**Problem statement**

XYZ wants to build an online movie ticket booking platform that caters to both B2B (theatre partners) and B2C (end customers) clients.

Key goals it wants accomplished as part of its solution:

* Enable theatre partners to onboard their theatres over this platform and get access to a bigger customer base while going digital.
* Enable end customers to browse the platform to get access to movies across different cities, languages, and genres, as well as book tickets in advance with a seamless experience.

**Solution:**

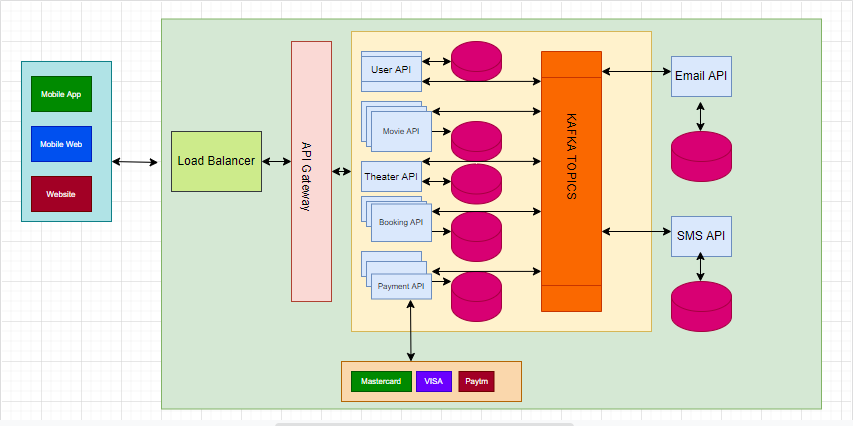
The main purpose of our online ticket booking system is to provide an alternate and convenient way for a customer to buy cinema tickets. It is an automatic system. After the data has been fed into the database, the staff does not need to do anything with the order once it is received through the system.

I developed a **Microservice architecture based solution** for problem statement. The Microservice architecture is a [software design approach](https://semaphoreci.com/blog/domain-driven-design-microservices) that decomposes an application into small independent services that communicate over well-defined APIs. Since each service can be developed and maintained by autonomous teams, it is the most scalable method for software development.

**Technologies recommended**

* Server language and Framework: Java, Spring Boot, Swagger, JPA
* Security: Spring Security
* IAM: Keycloak
* Database: MySQL
* Server: Tomcat
* Caching: Redis.
* Notifications: Apache Kafka for distributed message queue for push notifications.
* Deployment: Docker & Kubernetes
* Code repository: Git
* Logging: Log4J
* Log Management: ELK Stack
* Distributed Tracing: Spring Cloud Sleuth and Zipkin
* Monitoring: Prometheus and Grafana

**Architecture Diagram:**



**CI-CD and Deployment (Docker, Kubernetes):**

I have used Docker to build the image and gitlab-ci.yml to buld and continus Integration and deployment. I have added the below three files in movie-service project.

1. Dockerfile
2. Gitlab-ci.yml
3. Deployment-dev.yml (This contains ConfigMAP, Deployment and Service yml for deployment in Kubernetes)

**Dockerfile:**

#Stage 1

FROM maven:3.6.0-jdk-8

COPY movie-service /app

WORKDIR /app

RUN mvn clean install

#Stage 2

FROM registry.comapanyname.com/base-images/alpine-java/alpine3.11-jre8:0.1

RUN mkdir app

EXPOSE 8094

COPY --from=0 /app/target/movie-service-\*.jar /app/

WORKDIR /app

RUN adduser bhupendra -D && chown -R bhupendra /app && chmod -R a+x /app

USER bhupendra

ENTRYPOINT java -jar movie-service-\*.jar

**Gitlab-ci.yml:**

variables:

VERSION: 1.0.0

stages:

- lint

- release

- deploy

sonar:

stage: lint

image: maven:3.6.0-jdk-8

script:

- mvn -f movie-service/pom.xml clean install -s movie-service/settings.xml sonar:sonar -Dmaven.test.failure.ignore=true

build image:

stage: release

image: docker:stable

services:

- docker:dind

script:

- docker login -u ${CI\_REGISTRY\_USER} -p ${CI\_REGISTRY\_PASSWORD} ${CI\_REGISTRY}

- docker build -t ${CI\_REGISTRY\_IMAGE}:${VERSION} .

- docker push ${CI\_REGISTRY\_IMAGE}:${VERSION}

only:

- master@movie-booking-app/booking-movie-service/movie-service

deploy to server:

stage: deploy

image: registry.companyname.com:443/project-name/kube-client:1.0.0

script:

- kubectl --insecure-skip-tls-verify get pods

only:

- master@movie-booking-app/booking-movie-service/movie-service

**Kubernetes Deployment YML (ConfigMAP, Deployment and Service):**

apiVersion: v1

kind: ConfigMap

metadata:

name: movie-service

labels:

app: movie-service

component: movie-service

namespace: test-dev

data:

application.yml: |-

server:

port: 8082

spring:

application:

name: movie-service

zipkin:

enabled: true

datasource:

driver-class-name: com.mysql.cj.jdbc.Driver

password: root

url: "jdbc:mysql://localhost:3306/movie?createDatabaseIfNotExist=true"

username: root

jpa:

database-platform: org.hibernate.dialect.MySQL8Dialect

generate-ddl: true

hibernate:

ddl-auto: update

session:

jdbc:

initialize-schema: always

store-type: jdbc

timeout:

seconds: 10000

eureka:

server:

peer-node-read-timeout-ms: 25000

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka

logging:

level:

com:

movie: DEBUG

root: WARN

file:

name: 'E:/movie-app-logs/logs/elk/elk-stack.log'

management:

endpoint:

metrics:

enabled: true

prometheus:

enabled: true

endpoints:

web:

exposure:

include: prometheus

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: movie-service

namespace: test-dev

spec:

selector:

matchLabels:

app: movie-service

replicas: 1

template:

metadata:

labels:

app: movie-service

owner: boking

spec:

volumes:

- name: movie-service

configMap:

name: movie-service

items:

- key: application.yml

path: application.yml

imagePullSecrets:

- name: gitlab

containers:

- name: movie-service

image: ${image\_path}:${RELEASE\_NUMBER}

volumeMounts:

- name: user-management

mountPath: /app/configs/application.yml

subPath: application.yml

ports:

- containerPort: 8082

livenessProbe:

httpGet:

path: /movie/healthz

port: 8082

initialDelaySeconds: 60

periodSeconds: 30

timeoutSeconds: 30

readinessProbe:

httpGet:

path: /movie/healthz

port: 8082

initialDelaySeconds: 60

periodSeconds: 30

timeoutSeconds: 30

resources:

limits:

cpu: 1000m

memory: 2Gi

requests:

cpu: 50m

memory: 100Mi

---

apiVersion: v1

kind: Service

metadata:

name: movie-service

namespace: test-dev

spec:

selector:

app: movie-service

ports:

- protocol: TCP

port: 8082

targetPort: 8082

**Microservices API documentation**

Please refer the Swagger API documentation for movie-service, theatre-service, booking-service, user-services are given below:

Maven Dependencies for Swagger:

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.6.1</version>

</dependency>

<dependency>

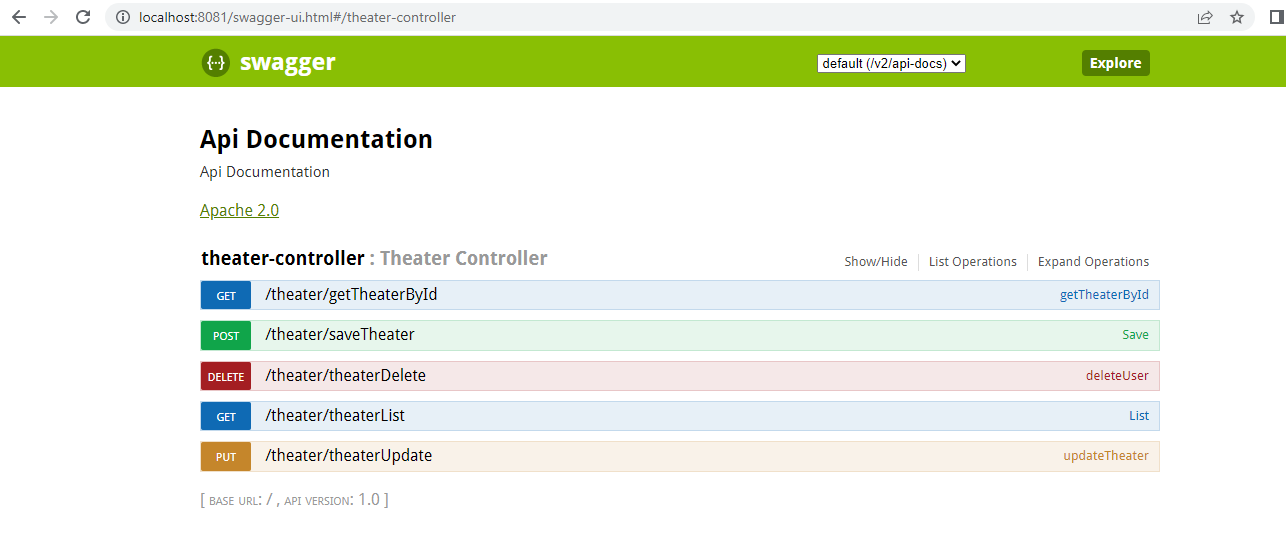
<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

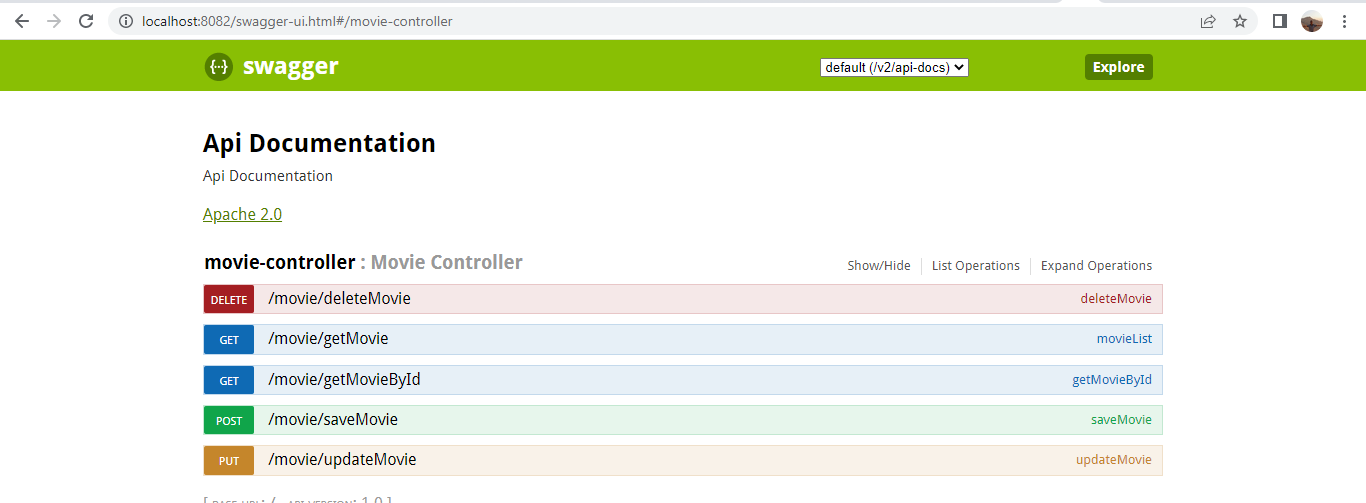
<version>2.6.1</version>

</dependency>

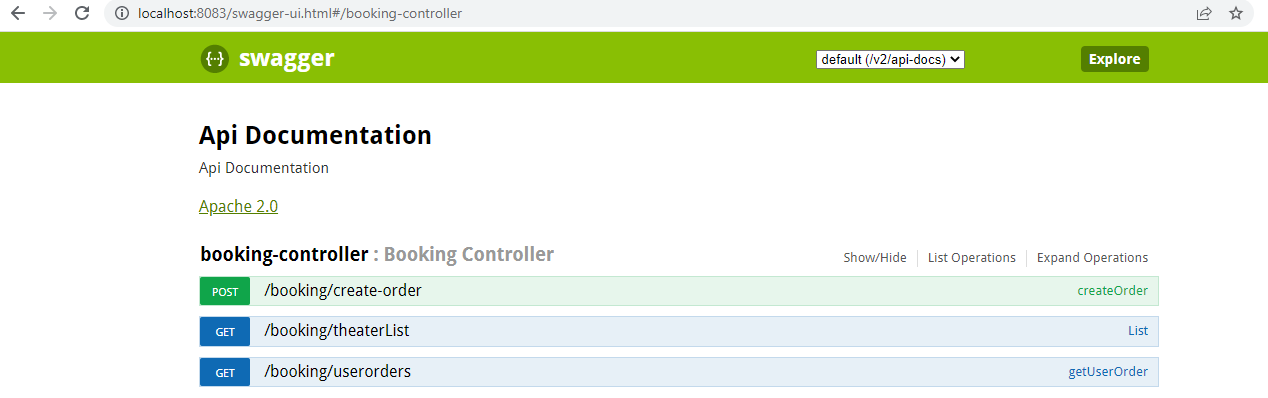
**Theater API Swagger documentation:**



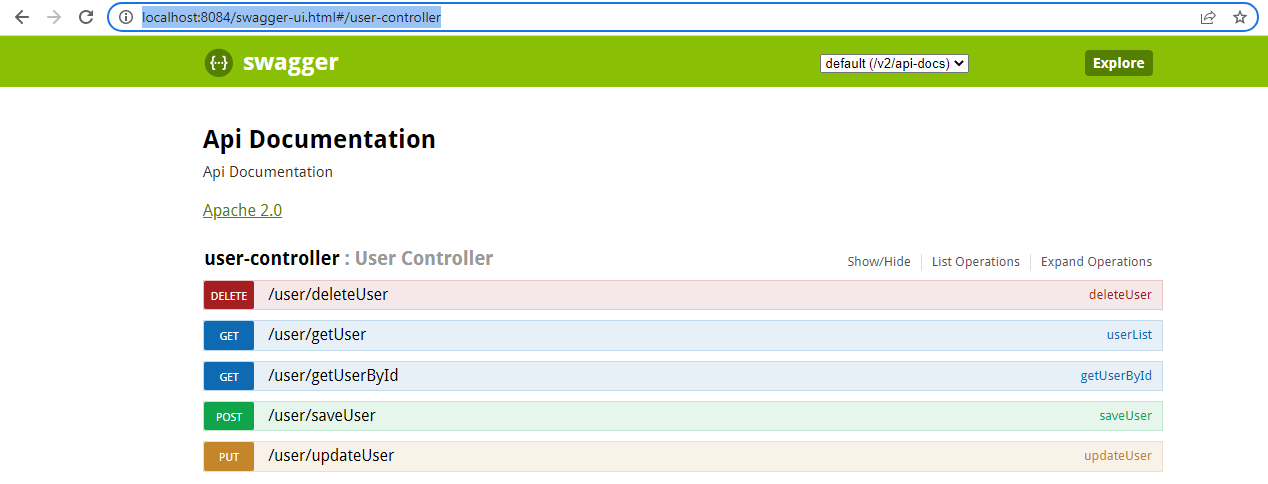
**Movie API Swagger documentation:**



**Booking API Swagger documentation:**



**User API Swagger documentation:**



**Project and Software Configuration**

https://github.com/bhupendrasingjava/moviebooking.git

Step 1: Start Keycloak 16.0.0

Step 2: Start Zipkin

E:\ZIPKIN>java -jar zipkin-server-2.23.18-exec.jar

Step 3: Start Elastic Search

E:\ELK\elasticsearch-7.6.2\bin>elasticsearch.bat

URL: http://localhost:9200/

Step 3: Start Logstash

E:\ELK\logstash-7.6.2\bin\logstash.bat -f E:\ELK\logstash-7.6.2\bin\logstash.conf

URL: http://localhost:9600/

Step 4: Uncomment the elastic search URL from E:\ELK\kibana-7.6.2\config\kibana.yml

elasticsearch.hosts: ["http://localhost:9200"] and start the kibana

Start the kibana :

E:\ELK\kibana-7.6.2\bin> kibana.bat

URL: http://localhost:5601

Step 5: Start prometheus after updating prometheus.yml

prometheus.exe --config.file prometheus.yml --web.listen-address ":9090" --storage.tsdb.path "data"

URL: <http://localhost:9090>

Step 6: Start Grafana

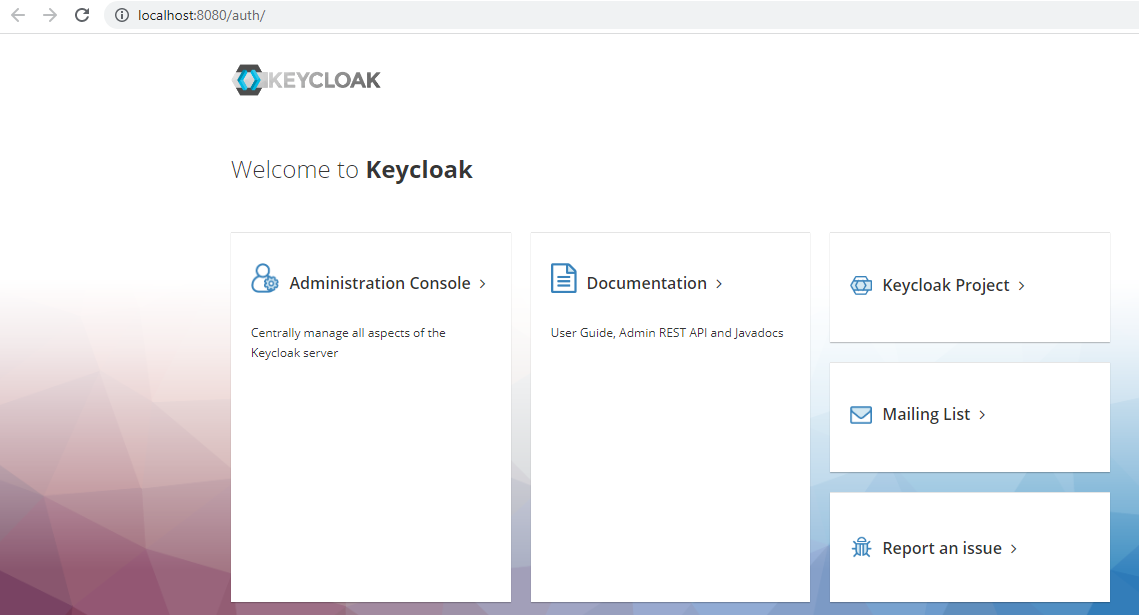
E:\grafana-7.5.16\bin>grafana-server.exe

URL http://localhost:3000/login

**Securing REST API using Keycloak and Spring Oauth2:** [Keycloak](https://www.keycloak.org/) is Open Source Identity and Access Management Server, which is a [OAuth2](https://oauth.net/2/) and [OpenID Connect](https://en.wikipedia.org/wiki/OpenID)(OIDC) protocol complaint. This article is to explain how [Spring Boot](https://spring.io/projects/spring-boot) REST APIs can be secured with Keycloak using [Spring OAuth2](https://spring.io/projects/spring-security-oauth) library.

**Step 1**: Getting Started With Keycloak : After running Keycloak, access keycloak admin console using <http://localhost:8080/auth>

Setup keycloak username=admin, password=admin.



# Step 2: Create My-Realm Realm

# Step 3: Create a Client (spring-gateway-client):

# 

# Step 4: Configure Client:  Keycloak runs on Port 8080, make sure your microservice runs on another port.

# 

# 

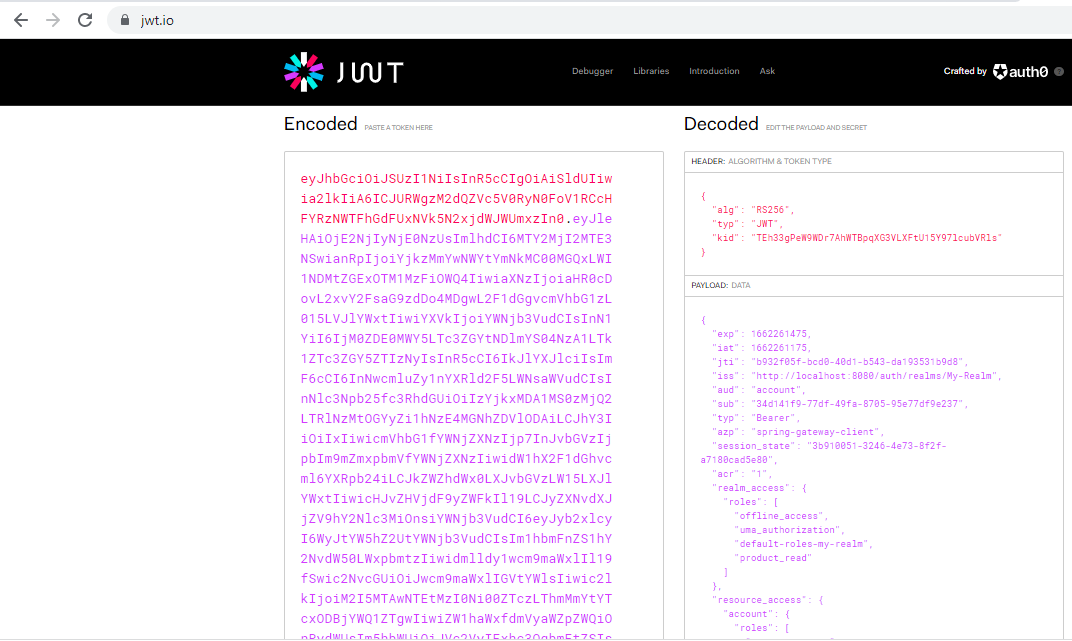
# Step 5: Get Configuration From OpenID Configuration Endpoint: *http://localhost:8080/auth/realms/My-Realm/.well-known/openid-configuration*

# 

# Step 6: Create User to access rest APIs:We created a test user to access the REST API via access token.

# Step 7: Configuration in Spring Boot Application:

# 

* **Step 8: Verifying AccessToken:** Keycloak access token is a [JWT](https://jwt.io/). It is a JSON and each field in that JSON is called a claim. By default, logged in username is returned in a claim named “preferred\_username” in access token. We may also use [**https://jwt.io**](https://jwt.io/) to inspect the contents of token received.
* 

# Step 9: Java code to get AccessToken:

# C:\Users\Lenovo\Desktop\PPT\access_token.PNG

# Step 10: Java code to call REST API using AccessToken:

# C:\Users\Lenovo\Desktop\PPT\api_call_using_new_access_token.PNG

# Step 11: Java code to get new AcessToken by using RefreshToken:

# C:\Users\Lenovo\Desktop\PPT\refresh_token.PNG

# Note: I will give live demo for this during interview.

# 

# Spring Cloud: Service Discovery With Eureka:

**Spring Cloud: Service Discovery with Eureka:** *Client-side service discovery***allows services to find and communicate with each other without hard-coding the hostname and port.** The only ‘fixed point' in such an architecture is the *service registry,* with which each service has to register. Logging and Tracing: We set up a Netflix Eureka service registry and then build a client that both registers itself with the registry and uses it to resolve its own host. A service registry is useful because it enables client-side load-balancing and decouples service providers from consumers without the need for DNS.

@SpringBootApplication

@EnableEurekaServer

**public** **class** EServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(EServerApplication.**class**, args);

}

}

**Application.yml:**

eureka:

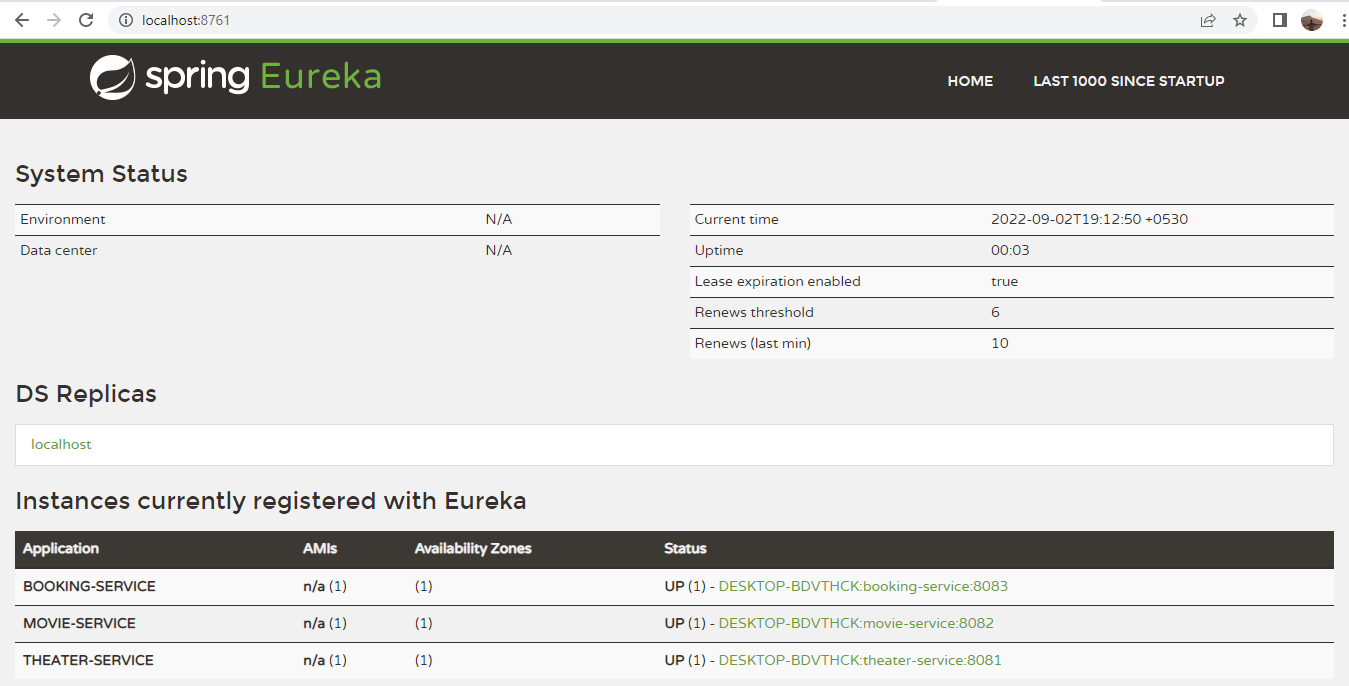
client:

registerWithEureka: false

fetchRegistry: false

serviceUrl:

defaultZone: http://localhost:8761/eureka



**Distributed Logging and Tracing:**

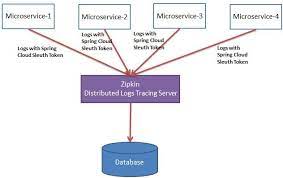
Logging is an important aspect of software development. It not only helps in troubleshooting the issues but also help us in understanding the behaviour of our Softwares. With multiple microservices, logging becomes a multifold challenge. The *Session, Thread,*or the *Http Request references* do not help much when the request is getting processed across multiple services. We will be lost, digging the logs.

**Implementing Distributed Tracing:**

Our sample implementation consists of two parts:

1: **Implementing Tracing** — In this part we will focus on pattern implementation to the core including Trace Id generation, passing it along the service calls and including it in the logs. We will be using **Spring Cloud Sleuth library** to achieve this.

2: **Enabling Tracing System** — In this part we will focus on how the traces are collected and visualized to get better insights into the service interactions. This reduces time in triage by contextualizing errors and delays. We will be using **Zipkin**to implement this.



**Maven Dependency:**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

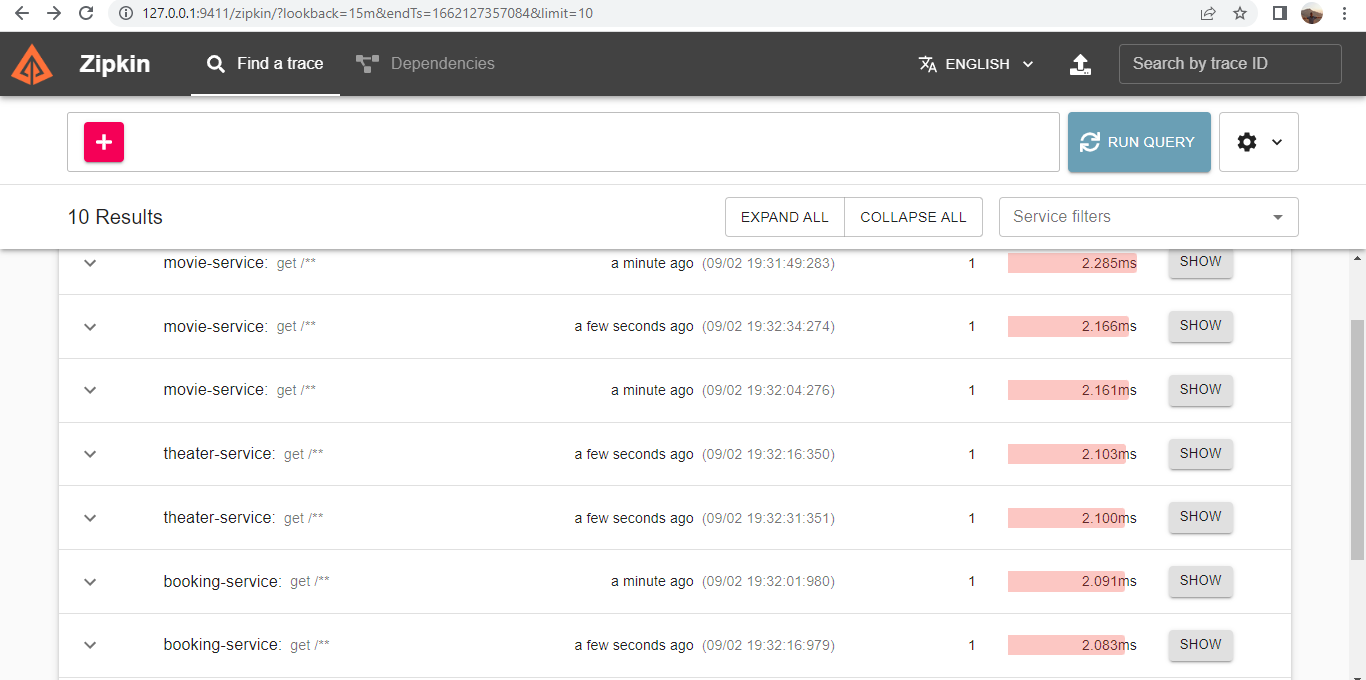
</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth-zipkin</artifactId>

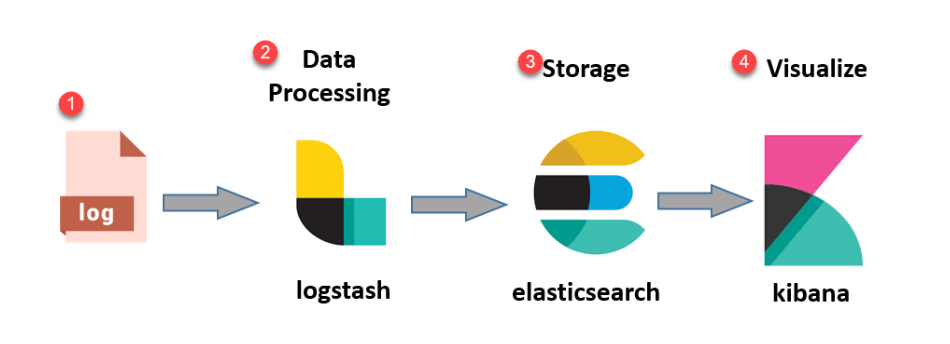
</dependency>



**Centralized logging:** The ELK Stack offers centralized logging capabilities to aggregate server logs from complex cloud environments into a single searchable index. This helps with security monitoring and root cause analysis, as data can be correlated from multiple sources.

The **ELK Stack** is a collection of three open-source products — Elasticsearch, Logstash, and Kibana. ELK stack provides centralized logging in order to identify problems with servers or applications. It allows you to search all the logs in a single place. It also helps to find issues in multiple servers by connecting logs during a specific time frame.

* **E** stands for **ElasticSearch:** used for storing logs
* **L** stands for **LogStash** : used for both shipping as well as processing and storing logs
* **K** stands for **Kibana**: is a [visualization tool](https://www.guru99.com/best-data-visualization-tools.html) (a web interface) which is hosted through Nginx or Apache



**Logstash.conf:**

input {

file {

path => "E:/movie-app-logs/logs/elk/elk-stack.log"

start\_position => "beginning"

}

}

output {

elasticsearch {

hosts => ["localhost:9200"]

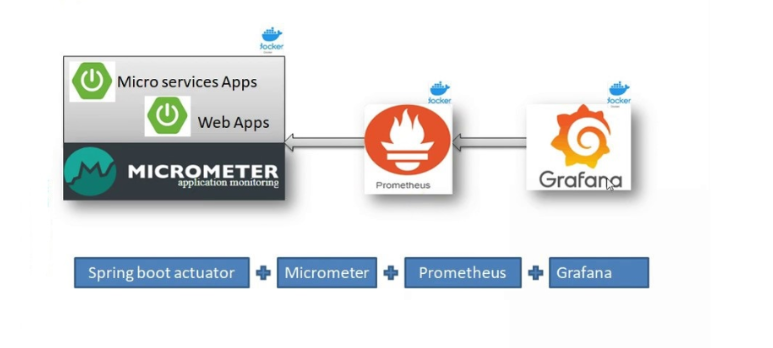
}

stdout { codec => rubydebug }

}

**Monitoring Applications with Prometheus + Grafana + Spring Boot Actuator:**

Monitoring is often critical in enterprise-level applications. Monitoring can help us prevent failures, predict trends, alert when thresholds are reached, and provide more information to troubleshoot production problems. If we can monitor in advance, we can prepare early so that we don’t get confused after an accident, but of course we can’t avoid the system from not generating a little accident, but we can reduce the number of system accidents. At the same time can also see the system problems, early optimization, to avoid greater accidents.

****

**Configuration in Springboot Application:**

**Maven Dependency:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

**Configuration in application.yml :**

management:

endpoint:

metrics:

enabled: true

prometheus:

enabled: true

endpoints:

web:

exposure:

include: prometheus

[**Prometheus**](https://prometheus.io/)**:**

[SoundCloud](https://soundcloud.com/) developed [Prometheus](https://prometheus.io/), an open-source system monitoring and alerting toolkit. Prometheus gathers and saves metrics as time series data, which means that metrics data is kept alongside the timestamp at which it was captured, as well as optional key-value pairs known as labels.

**Configuring Promotheus:**

Prometheus has a property file **prometheus.yml** .In order to show our metrics in Prometheus we need to add our configuration to this file. When we start the Spring Boot Prometheus Example, and visit to [**http://localhost:8080/actuator/prometheus**](http://localhost:8080/actuator/prometheus), we can see the statistics of the application.

# my global config

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.

rule\_files:

# - "first\_rules.yml"

# - "second\_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'prometheus'

scrape\_interval: 5s

# metrics\_path defaults to '/metrics'

# scheme defaults to 'http'.

static\_configs:

- targets: ['localhost:9090']

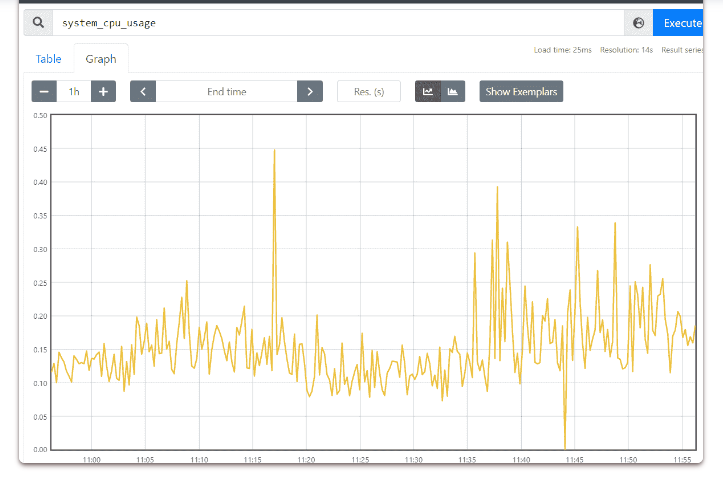
- job\_name: 'spring-actuator'

metrics\_path: '/actuator/prometheus'

scrape\_interval: 5s

static\_configs:

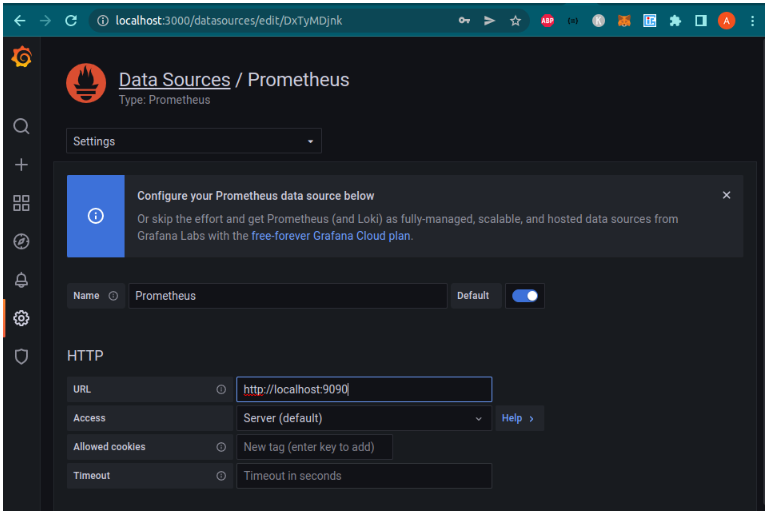
- targets: ['localhost:8081','localhost:8082','localhost:8083','localhost:8083','localhost:8761','localhost:7000']

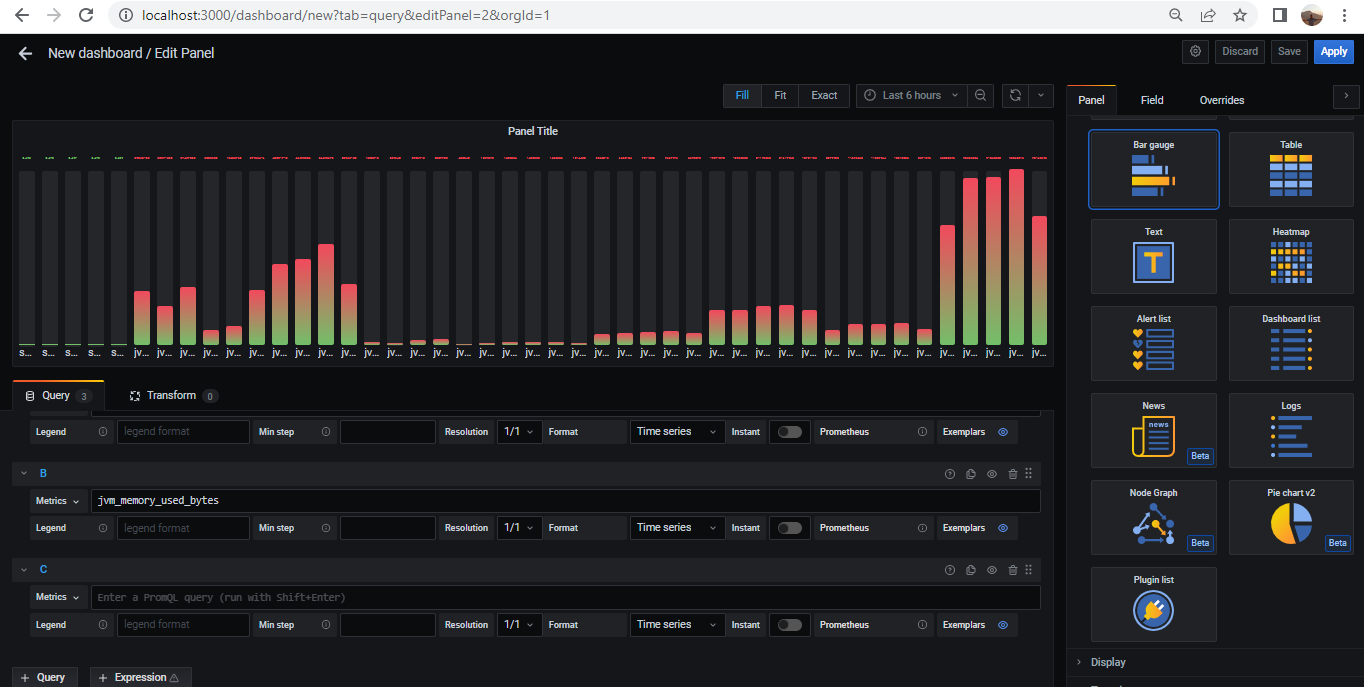


**GRAFANA:** Grafana is an open-source [observability platform](https://www.howtogeek.com/devops/what-is-observability-and-why-does-it-matter/) for visualizing metrics, logs, and traces collected from your application. Grafana connects to a variety of data sources such as [Prometheus](https://www.howtogeek.com/devops/what-is-prometheus-and-why-is-it-so-popular/), InfluxDB, [ElasticSearch](https://www.howtogeek.com/devops/how-to-set-up-an-elasticsearch-analytics-and-monitoring-panel-for-your-business/), and traditional relational database engines. Complex dashboards are created by using these sources to select relevant fields from your data. Dashboards can incorporate a varied range of visualization components such as graphs, heat maps, and histograms.

To make it easy for us and Grafana to access the Prometheus endpoint. We can open the URL http://localhost:3000 and access Grafana using the username and password as “admin”.

**Configuring Grafana Data Source:** Add the Prometheus data source. To do that, Navigate to “add a data source” and select Prometheus. Then you need to only add a single property i.e the Prometheus URL [http://localhost:9090.](http://localhost:9090./)





**Non-Functional Requirements**

Q1: Describe transactional scenarios and design decisions to address the same

Ans: Currently, there are many services serving the entire theatre requirement (as developed) so transaction will depend on multiple service. In a single service, used Spring transaction management by enabling JPA transaction while doing changes into the database.

In our case, multiple micro-service then distributed transaction management required like SAGA orchestration or choreography. For this, we should also have an asynchronous communication in place with the help of a message queue (Kafka). I do have hands on experience on this but didn’t implement it here.

Q2: Integrate with theatre having existing IT system and new theatre and localization

Ans: The functional requirement achieved with REST based services so there a theatre partner already have IT solutions, it can be easily integrate. In case of any conventional integration requirement, then we can use Adapters or any queue mechanism where the existing system can publish the data and our service can consume that.

Q3: How will you scale to multiple cities, countries and guarantee platform availability 99.99%

Ans: In a nutshell, we can say that below are the major points we need to consider

* Clustering
* Load Balancing
* Configure a monitoring tool
* CDN
* App level optimization
* Database scaling

In case of on prem, scaling the service requires data centres and deployment across the data centres so that there will be low band width while accessing the application. If we move to any Cloud platform, it can easily be achieve using platform services like AKS, EKS or GKS. We can use HPA(Horizontal Autoscaling) feature to scale the application

To increase the availability, application instances can be deployed into multiple region and traffic can be managed using Application Gateways like Azure Front Door. It will help in setting up High-Availability (HA) as well as Disaster Recovery (DR).

Q4: Integration with payment gateways

Ans: Yes, payment gateways can be integrated with application very easily. I have worked earlier to integrate an application with PayPal using their REST based integration.

Most of the payment gateways like PayPal provides a Software Development Kit (SDK) that helps programmers writing code to integrate payment into an existing application. These SDK supports many programming languages, including Java. It communicates with service providers server via Restful web services, but it’s a high-level API which means you don’t have to deal with web services directly.

Q5: How do you monetize the platform?

Ans: Monetization is the process of deriving revenue from the value you offer to your users. Your product – if it's a product worth using – is delivering meaningful value to its users in some way. It's only natural therefore that you can expect to receive something in return – including revenue - Attractive, faster and easier application interfaces.

* Advertising Space-Based Monetization Methods.
* Content monetization is a way of leveraging content so that, when users consume it, you earn money.
* Monetize payments-By integrating payments into your platform, you can offer a more convenient product experience.
* Affiliate Marketing
* License/Subscription-Based Monetization

Q6: How to protect against OWASP 10 threats?

Ans: Please refer the below solution to protect against OWSAP 10 threats:

**SQL Injection-** To prevent SQL Injection attacks is input validation and parameterized queries including prepared statements. The application code should never use the input directly. The developer must sanitize all input

**Authentication Break:** It is important to use secure communication with two-way SSL and authentication standards (such as OAuth2).

**Exposure of Sensitive Data:** To prevent this vulnerability, it is possible to use log obfuscation and data obfuscation, encrypt the communication channel, and use Two-way SSL.

**Scan your images:** Use Trivy or similar tools find vulnerabilities such as misconfigurations, missing patches, encryption weaknesses, and application bugs, including SQL Injection, Cross-Site Scripting, OWASP top 10, and more.

**SonarQube :** SonarQube to identify code level vulnerabilities and bugs.

**Platform Provisioning, sizing and release requirements**

Q1: Discuss your technology choices and decision through key drivers

Ans: Microservices can be written in different technologies like (Java,Python, GO etc. Therefore choosing the technology depends upon:

Organizational capabilities

Costing factor

Design and technology fitness with business domain

Technical Feasibility, suppose using a programming language which is not mature for the technical need

Deployment technologies etc.

- Reviews security policies and procedures, checks for an active, and successful, business continuity plan and aids in ensuring compliance requirements are met.

Avoid long checklists of requirements and instead tell digital stories. You can apply standard user-centred design (UCD) methodologies here, emphasizing use cases, stories or top tasks — it doesn't matter what you call them. Just do it.

That way, when vendors pursue a competition that includes demos and prototypes and training, the experience becomes interactive (because you’re getting hands-on) and real (because you’re telling your company’s unique stories).

Q2: Discuss database, transactions and data modelling

Ans: Database design must follow the functional needs by keeping is simple. If application designed with micro-service design pattern, it supports both model like database per micro- service and shared database concept.

The decision made based upon factors like: Size and complexity of data to be used in solution, Data centric business logic handling etc.

The transaction manage depends upon need like, if the transaction does not spread across the micro-service, the transaction must be handled internal to micro-service. Otherwise, SAGA

design can be implemented to handle distributed transaction management with middleware solutions like Eventuate-tram.

Q3: Discuss enterprises systems that you may need to manage specific areas Ans: Not much clear about this question ask.

Have worked in enterprises application used in banking domain which spreads across the different system to complete the business flows.

Q4: Discuss hosting solutions and sizing

Mainly three ways of hosting available:

Cloud: Application can be hosted in any major cloud vendors like AWS, Azure, GCP, OCI etc.

Hybrid: Where an application deployed in both on-prem and Cloud.

Each hosting solution has their own pros and cons. Also user base, cost, availability,

performance etc are major factors which are considered while taking hosting solution decision. [Cloud hosting](https://www.dreamhost.com/blog/dreamhost-builds-cloud/) is where things get a little trickier. When people talk about the “cloud,” they’re usually referring to distributed VPSs. The way this works is that instead of having a virtual server stored on one computer, it’s hosted on a whole network of them.

Application sizing mainly depends on concurrent users and performance of the application. In case of user base grows, the application instances also need to increase to handle the load.

Q5: Discuss release management across cities, language etc.

Application release across the cities, countries, languages depends upon several factors e.g.:

Application design: If application developed to handle multiple countries, language, same code based can be release across the regions.

Optimize test environment method

Define stages and controlling activities

Ensure clarity with stakeholder engagement

Enable continuous communication

Track metrics

User Acceptance Testing is during the end-users the product was created to use the product and give feedback related to the product. This is usually performed as a free beta test shared with a collection of employees within the organization. UAT is the most crucial action to release management because of the number of data gathered and fixes needed to prepare the build to anywhere it needs to be for the approved launch

Governing policies: It could be organizational or government policies as which influence the release management

Infrastructure Availability: In certain cities, it may be possible required parters are not available, infra is not available. It release management requires to analyse the facts and decide how and when application will be released.

Q6: Provide details on monitoring system

Ans: Application monitoring covers below aspects:

Application Log – Log monitoring helps in many aspects like production support for customers, identifying bugs / enhancement etc.

Application hardware metrics – Helps in identification hardware utilization with load and provides insight hardware usages in application.

Monitoring improves your customer service and corporate image

By avoiding falls in the service and minimizing the time of resolution, the overall customer service will be improved.

- System alerts-Once a particular event happens, a notice, which will reach the right people, will be generated. Alerts can normally be configured so that the event that triggers them can be chosen by its owner. For example, a hard disk reaching its maximum capacity, its 80% capacity or whatever you need to monitor in that particular hard disk.

-Notifications-Alerts are important, but more important is the way we receive them. Via email, SMS

-Monitoring tool like ELK, Zabbix, Nagios, Genious are some tools which helps in liveness probe. It can be used easily with the project.

Q7: Discuss overall KPIs .

Below are few KPIs:

Development Velocity: Using the technologies used in this project, easy and fast development can be achieved as Java and Spring are very popular language and platform which helps in easy setup and manage development team

Release – The DevOps can be easily used with this project to achieve high automation for build and release application very easily with lower the manual effort and cost as well

High Performance – with underlying technologies, application can easily deployment into multiple regions, become highly available

Q8: Create a high-level project plan and estimate breakup

**PROJECT PLAN**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PROJECT NAME** | Movie Ticket Booking | | | **PROJECT MANAGER** | Bhupendra Singh |
| **PROJECT LOCATION** | Noida | | | **PROJECT DELIVERABLE** |  |
| **PROJECT SCOPE** | POC | | | | |
| **COMPANY** | Sapient | | | | |
| **CONTACT NAME** | Bhupendra Singh | | | | |
| **MAILING ADDRESS** | B-121 Assotech The Nest, Crossing Republic, Ghaziabad, UP, India, 201016 | | | | |
| **EMAIL** | Bhupendra.java.mobility@gmail.com | | | **PHONE** | 9456667341 |
| **START DATE** | 9/11/2022 | **END DATE** | 1/31/2023 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SL. No.** | **ACTIVITY/ TASK NAME** | **RESOURCE ASSIGNED** | **START DATE** | **END DATE** | **DURATION**  (in days) | **TASK STATUS** |
| **1** | Base Framework Setup | Team member - 1 | 9/12/2022 | 9/16/2022 | 5 | Completed |
| **2** | Movie-Service – Add movie | Team member - 2 | 9/19/2022 | 9/19/2022 | 1 | Completed |
| **3** | Movie-Service – Update movie | Team member - 2 | 9/20/2022 | 9/20/2022 | 1 | Completed |
| **4** | Movie-Service – Get movie by ID and Get Movie List | Team member - 1 | 9/19/2022 | 9/19/2022 | 1 | Completed |
| **5** | Movie Service – Delete Movie and API document updating | Team member - 1 | 9/20/2022 | 9/20/2022 | 2 | Completed |
| **6** | Junit and Sonar issue verification and code review for movie module | Team member - 2 | 9/21/2022 | 9/22/2022 | 2 | Completed |
| **7** | Theater-Service – Add Theater | Team member - 3 | 9/19/2022 | 9/19/2022 | 1 | Completed |
| **8** | Theater-Service – Update Theater | Team member - 3 | 9/20/2022 | 9/20/2022 | 1 | Completed |
| **9** | Theater-Service – Get Theater by ID and Get Theater List | Team member - 1 | 9/21/2022 | 9/21/2022 | 1 | Completed |
| **10** | Theater Service – Delete Movie and API document updating | Team member - 1 | 9/22/2022 | 9/23/2022 | 2 | In Progress |
| **11** | Junit and Sonar issue verification and code review for Theater module | Team member - 2 | 9/23/2022 | 9/26/2022 | 1 | In Progress |