM-Engineering-Test/multithreaded-int-to-roman-docker
3) mvn clean install
4) docker build -f Dockerfile -t multithreaded-docker-int-to-roman .
5) Check image generated using the command docker images
6) docker run -p 8080:8080 multithreaded-docker-int-to-roman

Test using browser:

1) Test url : http://localhost:8080/romannumeral?min=1&max=3
   Output:

{"conversions":[{"input":"1","output":"I"},{"input":"2","output":"II"},{"input":"3","output":"III"}]}

2) Test url : http://localhost:8080/romannumeral?min=3&max=1
   Output:
   Min number is greater than or equal to Max number

3) Test url : http://localhost:8080/romannumeral?min=0&max=3999
   Output:
   Invalid Input Range

3) Test url : http://localhost:8080/romannumeral?min=1&max=abcd
   Output:
   Invalid input specified.

Dependency attribution:

Maven - project build and dependency management
Dependencies in pom.xml:

1. spring-boot-starter-web - Starter for building RESTful services
2. spring-boot-starter-test - Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito
3. springdoc-openapi-ui - For generation of API documentation
4. spring-boot-starter-actuator - Starter for using Spring Boot's Actuator for metrics and monitoring
5. micrometer-registry-prometheus - For Prometheus monitoring
6. opentracing-spring-jaeger-web-starter - For Jaeger distributed tracing

Engineering and Testing methodology:
I have used JAVA SpringBoot framework to develop REST endpoints and structured the code as controller, service and entity layers. Actuator endpoints are used to monitor and interact with the application and Maven is used for project build. I have followed a Test-Driven Development methodology and developed Junit test cases for service layer and REST controller.

Tests:
Performed testing using Postman and manual testing in the browser using the URL. Used JUnit Test for the following tests:

1. **Controller tests:**
   1.1 validInput - test covering positive scenario, where input is integer within the specified range.
   1.2 invalidInput - test covering a scenario, where input is not integer.
   1.3 outOfRangeInputBelowMinimumValue - test covering a scenario, where input is less than the specified range.
   1.4 outOfRangeInputAboveMaximumValue - test covering a scenario, where input is greater than the specified range.

2. **Functionality tests:**
   2.1 testValidInput - test if input parameter can be converted to integer.
   2.2 testInvalidInput - test if input parameter cannot be converted to integer.
   2.3 testValidInputRange - test if input is in the specified range.
   2.4 testInvalidInputRange - test if input is not in the specified range.
   2.5 testGetRoman - test if the input is converted to roman numeral correctly.

Error handling:
Following scenarios are handled in the service and functional layer code. These errors are returned in plain text format.

1. If input is not a positive integer - "Invalid input specified"
2. If input is within the specified range - "Invalid Input Range"
3. If input is null - "Null received while converting integer to roman"
4. If input is empty - "Empty string received while converting integer to roman"
5. If minimum number is not less than maximum number - "Min number is greater than or equal to Max number"

References:

1. Specification for Roman numerals: https://en.wikipedia.org/wiki/Roman_numerals
2. Spring Boot: https://start.spring.io/
3. Junit: https://spring.io/guides/gs/testing-web/
4. Maven: https://docs.spring.io/spring-android/docs/2.0.0.M3/reference/html/maven.html
5. OpenAPI documentation: https://springdoc.org/
6. Prometheus: https://prometheus.io/docs/prometheus/latest/getting_started/
7. Grafana: https://grafana.com/docs/grafana/latest/getting-started/
8. Fluentd: https://docs.fluentd.org/quickstart
9. Elasticsearch: https://www.elastic.co/guide/en/elastic-stack-get-started/current/index.html
10. Kibana: https://www.elastic.co/guide/en/kibana/current/index.html
11. Jaeger: https://www.jaegertracing.io/docs/1.29/getting-started/

Github source code link:

https://github.com/PurvaDeekshit/AEM-Engineering-Test

Project Contributor:

Purva Deekshit