[**https://www.tutorialsteacher.com/sqlserver/tables-relations**](https://www.tutorialsteacher.com/sqlserver/tables-relations)

<https://www.knowledgefactory.net/2023/01/angular-spring-boot-hello-world-example.html>

https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/

|  |  |  |  |
| --- | --- | --- | --- |
| **Basic Git Commands**    **Git Init**  **Git Add**  **Git Commit**  **Git Clone**  **Git Repository**  **Git Index**  **Git Pull**  **Git Push**  **Git Checkout**  **Git Rebase**  **Git Ignore**  **Git Head**  **Git Stash**  **Git Ignore**  **Git Fork**  **Git Head**  **Git Origin Master**  **Git Remote**  **Git TagsUpstream & Downstream**    **Collaborating**    **Git Fetch**  **Git Pull**  **Git Push**  **Branching & Merging**    **Git BranchMerge & Merge Conflict**  **Git Rebase**  **Git Squash**      **Undoing changes**    **Git Checkout**  **Git Revert**  **Git Reset**  **Git Rm**  **Git Cherry-pick** | **Docker  commands**    **docker –version**  **docker push**  **docker pull**  **docker images**  **docker commit**  **docker build**  **docker run**  **docker stop**  **docker login**  **docker ps**  **docker ps -a**  **docker exec**  **docker kill**  **docker rm**  **docker rmi** | **Linex Commands.**    **1. su Command**    **1. pwd Command**    **2. mkdir Command**    **3. rmdir Command**    **4. ls Command**    **5. cd Command**    **6. touch Command**    **7. cat Command**    **8. rm Command**    **9. cp Command**    **10. mv Command**    **12. head Command**    **13. tail Command**    **14. tac Command**    **18. id Command**    **19. useradd Command**    **20. passwd Command**    **21. groupadd Command**    **22. cat Command or tac Command**    **23. cut Command**    **24. grep Command**    **27. tee command**    **30. wc Command**    **33. gzip Command**    **34. gunzip Command**    **44. exit Command** | **Https status code.**    **200 OK**  **201 Created**  **202 Accepted**  **203 Non-authoritative Information**  **204 No Content**  **205 Reset Content**    **300 Multiple Choices**  **301 Moved Permanently**  **302 Found**  **303 See Other**  **304 Not Modified**  **305 Use Proxy**    **400 Bad Request**  **401 Unauthorized**  **402 Payment Required**  **403 Forbidden**  **404 Not Found**  **405 Method Not Allowed**  **406 Not Acceptable**  **407 Proxy Authentication Required**  **408 Request Timeout**  **409 Conflict**    **500 Internal Server Error**  **501 Not Implemented**  **502 Bad Gateway**  **503 Service Unavailable**  **504 Gateway Timeout**  **505 HTTP Version Not Supported** |
|  |  |  |  |

# What is Apache Kafka?

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

### ****What role does ZooKeeper play in a cluster of Kafka?****

Apache ZooKeeper acts as a distributed, open-source configuration and synchronization service, along with being a naming registry for distributed applications. It keeps track of the status of the Kafka cluster nodes, as well as of Kafka topics, partitions, etc.

Since the data is divided across collections of nodes within ZooKeeper, it exhibits high availability and consistency. When a node fails, ZooKeeper performs an instant failover migration.

ZooKeeper is used in Kafka for managing service discovery for Kafka brokers, which form the cluster. ZooKeeper communicates with Kafka when a new broker joins, when a broker dies, when a topic gets removed, or when a topic is added so that each node in the cluster knows about these changes. Thus, it provides an in-sync view of the Kafka cluster configuration.

### ****Can Kafka be utilized without ZooKeeper?****

It is impossible to use Kafka without ZooKeeper because it is not feasible to go around ZooKeeper and attach it in a straight line with the server. If ZooKeeper is down for a number of causes, then we will not be able to serve customers’ demands.

### ****How to start a Kafka server?****

Given that Kafka exercises ZooKeeper, we can start the ZooKeeper’s server. One can use the convince script packaged with Kafka to get a crude but effective single-node ZooKeeper instance:

bin/zookeeper-server-start.shconfig/zookeeper.properties

Now the Kafka server can start:

bin/Kafka-server-start.shconfig/server.properties

## List the various components in Kafka.

The four major components of Kafka are:

* Topic – a stream of messages belonging to the same type
* Producer – that can publish messages to a topic
* Brokers – a set of servers where the publishes messages are stored
* Consumer – that subscribes to various topics and pulls data from the brokers.

## 3. Explain the role of the offset.

Messages contained in the partitions are assigned a unique ID number that is called the offset. The role of the offset is to uniquely identify every message within the partition.

## 4. What is a Consumer Group?

Consumer Groups is a concept exclusive to Kafka.  Every Kafka consumer group consists of one or more consumers that jointly consume a set of subscribed topics.

## Explain the concept of Leader and Follower.

Every partition in Kafka has one server which plays the role of a Leader, and none or more servers that act as Followers. The Leader performs the task of all read and write requests for the partition, while the role of the Followers is to passively replicate the leader. In the event of the Leader failing, one of the Followers will take on the role of the Leader. This ensures load balancing of the server.

## 8. What roles do Replicas and the ISR play?

Replicas are essentially a list of nodes that replicate the log for a particular partition irrespective of whether they play the role of the Leader. On the other hand, ISR stands for In-Sync Replicas. It is essentially a set of message replicas that are synced to the leaders.

## 9. Why are Replications critical in Kafka?

Replication ensures that published messages are not lost and can be consumed in the event of any machine error, program error or frequent software upgrades.

## 10. If a Replica stays out of the ISR for a long time, what does it signify?

It means that the Follower is unable to fetch data as fast as data accumulated by the Leader.

## 11. What is the process for starting a Kafka server?

Since Kafka uses ZooKeeper, it is essential to initialize the ZooKeeper server, and then fire up the Kafka server.

* To start the ZooKeeper server: > bin/zookeeper-server-start.sh config/zookeeper.properties
* Next, to start the Kafka server: > bin/kafka-server-start.sh config/server.properties

## 12. How do you define a Partitioning Key?

Within the Producer, the role of a Partitioning Key is to indicate the destination partition of the message. By default, a hashing-based Partitioner is used to determine the partition ID given the key. Alternatively, users can also use customized Partitions.

## 13. In the Producer, when does QueueFullException occur?

QueueFullException typically occurs when the Producer attempts to send messages at a pace that the Broker cannot handle. Since the Producer doesn’t block, users will need to add enough brokers to collaboratively handle the increased load.

## 14. Explain the role of the Kafka Producer API.

The role of Kafka’s Producer API is to wrap the two producers – kafka.producer.SyncProducer and the kafka.producer.async.AsyncProducer. The goal is to expose all the producer functionality through a single API to the client.

## 15. What is the main difference between Kafka and Flume?

Even though both are used for real-time processing, Kafka is scalable and ensures message durability.

### What is the main difference between Kafka and Flume?

Ans. The main difference between Kafka and Flume are: **Types of tool**

* Apache Kafka– As Kafka is a general-purpose tool for both multiple producers and consumers.
* Apache Flume– Whereas, Flume is considered as a special-purpose tool for specific applications.

**Replication feature**

* Apache Kafka– Kafka can replicate the events.
* Apache Flume- whereas, Flume does not replicate the events.

### Q.20 Is Apache Kafka is a distributed streaming platform? if yes, what you can do with it?

Ans. Undoubtedly, Kafka is a streaming platform. It can help: **To push records easily**. Also, can store a lot of records without giving any storage problems Moreover, it can process the records as they come in

### Q. 21 What can you do with Kafka?

Ans. It can perform in several ways, such as:

* In order to transmit data between two systems, we can build a real-time stream of data pipelines with it.
* Also, we can build a real-time streaming platform with Kafka, that can actually react to the data.

### Q.22 What is the purpose of retention period in Kafka cluster?

Ans. However, retention period retains all the published records within the Kafka cluster. It doesn’t check whether they have been consumed or not. Moreover, the records can be discarded by using a configuration setting for the retention period. And, it results as it can free up some space.

### Q.23 Explain the maximum size of a message that can be received by the Kafka?

Ans. The maximum size of a message that can be received by the Kafka is approx. 1000000 bytes.

### Q.24 What are the types of traditional method of message transfer?

Ans. Basically, there are two methods of the traditional message transfer method, such as:

* **Queuing**: It is a method in which a pool of consumers may read a message from the server and each message goes to one of them.
* **Publish-Subscribe**: Whereas in Publish-Subscribe, messages are broadcasted to all consumers.

### Q.25 What does ISR stand in Kafka environment?

Ans. ISR refers to In sync replicas. These are generally classified as a set of message replicas which are synced to be leaders.

### Q.26 What is Geo-Replication in Kafka?

Ans. For our cluster, Kafka MirrorMaker offers geo-replication. Basically, messages are replicated across multiple data centers or cloud regions, with MirrorMaker. So, it can be used in active/passive scenarios for backup and recovery; or also to place data closer to our users, or support data locality requirements.

### Q.27 Explain Multi-tenancy?

Ans. We can easily deploy Kafka as a multi-tenant solution. However, by configuring which topics can produce or consume data, Multi-tenancy is enabled. Also, it provides operations support for quotas.  
<https://softwareengineering.stackexchange.com/questions/351524/what-exactly-is-a-multi-tenant-application>

### Q.28 What is the role of Consumer API?

Ans. An API which permits an application to subscribe to one or more topics and also to process the stream of records produced to them is what we call Consumer API.

### Q.29 Explain the role of Streams API?

Ans. An API which permits an application to act as a stream processor, and also consuming an input stream from one or more topics and producing an output stream to one or more output topics, moreover, transforming the input streams to output streams effectively, is what we call Streams API.

### Q.30 What is the role of Connector API?

Ans. An API which permits to run as well as build the reusable producers or consumers which connect Kafka topics to existing applications or data systems is what we call the Connector API.

### Q.31 Explain Producer?

Ans. The main role of Producers is to publish data to the topics of their choice. Basically, its duty is to select the record to assign to partition within the topic.

### Explain the term “Topic Replication Factor”.

Ans. It is very important to factor in topic replication while designing a Kafka system. Hence, if in any case, broker goes down its topics’ replicas from another broker can solve the crisis.

### What are Replication Tool and its types?

Ans. For the purpose of stronger durability and higher availability, replication tool is available here. Its types are −

* Create Topic Tool
* List Topic Tool
* Add Partition Tool

## Mention what is the Maximum Size of the Message does Kafka server can Receive?

The maximum size of the message that Kafka server can receive is 1000000 bytes.

## Why Replication is required in Kafka?

Replication of message in Kafka ensures that any published message does not lose and can be consumed in case of machine error, program error or more common software upgrades.

## Explain the concept of Leader and Follower?

Every partition in Kafka has one server which plays the role of a Leader, and none or more servers that act as Followers. The Leader performs the task of all read and write requests for the partition, while the role of the Followers is to passively replicate the leader. In the event of the Leader failing, one of the Followers will take on the role of the Leader. This ensures load balancing of the server.

## How do you define a Partitioning Key?

Within the Producer, the role of a Partitioning Key is to indicate the destination partition of the message. By default, a hashing-based Partitioner is used to determine the partition ID given the key. Alternatively, users can also use customized Partitions.

### ****What do you know about a partition key?****

A partition key is used to point to the aimed division of communication in Kafka producer. Usually, a hash-oriented divider concludes the division ID with the input, and also people use modified divisions.

### ****Describe an Offset.****

The messages in partitions will be given a sequential ID known as an offset, and the offset will be used to identify each message in the partition uniquely. With the aid of ZooKeeper, Kafka stores the offsets of messages used for a specific topic and partition by a consumer group.

### ****What major role does a Kafka Producer API play?****

It is responsible for covering two producers: **kafka.producer.SyncProducer** and **kafka.producer.async.AsyncProducer**. Kafka Producer API mainly provides all producer performance to its clients through a single API.

**Apache Kafka Core Concepts**

We will discuss following Apache Kafka core concepts:

1. Kafka Cluster  
2. Kafka Broker  
3. Kafka Producer  
4. Kafka Consumer  
5. Kafka Topic  
6. Kafka Partitions  
7. Kafka Offsets  
8. Kafka Consumer Group

## 1. Kafka Cluster

Since Kafka is a distributed system, it acts as a cluster. A Kafka cluster consists of a set of brokers. A cluster has a minimum of 3 brokers.

The following diagram shows Kafka cluster with three Kafka brockers:

**[](https://blogger.googleusercontent.com/img/a/AVvXsEi9hrIWpPIyRZ5rVEc0K3UZ7_s2IUxnZvF5DB0W5Pp9mfZQFKuiwVsasWAlUOueWCQv9z8qR8DcoZqIxImgjEpmDrQHAne1y7odpkE87e-QGzwWCarB2k8ZA_ZH07AMft9ZPj7xLZXz8cu4ugCDU5c2XCeJTChNwJ-8zb9MWxdvkzAx-Z1wEPqZK2JZ)**

## . Kafka Broker

The broker is the Kafka server. It's just a meaningful name given to the Kafka server. And this name makes sense as well because all that Kafka does is act as a message broker between producer and consumer.

The producer and consumer don't interact directly. They use the Kafka server as an agent or a broker to exchange messages.

The following diagram shows a Kafka broker, it acts as an agent or broker to exchange messages between Producer and Consumer:

**[](https://blogger.googleusercontent.com/img/a/AVvXsEiBTAZMBDniTdXHsiadf7DJbypCOFO0WjQhj5PnWAYTF58saqG6sbQu6CJBXFuB3mvbtm_D29IXzwScxUwyXGyuT3Sq3XCzymEGVUBReiEWfn7EcV-mPpRRXCIvIiSs6FBGLkcjb7YJRV3ysSoUMc5unD7G5867FY77MLxk95I_LeHYi1cH71TOPe8w)**

## 3. Kafka Producer

Producer is an application that sends messages. It does not send messages directly to the recipient. It sends messages only to the Kafka server.

The following diagram shows Producer sends messages directly to Kafka broker:

**[](https://blogger.googleusercontent.com/img/a/AVvXsEiBTAZMBDniTdXHsiadf7DJbypCOFO0WjQhj5PnWAYTF58saqG6sbQu6CJBXFuB3mvbtm_D29IXzwScxUwyXGyuT3Sq3XCzymEGVUBReiEWfn7EcV-mPpRRXCIvIiSs6FBGLkcjb7YJRV3ysSoUMc5unD7G5867FY77MLxk95I_LeHYi1cH71TOPe8w)**

## 4. Kafka Consumer

Consumer is an application that reads messages from the Kafka server.

If producers are sending data, they must be sending it to someone, right? The consumers are the recipients. But remember that the producers don't send data to a recipient address. They just send it to the Kafka server.

Anyone who is interested in that data can come forward and take it from the Kafka server. So, any application that requests data from a Kafka server is a consumer, and they can ask for data sent by any producer provided they have permission to read it.

The following diagram shows Producer sends messages directly to the Kafka broker and the Consumer consumes or reads messages from the Kafka broker:

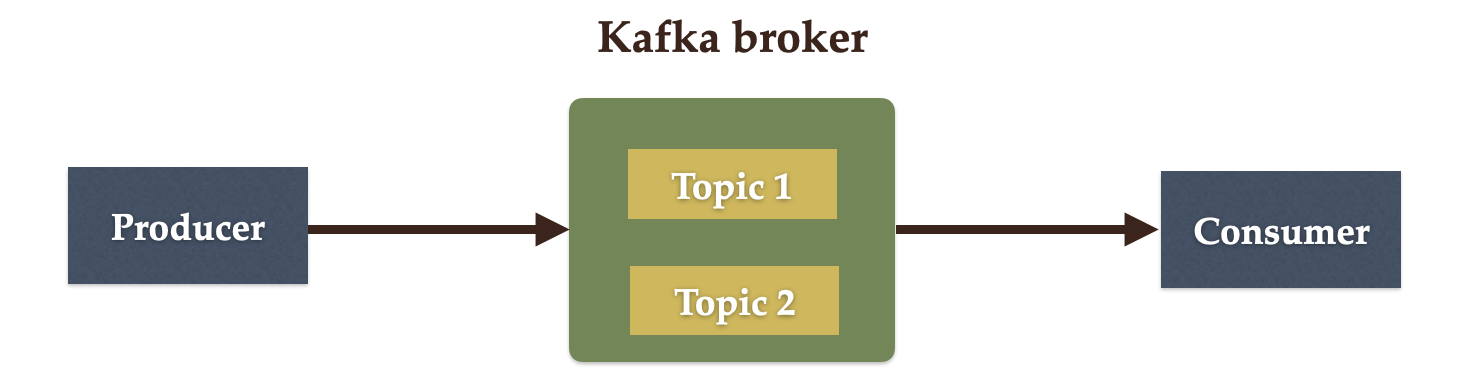
**[](https://blogger.googleusercontent.com/img/a/AVvXsEiBTAZMBDniTdXHsiadf7DJbypCOFO0WjQhj5PnWAYTF58saqG6sbQu6CJBXFuB3mvbtm_D29IXzwScxUwyXGyuT3Sq3XCzymEGVUBReiEWfn7EcV-mPpRRXCIvIiSs6FBGLkcjb7YJRV3ysSoUMc5unD7G5867FY77MLxk95I_LeHYi1cH71TOPe8w)**

## 5. Kafka Topic

We learned that producer sends data to the Kafka broker. Then a consumer can ask for data from the Kafka broker. But the question is, Which data? We need to have some identification mechanism to request data from a broker. There comes the Kafka topic.

* Topic is like a table in a database or folder in a file system.
* Topic is identified by a name.
* You can have any number of topics.

The following diagram shows two Topics are created in a Kafka broker:

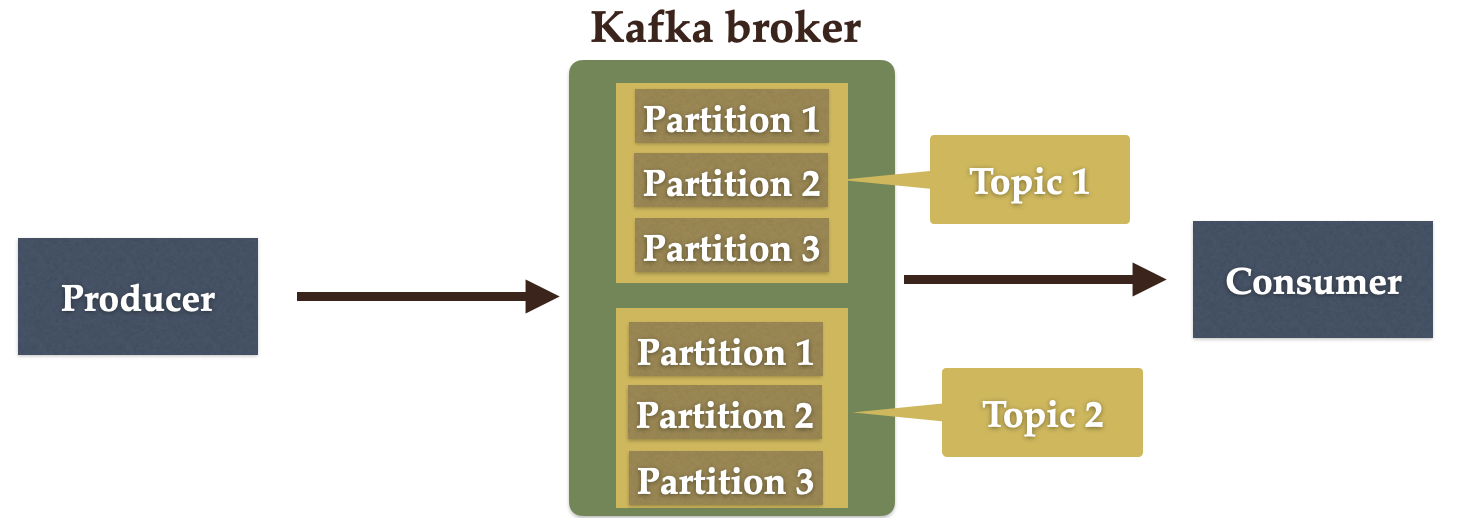
**[](https://blogger.googleusercontent.com/img/a/AVvXsEhlKR0H-usK8tqw4l6ZBotEdj0GlufBMIYymVIarTvZdq0mRwm4tmB0zZIuSFFEGh2r_c5oK2J2TYVxbzufIs3leif1K0ZOOG6gTZPI7_5vsGcSlvEHMBJVGM-WjSudn17PTFwcu4BBqb2Ip8Q9vZke57l7DmVOqOJXYyRaPGhjpX4yw47yOagYN_ws)**

## 6. Kafka Partitions

Kafka topics are divided into a number of partitions, which contain records in an unchangeable sequence.

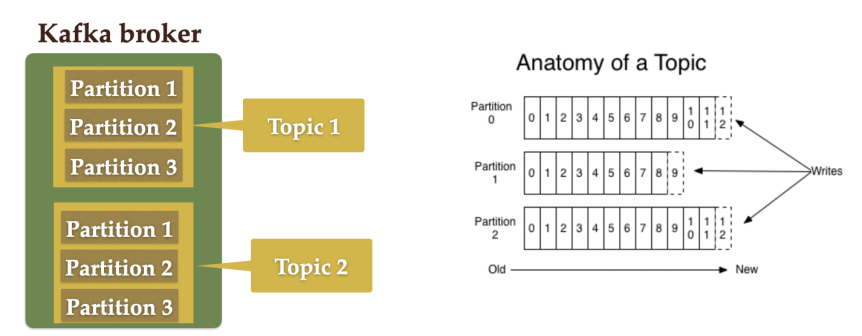
Kafka Brokers will store messages for a topic. But the capacity of data can be enormous and it may not be possible to store in a single computer. Therefore it will be partitioned into multiple parts and distributed among multiple computers since Kafka is a distributed system.

The following diagram shows Kafka's topic is further divided into a number of partitions:

**[](https://blogger.googleusercontent.com/img/a/AVvXsEjy5qEhC7_hrTlauMSEdhjOkybBZVp44eVVIpASv-s3-llmp_lqf9MqArwSu-LhIq6qovoOBXwcYz0VZK7fjsniTCjxUi09UlhhO4-R6VYZSpA9ZJyskfGXZaOJQTnt93fvcUdaoY9KEyph6QfeTQegZ5Os5Y_tVoV-UwPl1Ca-jrTpx89SnmWkh3L3)**

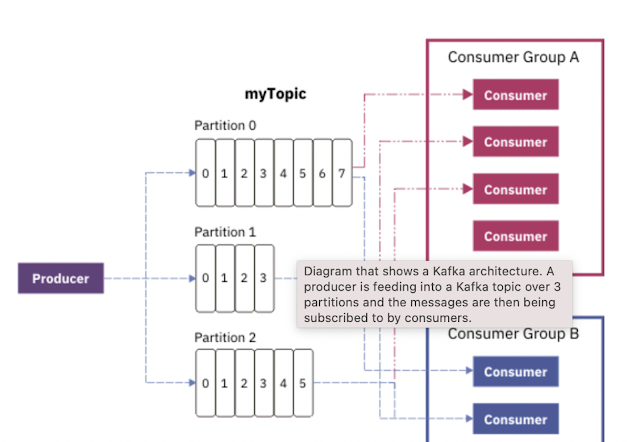
## 7. Kafka Offsets

Offset is a sequence of ids given to messages as they arrive at a partition. Once the offset is assigned it will never be changed. The first message gets an offset zero. The next message receives an offset one and so on.

**[](https://blogger.googleusercontent.com/img/a/AVvXsEgLjNHk8K7-N07M7jGV7a_pSMB94bH0e1S4iWghXORilNuVztnJiKMZsbHLt9rQvjoEY8tz4vCafPTwBecVEwO1z5lDXqdI2qOGsnxWwVEjNoL1U_SYGcQdsigcSuRHYzBxNaG4yF0G7cU82nxHRaIA6YtNueBwVyTonbD8t8cmrJ_SK-NppnLXuqoZ)**

## 8. Kafka Consumer Group

A consumer group contains one or more consumers working together to process the messages.

**[](https://blogger.googleusercontent.com/img/a/AVvXsEipRZwK4ijG2kIS4YhFXs0tSxXxRneQwp_BID8Kr5uPRQTWL6ps8F2z-bukwkGDJVTE-EBK8u2poCdJ-He4_O0Sie_KOPChIOZACHBA6A-1T4lqSVe_qO2064m-A93pPK4y6YhJYhyEXxInPw8I04qLxFFrrUcfV8q2ZNl7LrmFXB3XHc_D1lupVKwP)**

# 

# 

# kafka-publisher

Apache Kafka Publisher Example using SpringBoot

# start zookeeper.start bat file like below

zookeeper-server-start.bat D:\DEV-SOFTWARES\kafka\_2.12-1.1.0\config\zookeeper.properties

# start kafka server

kafka-server-start.bat D:\DEV-SOFTWARES\kafka\_2.12-1.1.0\config\server.properties

# Create Topic:

kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 -topic javatechie

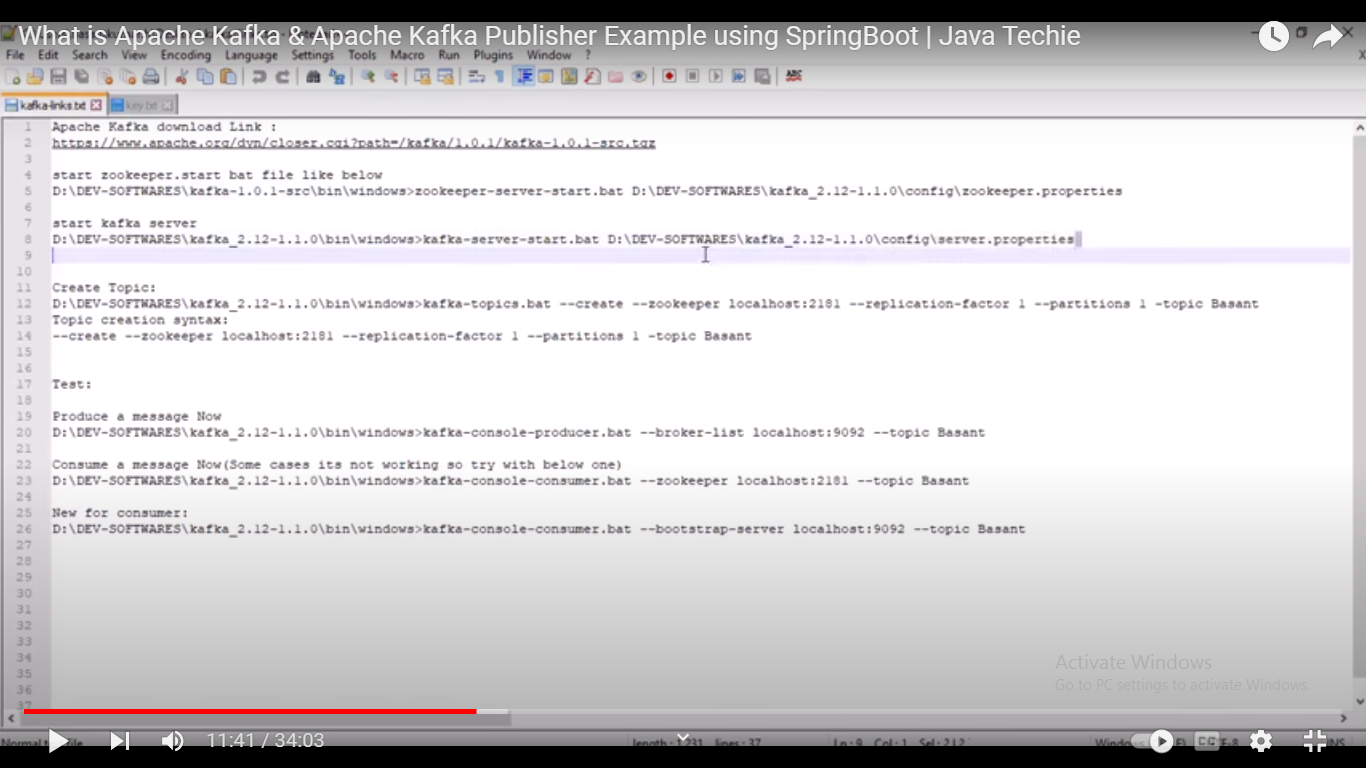
# Produce a message

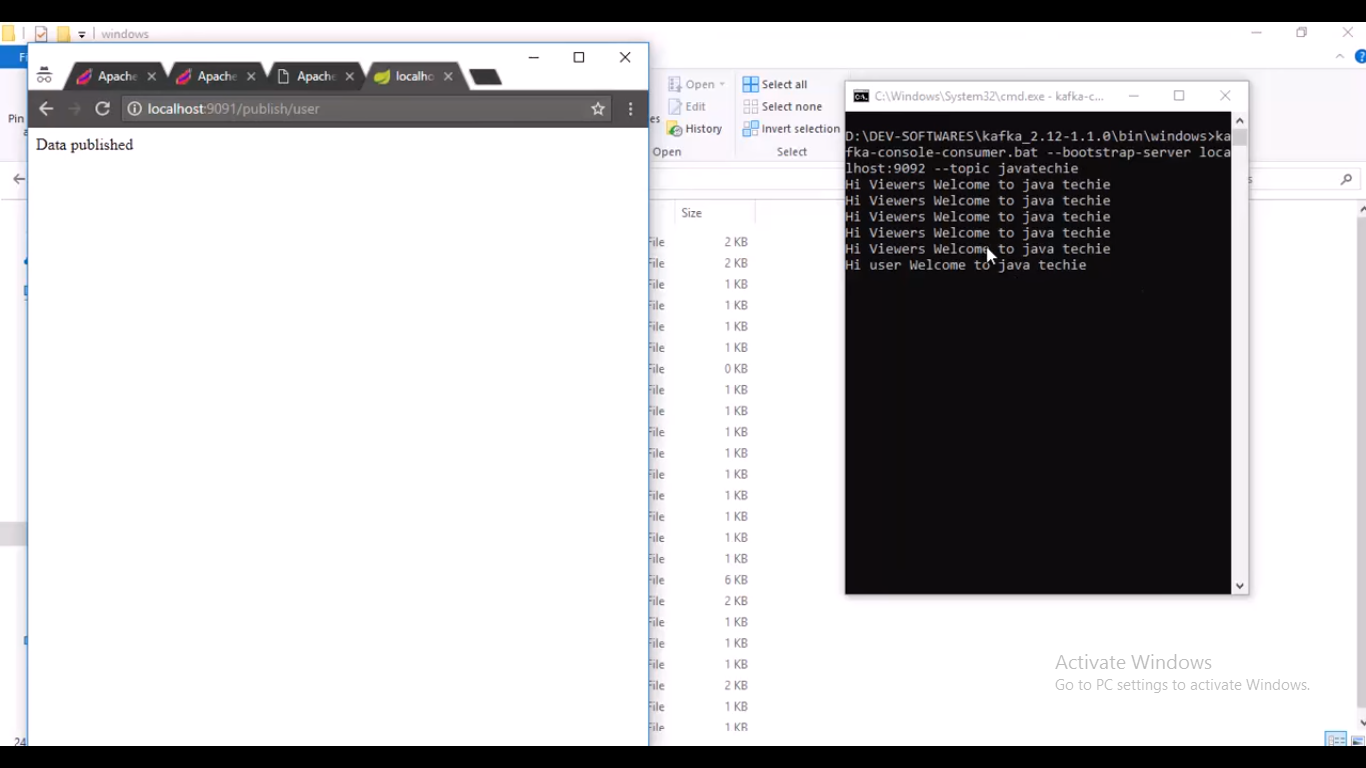
kafka-console-producer.bat --broker-list localhost:9092 --topic javatechie

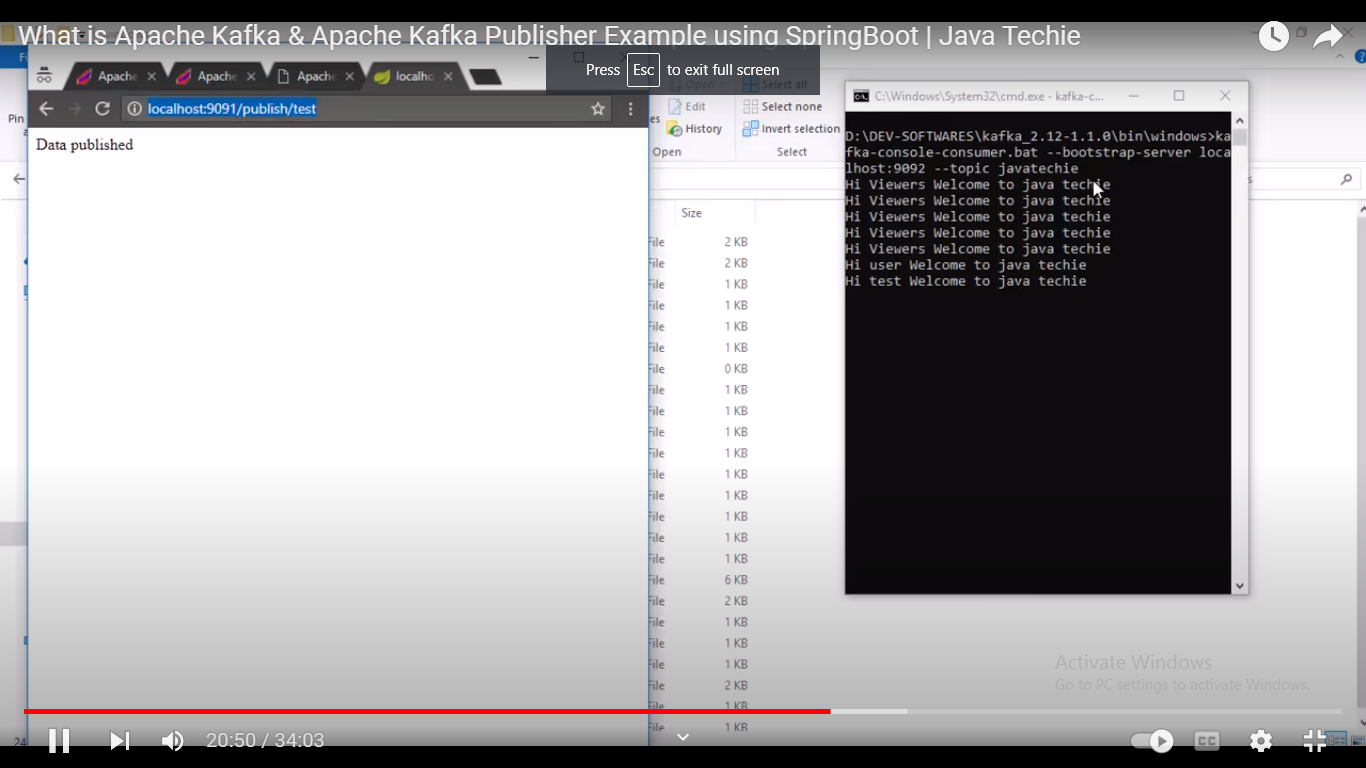
# Consume a message

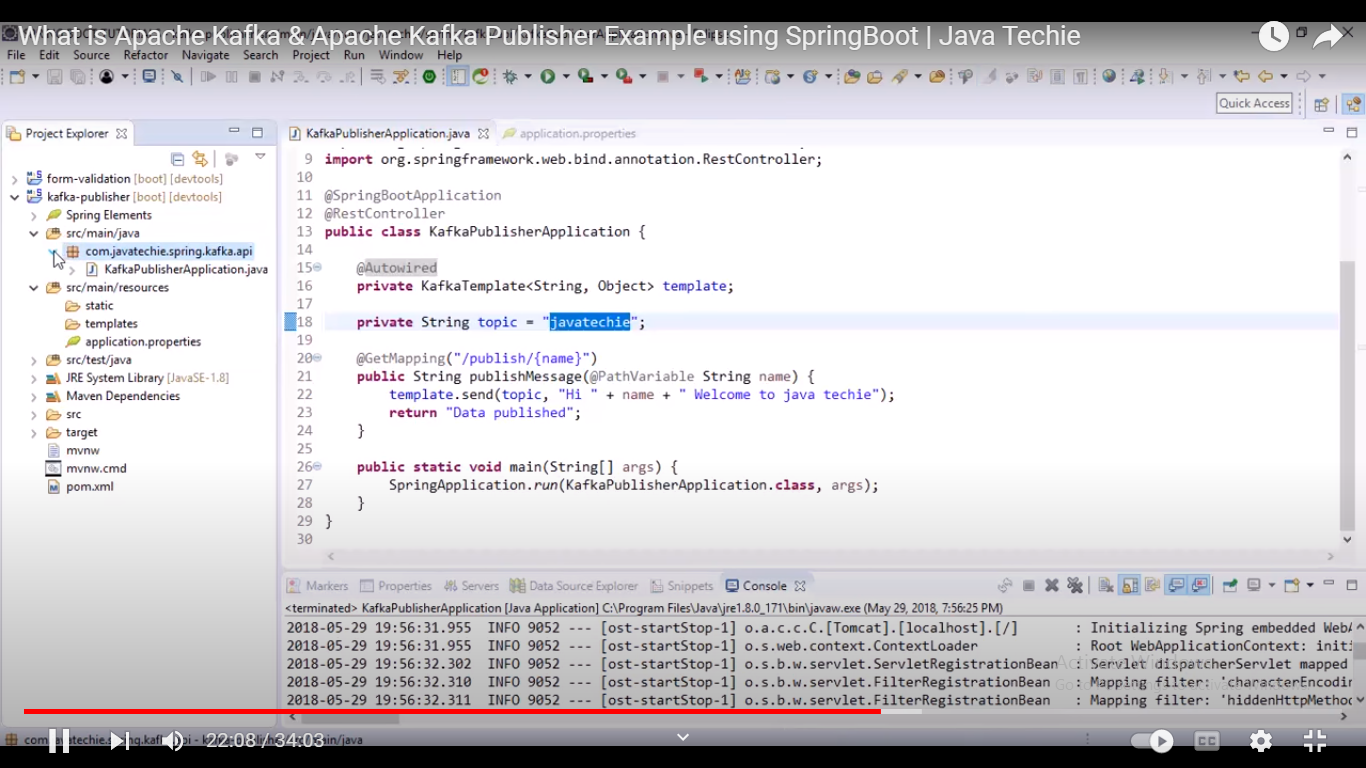
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic javatechie

# kafka-publisher

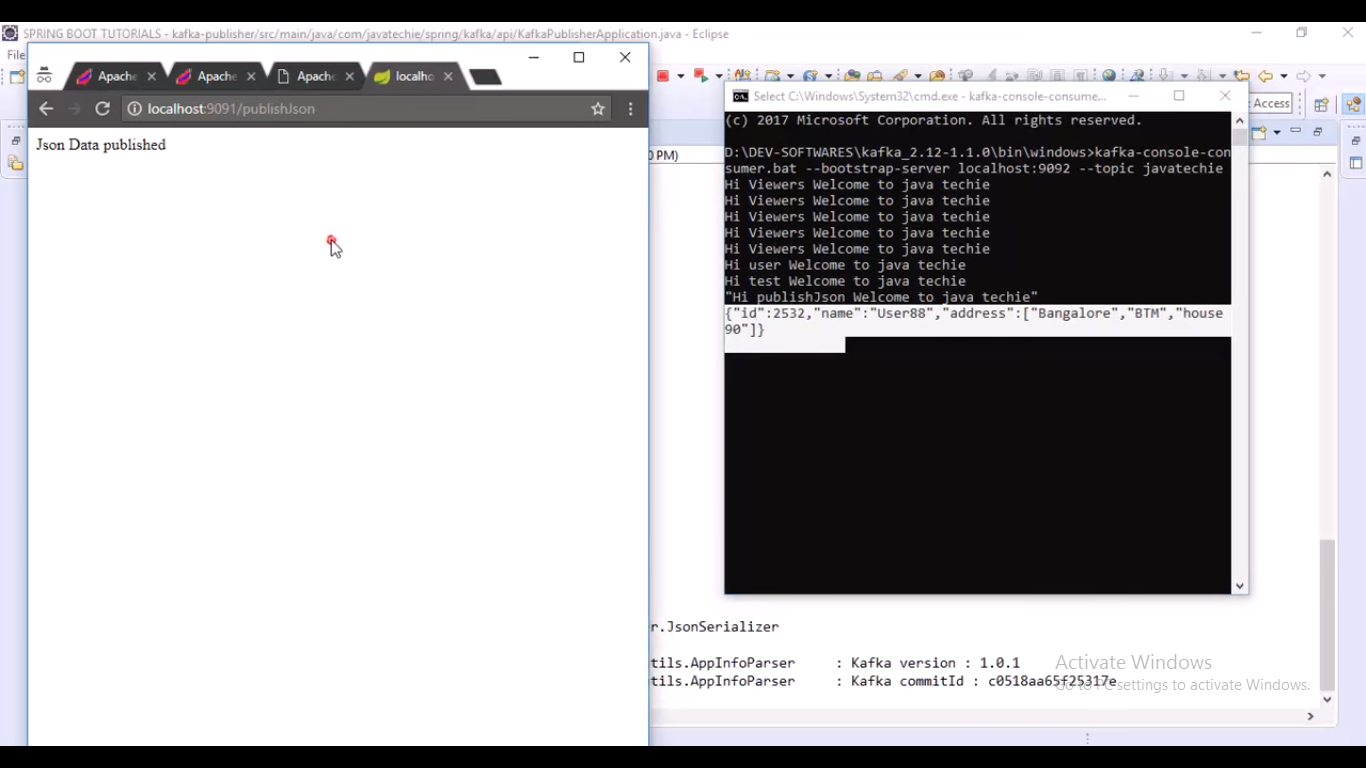












import java.util.HashMap;

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.common.serialization.StringSerializer;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.kafka.core.DefaultKafkaProducerFactory;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.core.ProducerFactory;

import org.springframework.kafka.support.serializer.JsonSerializer;

@Configuration

public class KafkaPublisherConfig {

@Bean

public ProducerFactory<String, Object> producerFactory() {

Map<String, Object> configs = new HashMap<>();

configs.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

configs.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);

configs.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, JsonSerializer.class);

return new DefaultKafkaProducerFactory<String, Object>(configs);

}

@Bean

public KafkaTemplate<String, Object> kafkaTemplate() {

return new KafkaTemplate<>(producerFactory());

}

}

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class User {

private int id;

private String name;

private String[] address;

}

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController

public class KafkaPublisherApplication {

@Autowired

private KafkaTemplate<String, Object> template;

private String topic = "javatechie";

@GetMapping("/publish/{name}")

public String publishMessage(@PathVariable String name) {

template.send(topic, "Hi " + name + " Welcome to java techie");

return "Data published";

}

@GetMapping("/publishJson")

public String publishMessage() {

User user = new User(2532, "User88", new String[] { "Bangalore", "BTM", "house 90" });

template.send(topic, user);

return "Json Data published";

}

public static void main(String[] args) {

SpringApplication.run(KafkaPublisherApplication.class, args);

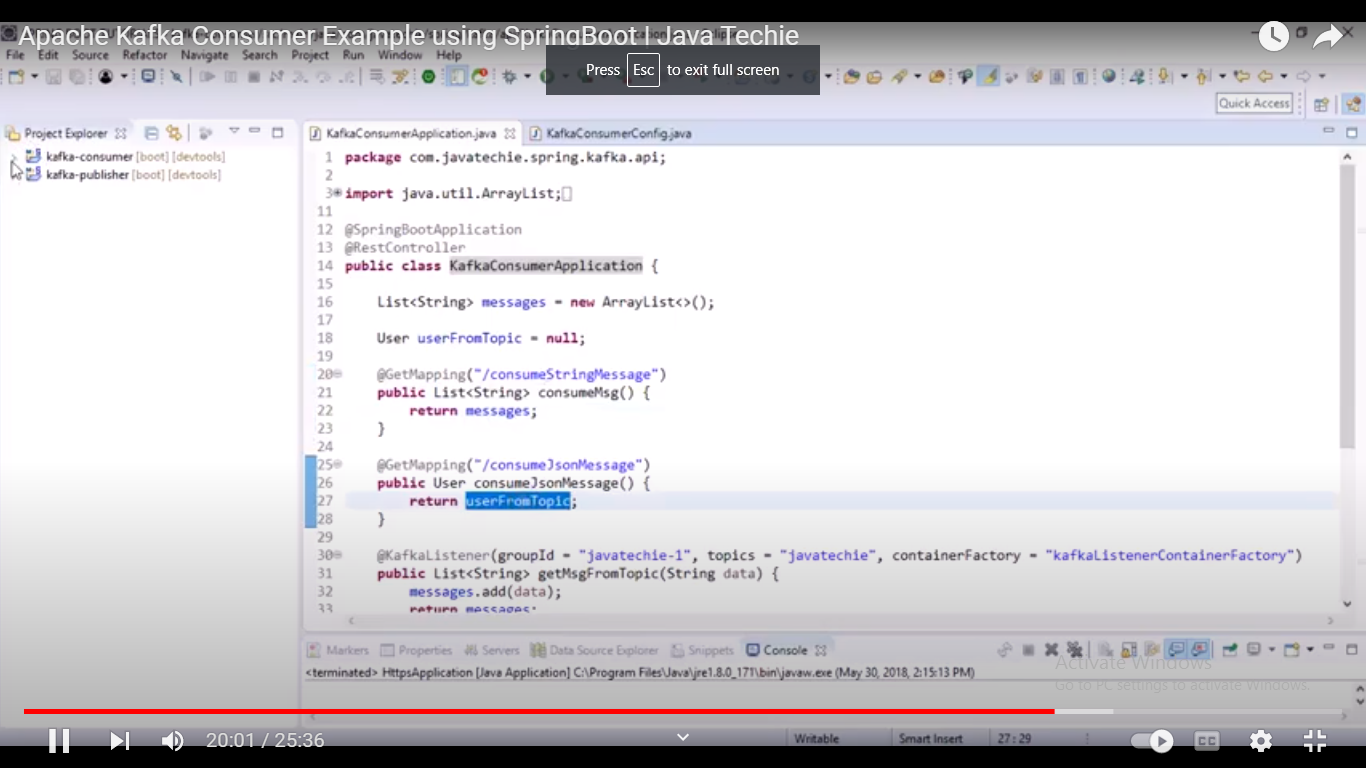
}

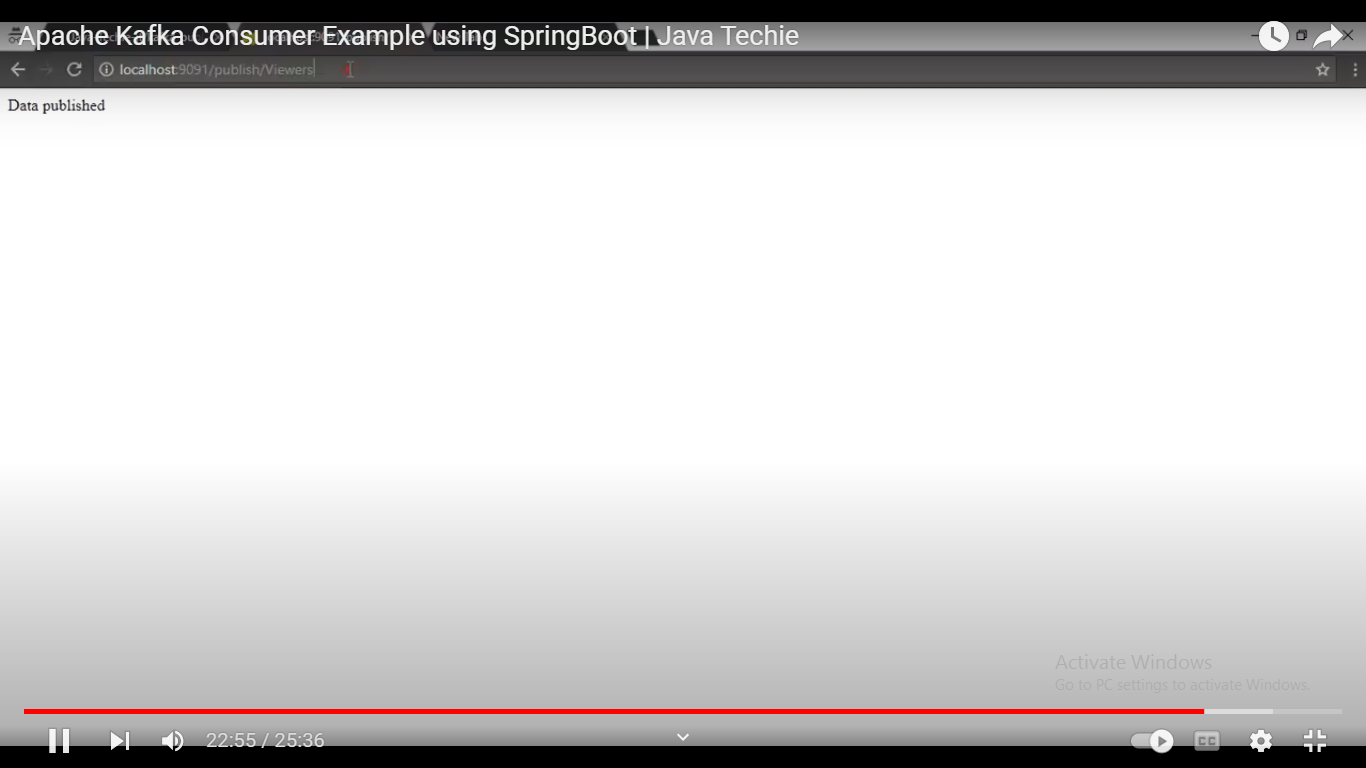
}

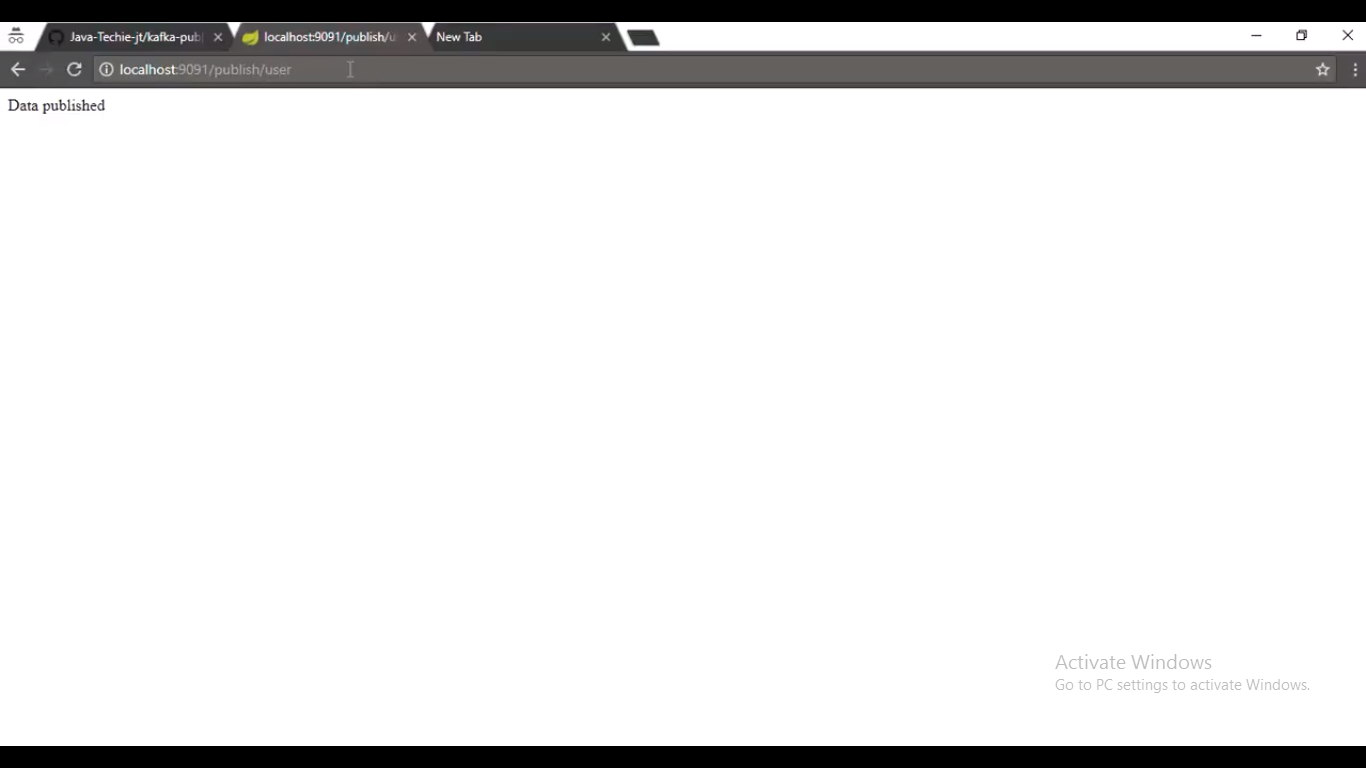
## ****application.properties****

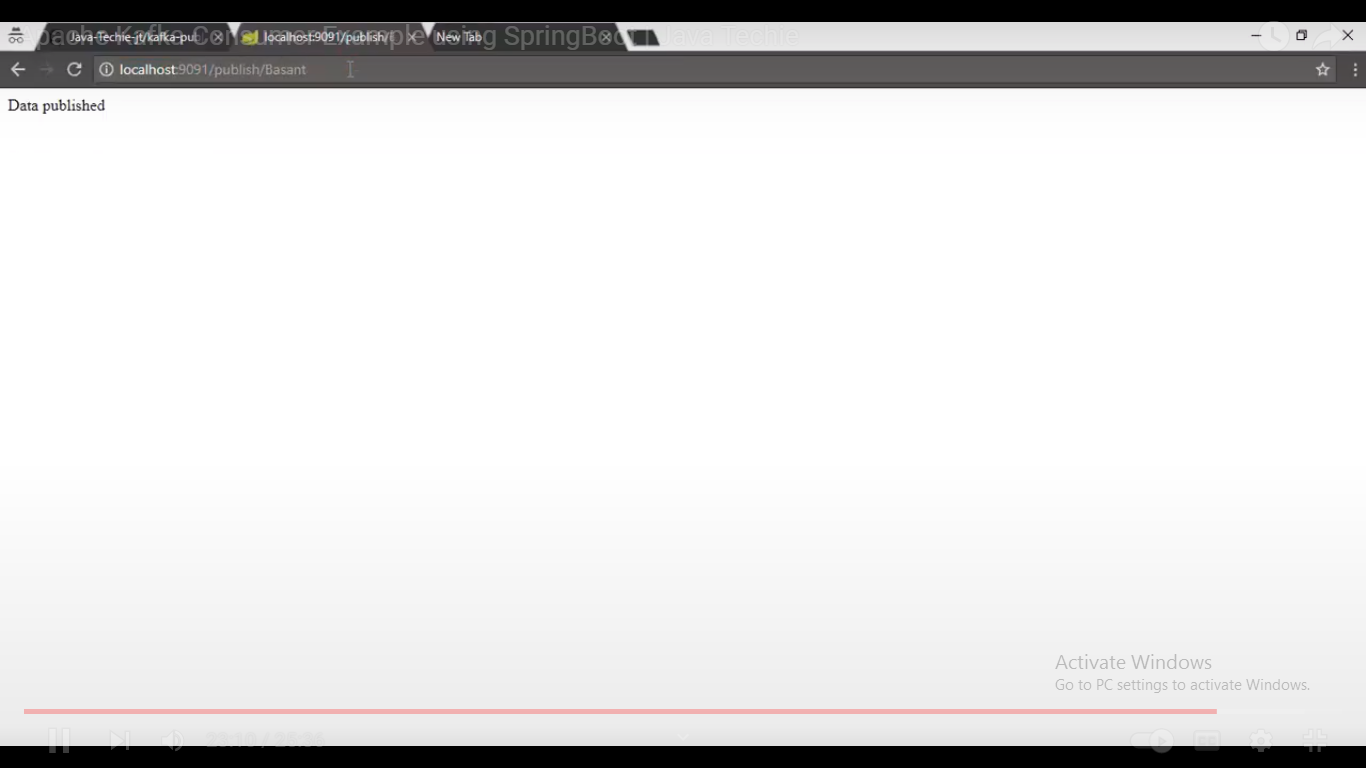
server.port=9091

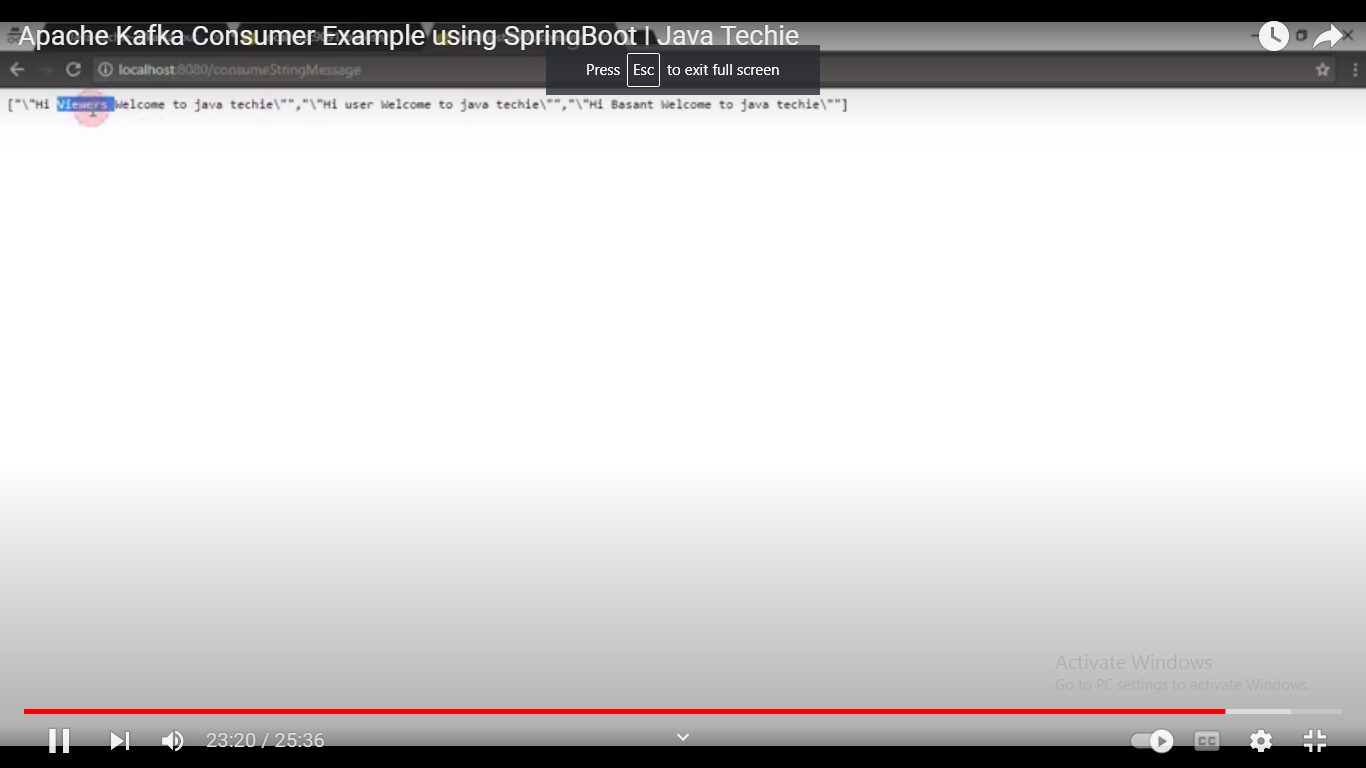
# kafka-consumer

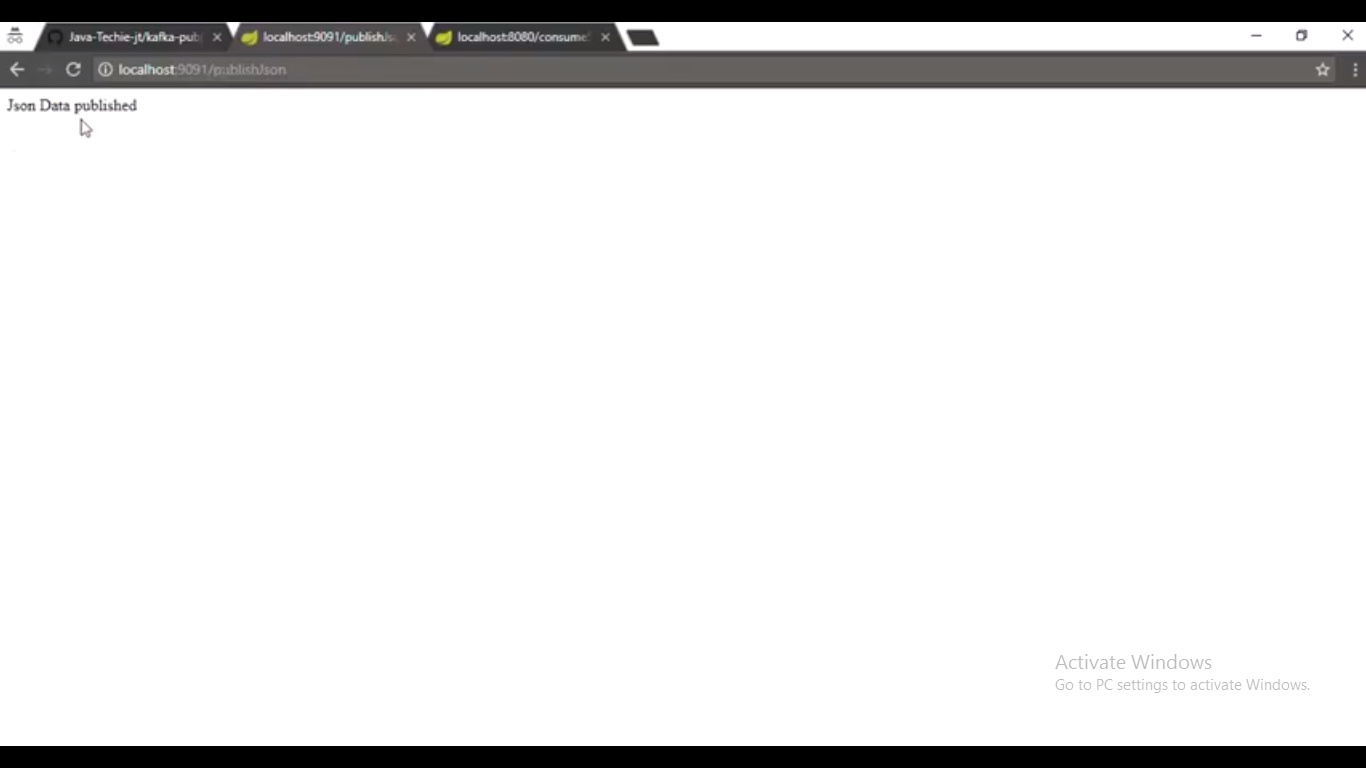


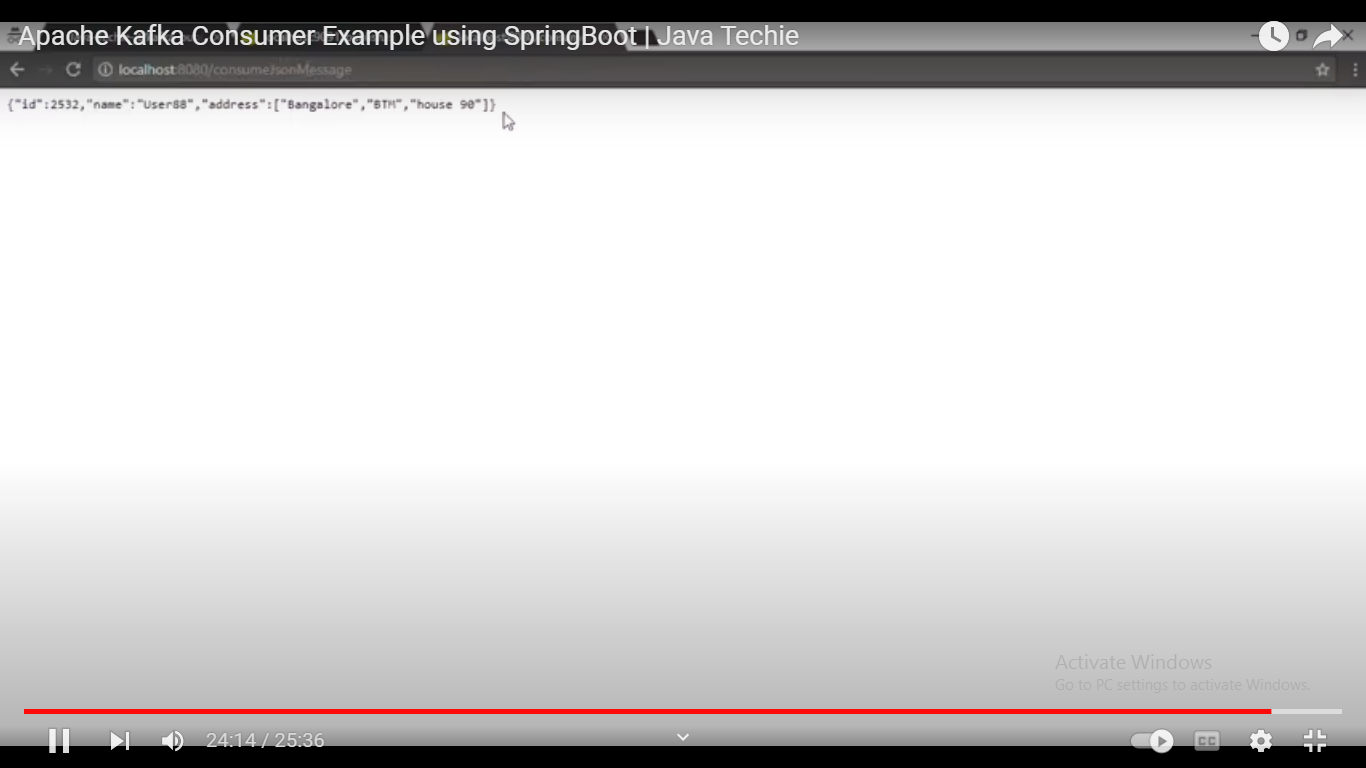


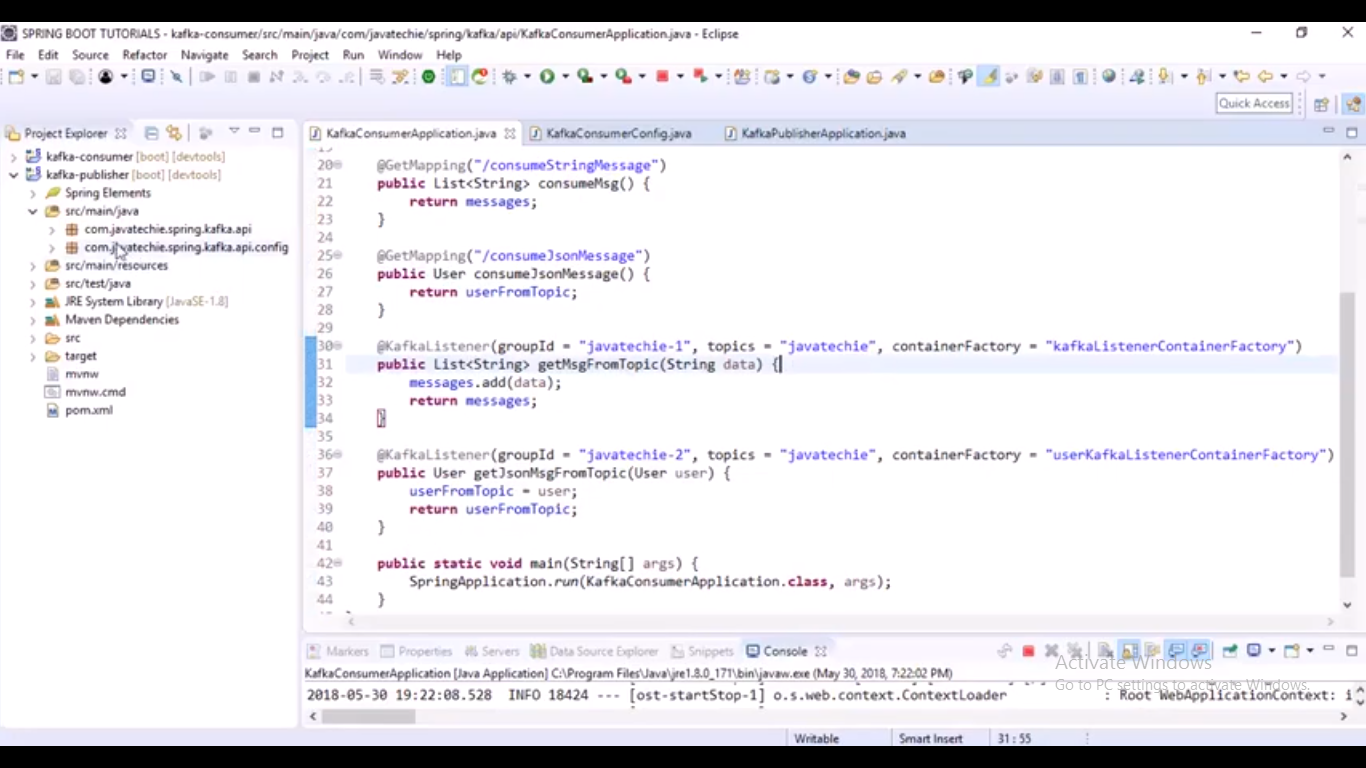












import java.util.HashMap;

import java.util.Map;

import org.apache.kafka.clients.consumer.ConsumerConfig;

import org.apache.kafka.common.serialization.StringDeserializer;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.kafka.annotation.EnableKafka;

import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;

import org.springframework.kafka.core.ConsumerFactory;

import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

import org.springframework.kafka.support.serializer.JsonDeserializer;

import com.javatechie.spring.kafka.api.User;

@Configuration

@EnableKafka

public class KafkaConsumerConfig {

// config for String plain text

@Bean

public ConsumerFactory<String, String> consumerFactory() {

Map<String, Object> configs = new HashMap<>();

configs.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

configs.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

configs.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

configs.put(ConsumerConfig.GROUP\_ID\_CONFIG, "javatechie-1");

return new DefaultKafkaConsumerFactory<>(configs);

}

@Bean

public ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, String> factory = new ConcurrentKafkaListenerContainerFactory<String, String>();

factory.setConsumerFactory(consumerFactory());

return factory;

}

// config for json data

@Bean

public ConsumerFactory<String, User> userConsumerFactory() {

Map<String, Object> configs = new HashMap<>();

configs.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

configs.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

configs.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, JsonDeserializer.class);

configs.put(ConsumerConfig.GROUP\_ID\_CONFIG, "javatechie-2");

return new DefaultKafkaConsumerFactory<>(configs, new StringDeserializer(), new JsonDeserializer<>(User.class));

}

@Bean

public ConcurrentKafkaListenerContainerFactory<String, User> userKafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, User> factory = new ConcurrentKafkaListenerContainerFactory<String, User>();

factory.setConsumerFactory(userConsumerFactory());

return factory;

}

}

import java.util.ArrayList;

import java.util.List;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController

public class KafkaConsumerApplication {

List<String> messages = new ArrayList<>();

User userFromTopic = null;

@GetMapping("/consumeStringMessage")

public List<String> consumeMsg() {

return messages;

}

@GetMapping("/consumeJsonMessage")

public User consumeJsonMessage() {

return userFromTopic;

}

@KafkaListener(groupId = "javatechie-1", topics = "javatechie", containerFactory = "kafkaListenerContainerFactory")

public List<String> getMsgFromTopic(String data) {

messages.add(data);

return messages;

}

@KafkaListener(groupId = "javatechie-2", topics = "javatechie", containerFactory = "userKafkaListenerContainerFactory")

public User getJsonMsgFromTopic(User user) {

userFromTopic = user;

return userFromTopic;

}

public static void main(String[] args) {

SpringApplication.run(KafkaConsumerApplication.class, args);

}

}

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class User {

private int id;

private String name;

private String[] address;

}

**Micro service**

**What is Microservices?**

Microservice Architecture is an architectural development style which builds an application as a collection of small autonomous services developed for a business domain.

* Maintainable and testable
* Loosely coupled
* Independently deployable
* Managed by a small team

# Challenges of Microservices Architecture

* Bounded Context
* Dynamic Scale up and Scale Down
* Monitoring
* Fault Tolerance
* Cyclic dependencies
* DevOps Culture

* .

**What are the advantages of microservices?**

Here, are some significant advantages of using Microservices:

* • Technology diversity, e., Microservices can mix easily with other frameworks, libraries, and databases
* • Fault isolation, e., a process failure should not bring the whole system down.
* • Greater support for smaller and parallel team
* • Independent deployment
* • Deployment time reduce

Maintainable and testable

Loosely coupled

Independently deployable

Managed by a small team

Disadvantages of Microservices

* Complete end-to-end testing is complex.
* There is a higher chance of failure during communication between different services.
* Deployment Challenges.
* The third-party applications are hard to control.
* Difficult to manage a large number of services.
* Microservices has all the associated complexities of the distributed system.
* Pre-planning is essential.
* Complex development.
* Requires a cultural shift.
* The developer needs to solve the problem, such as network latency and load balancing.
* Complex testing over a distributed environment.

Micro service Annotations.

**@EnableEurekaClient**

**@EnableDiscoveryClient**

**@EnableConfigServer**

**@EnableConfigClient**

**@EnableEurekaServer**

**@EnableCircuitBreaker**

**@EnableHystrixDashboard**

**@HystrixCommand(fallbackMethod = "myFallbackMethod")**

**@EnableAdminServer.**

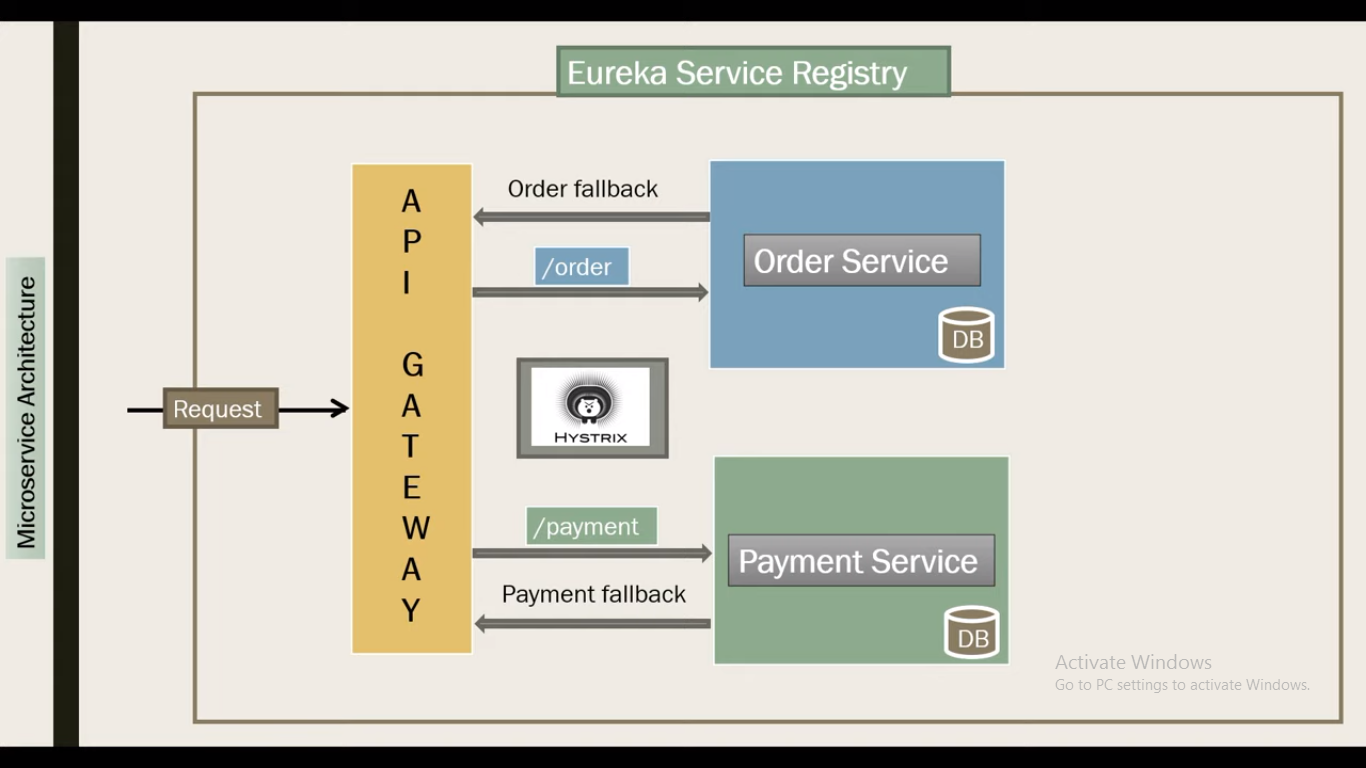
**@LoadBalanced.**

**@EnableFeignClients**

**@FeignClient(name="microservice-1", url="localhost:8000")**

**Explain microservices architecture**

**Explain microservices architecture**



import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "ORDER\_TB")

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Order {

@Id

private int id;

private String name;

private int qty;

private double price;

}

import com.javatechie.os.api.entity.Order;

import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order,Integer> {

}

import com.fasterxml.jackson.core.JsonProcessingException;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.javatechie.os.api.common.Payment;

import com.javatechie.os.api.common.TransactionRequest;

import com.javatechie.os.api.common.TransactionResponse;

import com.javatechie.os.api.entity.Order;

import com.javatechie.os.api.repository.OrderRepository;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.cloud.context.config.annotation.RefreshScope;

import org.springframework.context.annotation.Lazy;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

@RefreshScope

public class OrderService {

Logger logger= LoggerFactory.getLogger(OrderService.class);

@Autowired

private OrderRepository repository;

@Autowired

@Lazy

private RestTemplate template;

@Value("${microservice.payment-service.endpoints.endpoint.uri}")

private String ENDPOINT\_URL;

public TransactionResponse saveOrder(TransactionRequest request) throws JsonProcessingException {

String response = "";

Order order = request.getOrder();

Payment payment = request.getPayment();

payment.setOrderId(order.getId());

payment.setAmount(order.getPrice());

//rest call

logger.info("Order-Service Request : "+new ObjectMapper().writeValueAsString(request));

Payment paymentResponse = template.postForObject(ENDPOINT\_URL, payment, Payment.class);

response = paymentResponse.getPaymentStatus().equals("success") ? "payment processing successful and order placed" : "there is a failure in payment api , order added to cart";

logger.info("Order Service getting Response from Payment-Service : "+new ObjectMapper().writeValueAsString(response));

repository.save(order);

return new TransactionResponse(order, paymentResponse.getAmount(), paymentResponse.getTransactionId(), response);

}

}

import com.javatechie.os.api.common.TransactionResponse;

import com.javatechie.os.api.entity.Order;

import com.javatechie.os.api.service.OrderService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/order")

public class OrderController {

@Autowired

private OrderService service;

@PostMapping("/bookOrder")

public TransactionResponse bookOrder(@RequestBody TransactionRequest request) {

return service.saveOrder(request);

}

}

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Payment {

private int paymentId;

private String paymentStatus;

private String transactionId;

private int orderId;

private double amount;

}

import com.javatechie.os.api.entity.Order;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class TransactionRequest {

private Order order;

private Payment payment;

}

import com.javatechie.os.api.entity.Order;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class TransactionResponse {

private Order order;

private double amount;

private String transactionId;

private String message;

}

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;

@SpringBootApplication

@EnableEurekaClient

public class OrderServiceApplication {

public static void main(String[] args) {

SpringApplication.run(OrderServiceApplication.class, args);

}

@Bean

@LoadBalanced

public RestTemplate restTemplate(){

return new RestTemplate();

}

}

## ****application.yml****

server:

port: 9192

spring:

zipkin:

base-url: http://localhost:9411/

h2:

console:

enabled: true

application:

name: order-service

logging:

file: C:/Users/Basant.Hota/Desktop/oms.log

## ****bootstrap.yml****

spring:

cloud:

config:

uri:

- <http://localhost:9196>

PayMent Micro service

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "PAYMENT\_TB")

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Payment {

@Id

@GeneratedValue

private int paymentId;

private String paymentStatus;

private String transactionId;

private int orderId;

private double amount;

}

import com.javatechie.ps.api.entity.Payment;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PaymentRepository extends JpaRepository<Payment,Integer> {

Payment findByOrderId(int orderId);

}

import com.fasterxml.jackson.core.JsonProcessingException;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.javatechie.ps.api.entity.Payment;

import com.javatechie.ps.api.repository.PaymentRepository;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.Random;

import java.util.UUID;

@Service

public class PaymentService {

@Autowired

private PaymentRepository repository;

Logger logger= LoggerFactory.getLogger(PaymentService.class);

public Payment doPayment(Payment payment) throws JsonProcessingException {

payment.setPaymentStatus(paymentProcessing());

payment.setTransactionId(UUID.randomUUID().toString());

logger.info("Payment-Service Request : {}",new ObjectMapper().writeValueAsString(payment));

return repository.save(payment);

}

public String paymentProcessing(){

//api should be 3rd party payment gateway (paypal,paytm...)

return new Random().nextBoolean()?"success":"false";

}

public Payment findPaymentHistoryByOrderId(int orderId) {

Payment payment=repository.findByOrderId(orderId);

logger.info("paymentService findPaymentHistoryByOrderId : {}",new ObjectMapper().writeValueAsString(payment));

return payment ;

}

}

import com.javatechie.ps.api.entity.Payment;

import com.javatechie.ps.api.service.PaymentService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.Random;

@RestController

@RequestMapping("/payment")

public class PaymentController {

@Autowired

private PaymentService service;

@PostMapping("/doPayment")

public Payment doPayment(@RequestBody Payment payment) {

return service.doPayment(payment);

}

@GetMapping("/{orderId}")

public Payment findPaymentHistoryByOrderId(@PathVariable int orderId){

return service.findPaymentHistoryByOrderId(orderId);

}

}

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class PaymentServiceApplication {

public static void main(String[] args) {

SpringApplication.run(PaymentServiceApplication.class, args);

}

}

## ****application.yml****

server:

port: 9191

spring:

zipkin:

base-url: http://localhost:9411/

h2:

console:

enabled: true

application:

name: payment-service

logging:

file: C:/Users/Basant.Hota/Desktop/oms.log

## ****bootstrap.yml****

spring:

cloud:

config:

uri:

- <http://localhost:9196>

**service-registry**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class ServiceRegistryApplication {

public static void main(String[] args) {

SpringApplication.run(ServiceRegistryApplication.class, args);

}

}

## ****application.yml****

eureka:

client:

register-with-eureka: false

fetch-registry: false

server:

port: 8761

cloud gateway..

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.hystrix.EnableHystrix;

@SpringBootApplication

@EnableEurekaClient

@EnableHystrix

public class CloudGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(CloudGatewayApplication.class, args);

}

}

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import reactor.core.publisher.Mono;

@RestController

public class FallbackController {

@RequestMapping("/orderFallBack")

public Mono<String> orderServiceFallBack() {

return Mono.just("Order Service is taking too long to respond or is down. Please try again later");

}

@RequestMapping("/paymentFallback")

public Mono<String> paymentServiceFallBack() {

return Mono.just("Payment Service is taking too long to respond or is down. Please try again later");

}

}

## ****application.yml****

spring:

application:

name: GATEWAY-SERVICE

cloud:

gateway:

routes:

- id: order-service

uri: lb://ORDER-SERVICE

predicates:

- Path=/order/\*\*

filters:

- name: CircuitBreaker

args:

name: order-service

fallbackuri: forward:/orderFallBack

- id: payment-service

uri: lb://PAYMENT-SERVICE

predicates:

- Path=/payment/\*\*

filters:

- name: CircuitBreaker

args:

name: payment-service

fallbackuri: forward:/paymentFallback

server:

port: 8989

management:

endpoints:

web:

exposure:

include: hystrix.stream

hystrix:

command:

fallbackcmd:

execution:

isolation:

thread:

timeoutInMilliseconds: 5000

## ****bootstrap.yml****

spring:

cloud:

config:

uri:

- <http://localhost:9196>

**hystrix-dashboard**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

@SpringBootApplication

@EnableHystrixDashboard

public class HystrixDashboardApplication {

public static void main(String[] args) {

SpringApplication.run(HystrixDashboardApplication.class, args);

}

}

## ****application.properties****

server.port=9195

# spring-cloud-gateway-hystrix

## API-GateWay

URL : http://localhost:8989/order/bookOrder

HTTP Method : POST

Json Request :

{

"order":{

"id":103,

"name":"Mobile",

"qty":1,

"price":8000

},

"payment":{}

}

Json Response :

{

"order": {

"id": 26,

"name": "ear-phone",

"qty": 5,

"price": 4000

},

"amount": 4000,

"transactionId": "9a021fa6-2061-4332-bdb7-b1358b3430c2",

"message": "payment processing successful and order placed"

}

URL : http://localhost:8989/payment/26

HTTP Method : GET

Json Response :

{

"paymentId": 1,

"transactionId": "d86cfeca-0b26-455e-a1a2-ac3e53707829",

"orderId": 103,

"paymentStatus": "SUCCESS",

"amount":4000

}

## ****CloudConfigServer****

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

@EnableConfigServer

public class CloudConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(CloudConfigServerApplication.class, args);

}

}

## ****application.yml****

spring:

application:

name: CONFIG-SERVER

cloud:

config:

server:

git:

uri: https://github.com/Java-Techie-jt/cloud-config-server

server:

port: 9196

eureka:

client:

register-with-eureka: true

fetch-registry: true

service-url:

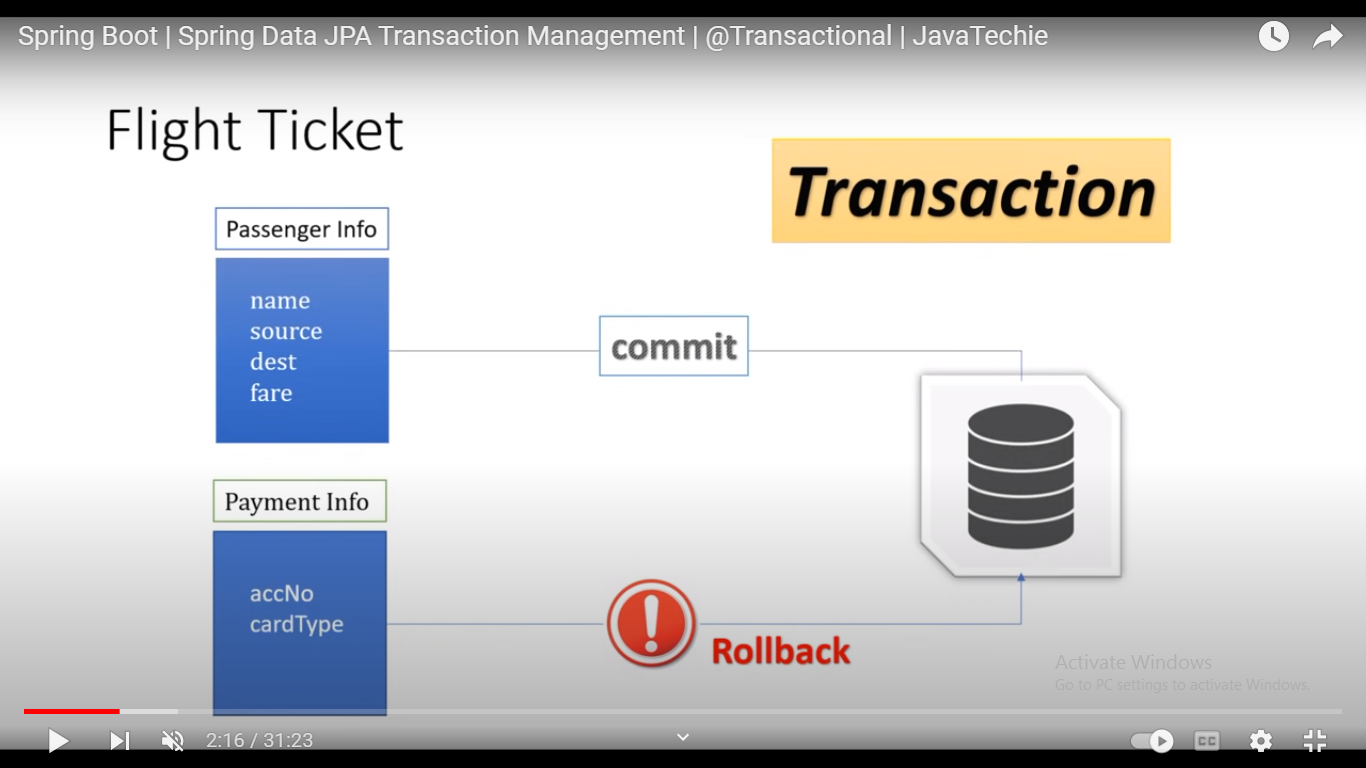
defaultZone: http://localhost:8761/eureka/

instance:

hostname: localhost

# Spring Boot | Spring Data JPA Transaction Management | @Transactional

# spring-transaction-example



import com.fasterxml.jackson.annotation.JsonFormat;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

import java.util.Date;

@Data

@AllArgsConstructor

@NoArgsConstructor

@Entity

@Table(name = "PASSENGER\_INFOS")

public class PassengerInfo {

@Id

@GeneratedValue

private Long pId;

private String name;

private String email;

private String source;

private String Destination;

@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy")

private Date travelDate;

private String pickupTime;

private String arrivalTime;

private double fare;

}

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import org.hibernate.annotations.GenericGenerator;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

@Data

@AllArgsConstructor

@NoArgsConstructor

@Entity

@Table(name = "PAYMENT\_INFO")

public class PaymentInfo {

@Id

@GeneratedValue(generator = "uuid2")

@GenericGenerator(name = "uuid2", strategy = "org.hibernate.id.UUIDGenerator")

private String paymentId;

private String accountNo;

private double amount;

private String cardType;

private Long passengerId;

}

package com.javatechie.tx.repository;

import com.javatechie.tx.entity.PassengerInfo;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PassengerInfoRepository extends JpaRepository<PassengerInfo,Long> {

}

package com.javatechie.tx.repository;

import com.javatechie.tx.entity.PaymentInfo;

import org.springframework.data.jpa.repository.JpaRepository;

public interface PaymentInfoRepository extends JpaRepository<PaymentInfo,String> {

}

package com.javatechie.tx.service;

import com.javatechie.tx.dto.FlightBookingAcknowledgement;

import com.javatechie.tx.dto.FlightBookingRequest;

import com.javatechie.tx.entity.PassengerInfo;

import com.javatechie.tx.entity.PaymentInfo;

import com.javatechie.tx.repository.PassengerInfoRepository;

import com.javatechie.tx.repository.PaymentInfoRepository;

import com.javatechie.tx.utils.PaymentUtils;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Isolation;

import org.springframework.transaction.annotation.Propagation;

import org.springframework.transaction.annotation.Transactional;

import java.util.UUID;

@Service

public class FlightBookingService {

@Autowired

private PassengerInfoRepository passengerInfoRepository;

@Autowired

private PaymentInfoRepository paymentInfoRepository;

@Transactional//(readOnly = false,isolation = Isolation.READ\_COMMITTED,propagation = Propagation.REQUIRED)

public FlightBookingAcknowledgement bookFlightTicket(FlightBookingRequest request) {

PassengerInfo passengerInfo = request.getPassengerInfo();

passengerInfo = passengerInfoRepository.save(passengerInfo);

PaymentInfo paymentInfo = request.getPaymentInfo();

PaymentUtils.validateCreditLimit(paymentInfo.getAccountNo(), passengerInfo.getFare());

paymentInfo.setPassengerId(passengerInfo.getPId());

paymentInfo.setAmount(passengerInfo.getFare());

paymentInfoRepository.save(paymentInfo);

return new FlightBookingAcknowledgement("SUCCESS", passengerInfo.getFare(), UUID.randomUUID().toString().split("-")[0], passengerInfo);

}

}

package com.javatechie.tx.exception;

public class InsufficientAmountException extends RuntimeException {

public InsufficientAmountException(String msg){

super(msg);

}

}

package com.javatechie.tx.utils;

import com.javatechie.tx.exception.InsufficientAmountException;

import java.util.HashMap;

import java.util.Map;

public class PaymentUtils {

private static Map<String, Double> paymentMap = new HashMap<>();

static {

paymentMap.put("acc1", 12000.0);

paymentMap.put("acc2", 10000.0);

paymentMap.put("acc3", 5000.0);

paymentMap.put("acc4", 8000.0);

}

public static boolean validateCreditLimit(String accNo, double paidAmount) {

if (paidAmount > paymentMap.get(accNo)) {

throw new InsufficientAmountException("insufficient fund..!");

} else {

return true;

}

}

}

package com.javatechie.tx;

import com.javatechie.tx.dto.FlightBookingAcknowledgement;

import com.javatechie.tx.dto.FlightBookingRequest;

import com.javatechie.tx.service.FlightBookingService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.transaction.annotation.EnableTransactionManagement;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController

@EnableTransactionManagement

public class FlightServiceExampleApplication {

@Autowired

private FlightBookingService service;

@PostMapping("/bookFlightTicket")

public FlightBookingAcknowledgement bookFlightTicket(@RequestBody FlightBookingRequest request){

return service.bookFlightTicket(request);

}

public static void main(String[] args) {

SpringApplication.run(FlightServiceExampleApplication.class, args);

}

}

package com.javatechie.tx.dto;

import com.javatechie.tx.entity.PassengerInfo;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class FlightBookingAcknowledgement {

private String status;

private double totalFare;

private String pnrNo;

private PassengerInfo passengerInfo;

}

package com.javatechie.tx.dto;

import com.javatechie.tx.entity.PassengerInfo;

import com.javatechie.tx.entity.PaymentInfo;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

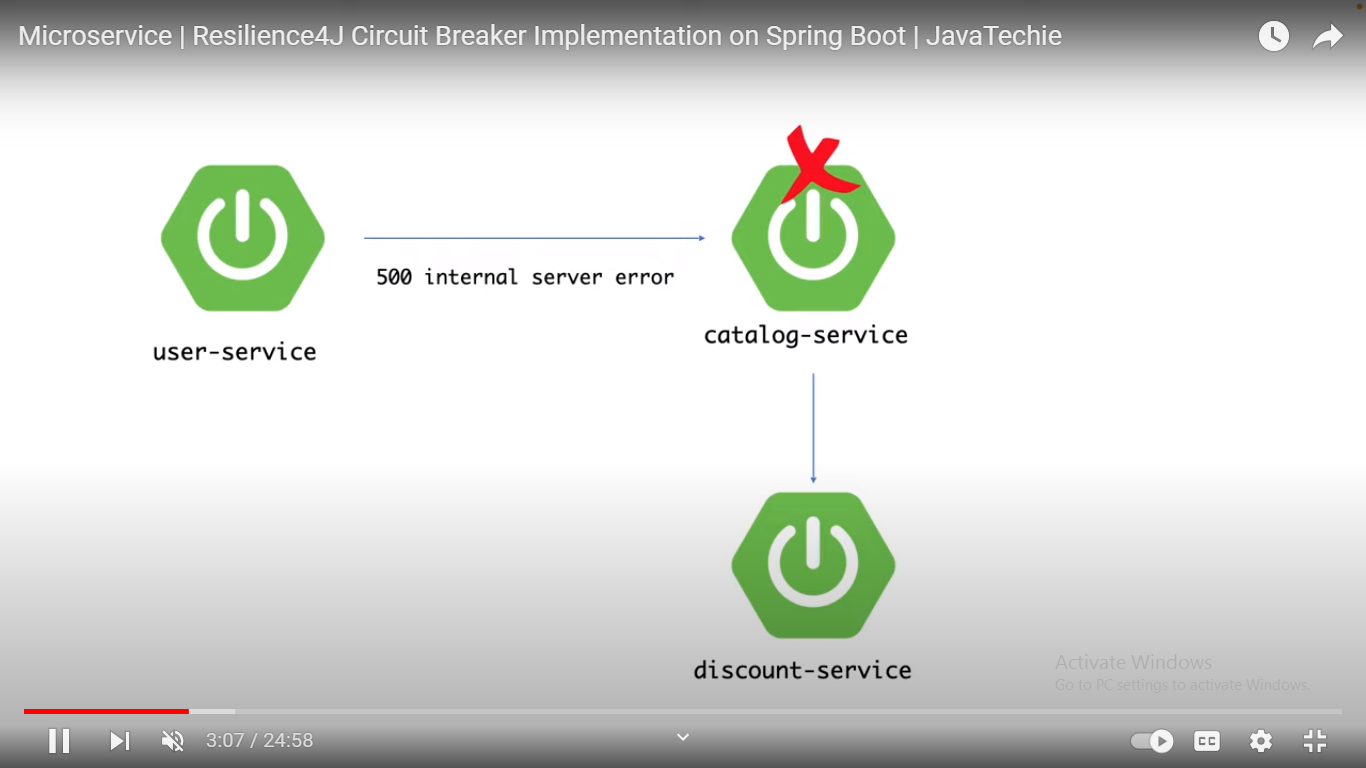
public class FlightBookingRequest {

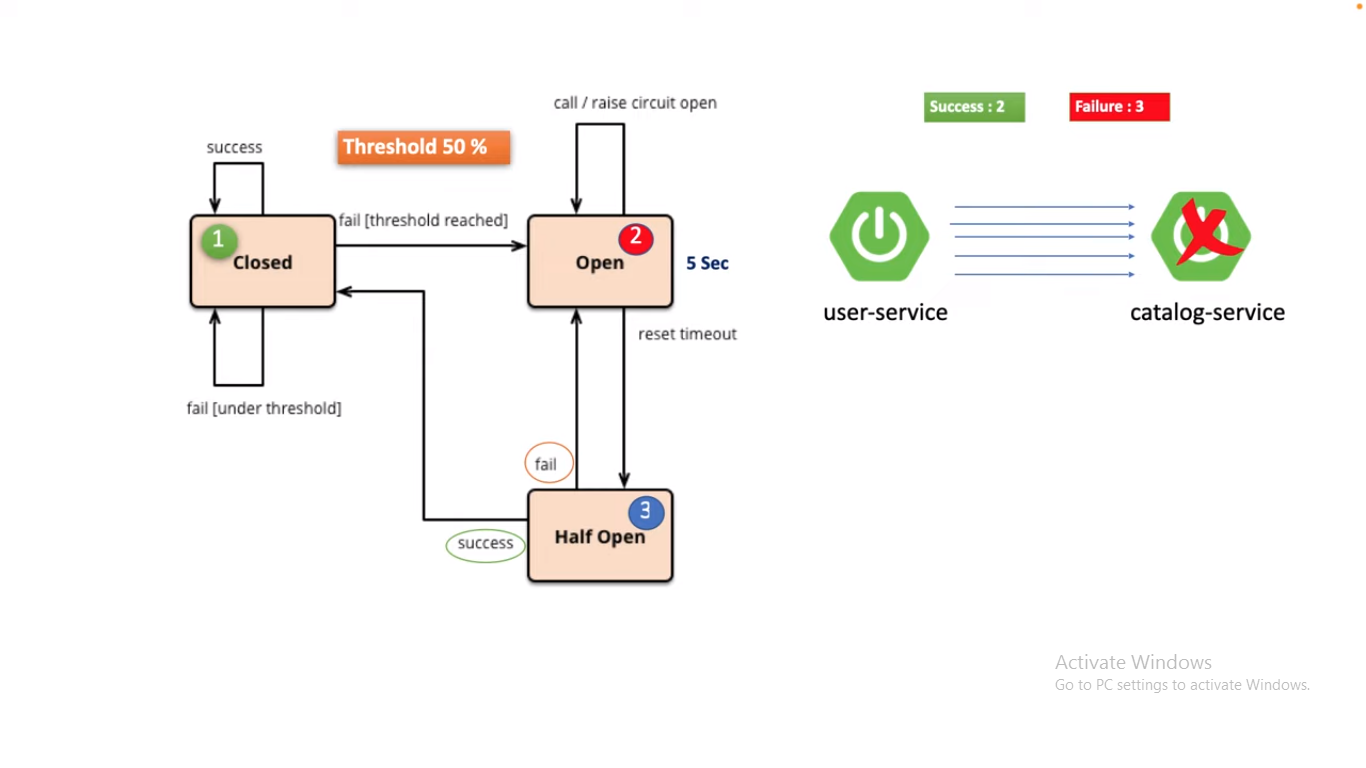
private PassengerInfo passengerInfo;

private PaymentInfo paymentInfo;

}

# Resilience4J Circuit Breaker Implementation on Spring Boot





package com.javatechie.us.dto;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class OrderDTO {

private int id;

private String name;

private String category;

private String color;

private double price;

}

package com.javatechie.us;

import com.javatechie.us.dto.OrderDTO;

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;

import io.github.resilience4j.ratelimiter.annotation.RateLimiter;

import io.github.resilience4j.retry.annotation.Retry;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Lazy;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

@SpringBootApplication

@RestController

@RequestMapping("/user-service")

public class UserServiceApplication {

@Autowired

@Lazy

private RestTemplate restTemplate;

public static final String USER\_SERVICE="userService";

private static final String BASEURL = "http://localhost:9191/orders";

private int attempt=1;

@GetMapping("/displayOrders")

// @CircuitBreaker(name =USER\_SERVICE,fallbackMethod = "getAllAvailableProducts")

@Retry(name = USER\_SERVICE,fallbackMethod = "getAllAvailableProducts")

public List<OrderDTO> displayOrders(@RequestParam("category") String category) {

String url = category == null ? BASEURL : BASEURL + "/" + category;

System.out.println("retry method called "+attempt++ +" times "+" at "+new Date());

return restTemplate.getForObject(url, ArrayList.class);

}

public List<OrderDTO> getAllAvailableProducts(Exception e){

return Stream.of(

new OrderDTO(119, "LED TV", "electronics", "white", 45000),

new OrderDTO(345, "Headset", "electronics", "black", 7000),

new OrderDTO(475, "Sound bar", "electronics", "black", 13000),

new OrderDTO(574, "Puma Shoes", "foot wear", "black & white", 4600),

new OrderDTO(678, "Vegetable chopper", "kitchen", "blue", 999),

new OrderDTO(532, "Oven Gloves", "kitchen", "gray", 745)

).collect(Collectors.toList());

}

public static void main(String[] args) {

SpringApplication.run(UserServiceApplication.class, args);

}

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

Application.properties

server.port=9292

## ****application.yml****

management:

health:

circuitbreakers:

enabled: true

endpoints:

web:

exposure:

include: health

endpoint:

health:

show-details: always

resilience4j:

circuitbreaker:

instances:

userService:

registerHealthIndicator: true

eventConsumerBufferSize: 10

failureRateThreshold: 50

minimumNumberOfCalls: 5

automaticTransitionFromOpenToHalfOpenEnabled: true

waitDurationInOpenState: 5s

permittedNumberOfCallsInHalfOpenState: 3

slidingWindowSize: 10

slidingWindowType: COUNT\_BASED

retry:

instances:

userService:

maxRetryAttempts: 5

waitDuration: 10s

package com.javatechie.os.entity;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "ORDERS\_TBL")

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Order {

@Id

@GeneratedValue

private int id;

private String name;

private String category;

private String color;

private double price;

public Order(String name, String category, String color, double price) {

this.name = name;

this.category = category;

this.color = color;

this.price = price;

}

}

package com.javatechie.os.repository;

import com.javatechie.os.entity.Order;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface OrderRepository extends JpaRepository<Order,Integer> {

List<Order> findByCategory(String category);

}

package com.javatechie.os;

import com.javatechie.os.entity.Order;

import com.javatechie.os.repository.OrderRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.\*;

import javax.annotation.PostConstruct;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

@SpringBootApplication

@RestController

@RequestMapping("/orders")

public class CatalogServiceApplication {

@Autowired

private OrderRepository orderRepository;

@PostConstruct

public void initOrdersTable() {

orderRepository.saveAll(Stream.of(

new Order("mobile", "electronics", "white", 20000),

new Order("T-Shirt", "clothes", "black", 999),

new Order("Jeans", "clothes", "blue", 1999),

new Order("Laptop", "electronics", "gray", 50000),

new Order("digital watch", "electronics", "black", 2500),

new Order("Fan", "electronics", "black", 50000)

).

collect(Collectors.toList()));

}

@GetMapping

public List<Order> getOrders(){

return orderRepository.findAll();

}

@GetMapping("/{category}")

public List<Order> getOrdersByCategory(@PathVariable String category){

return orderRepository.findByCategory(category);

}

public static void main(String[] args) {

SpringApplication.run(CatalogServiceApplication.class, args);

}

}

## ****application.yml****

server:

port: 9191

spring:

datasource:

url: jdbc:h2:mem:testdb

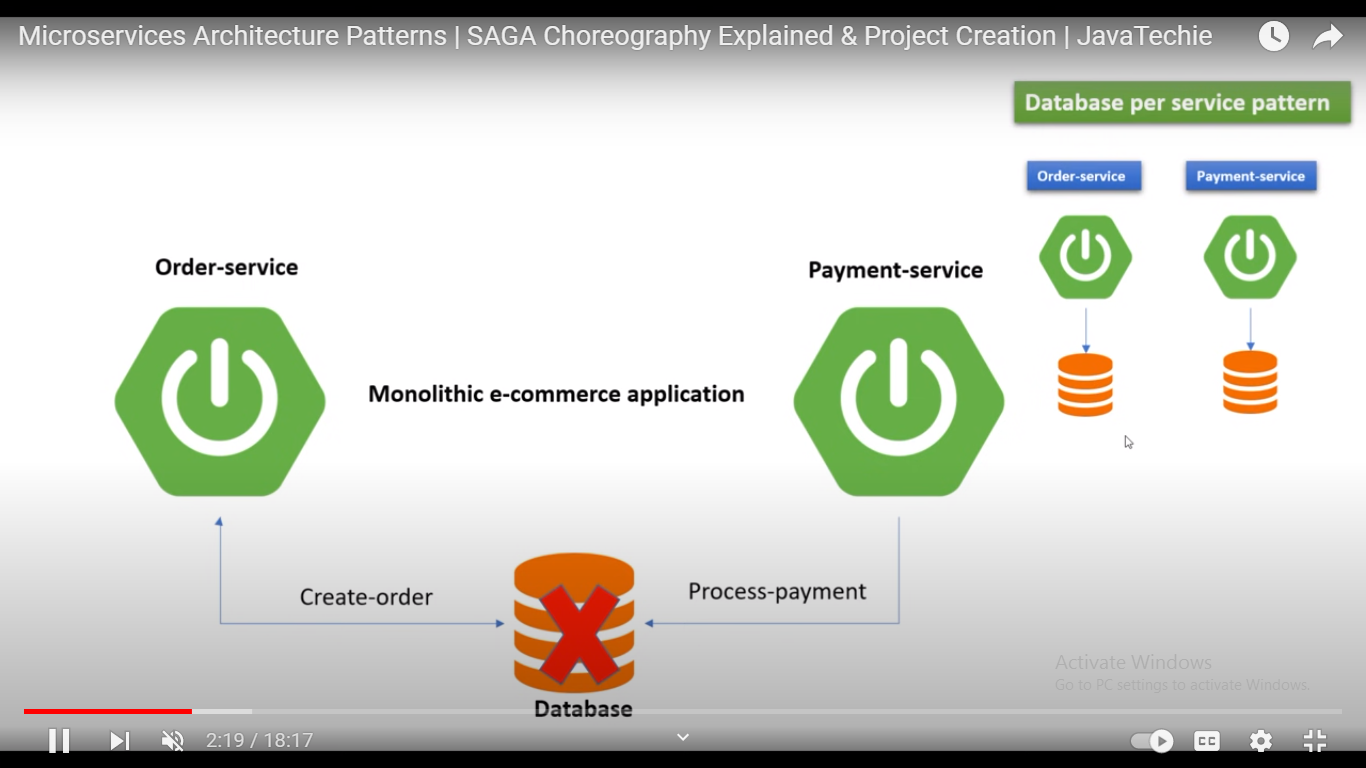
h2:

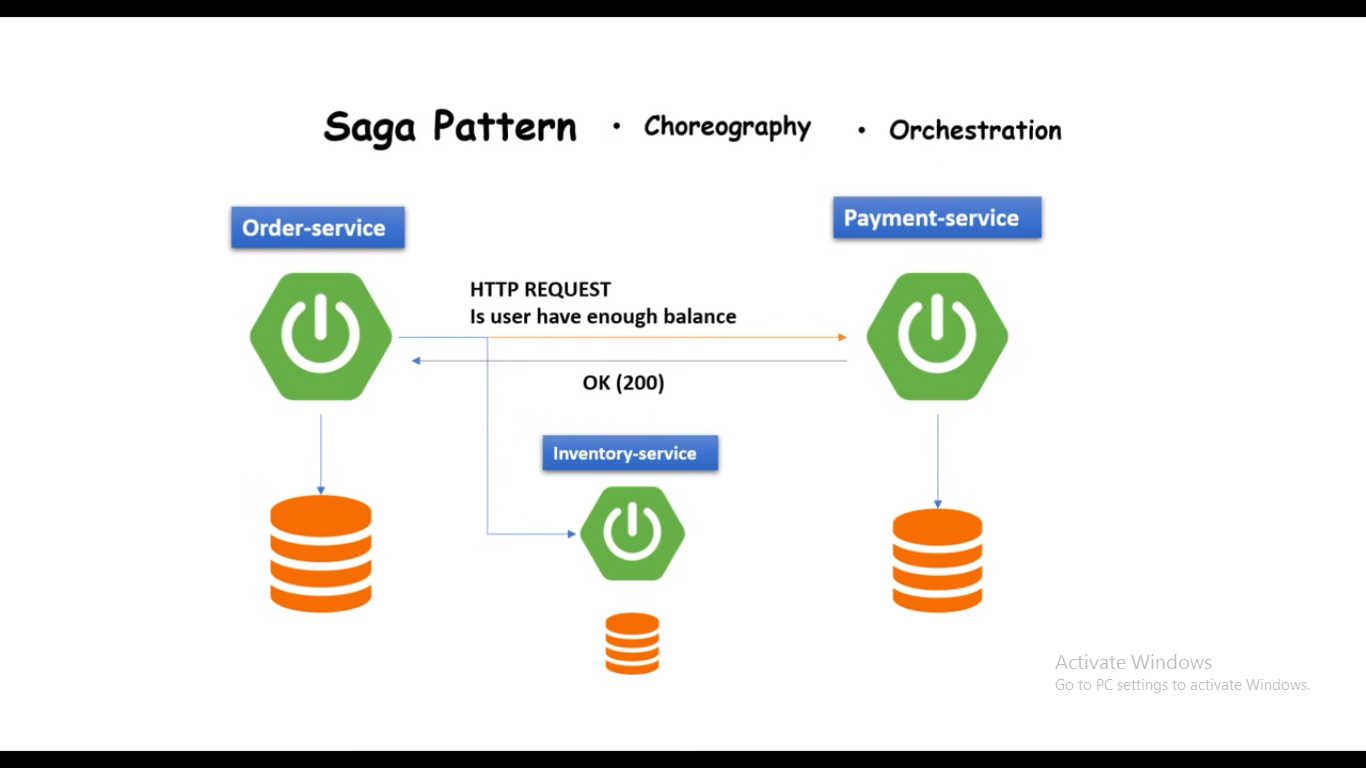
console:

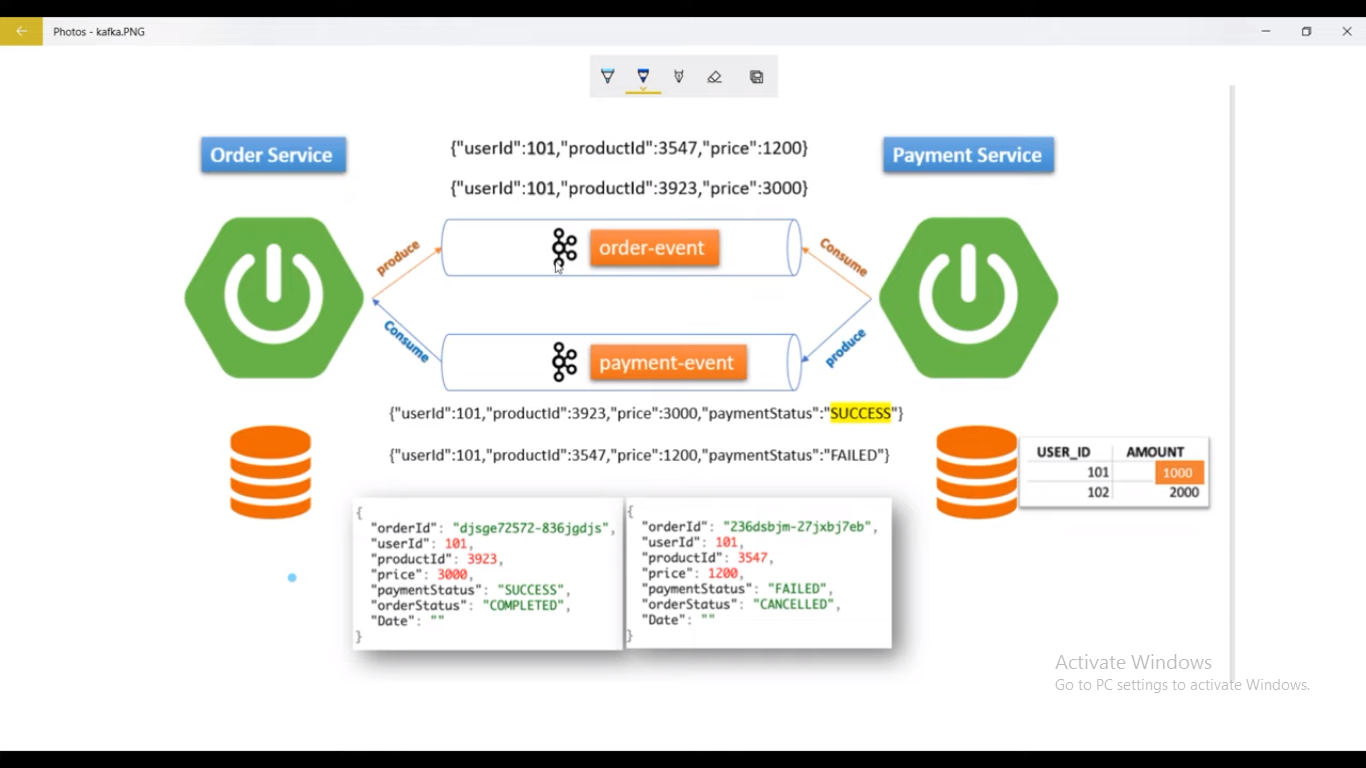
enabled: true

path: /h2

# SAGA Choreography Implementation

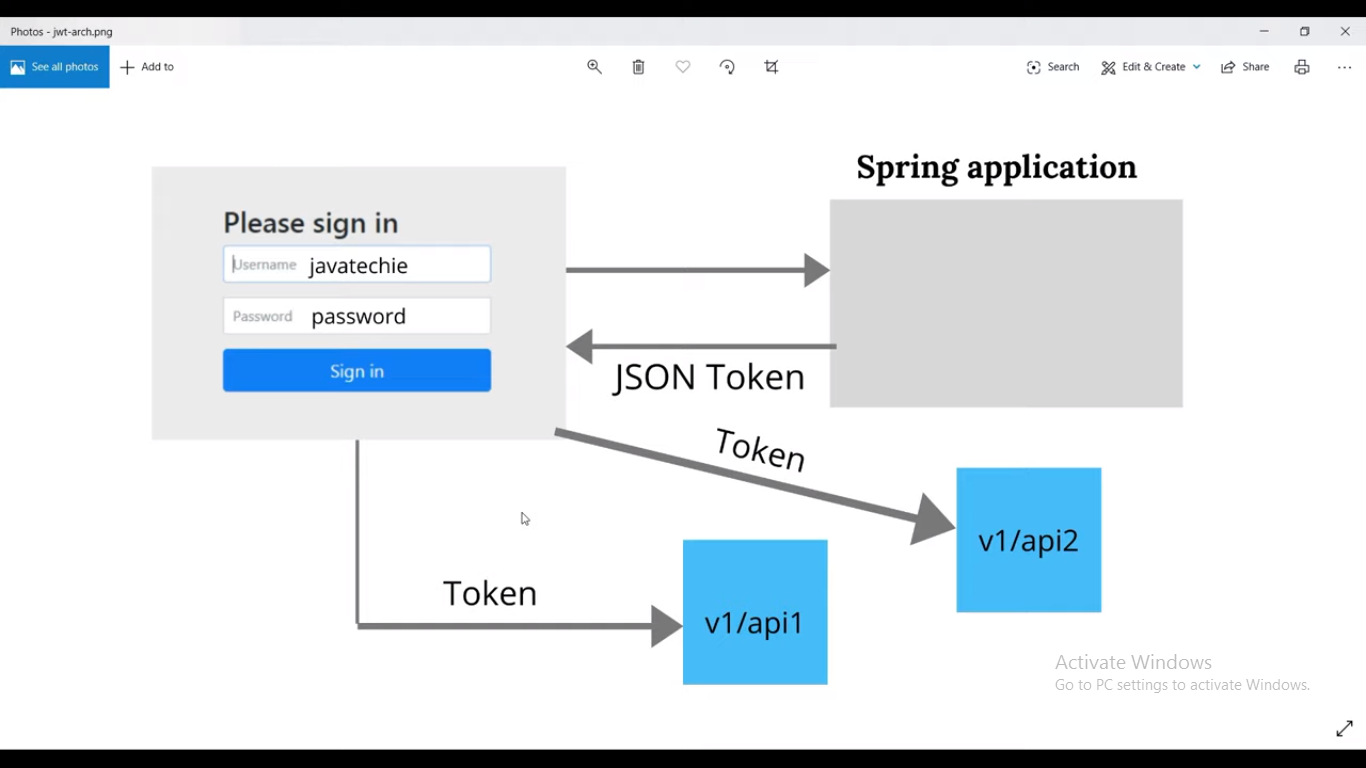


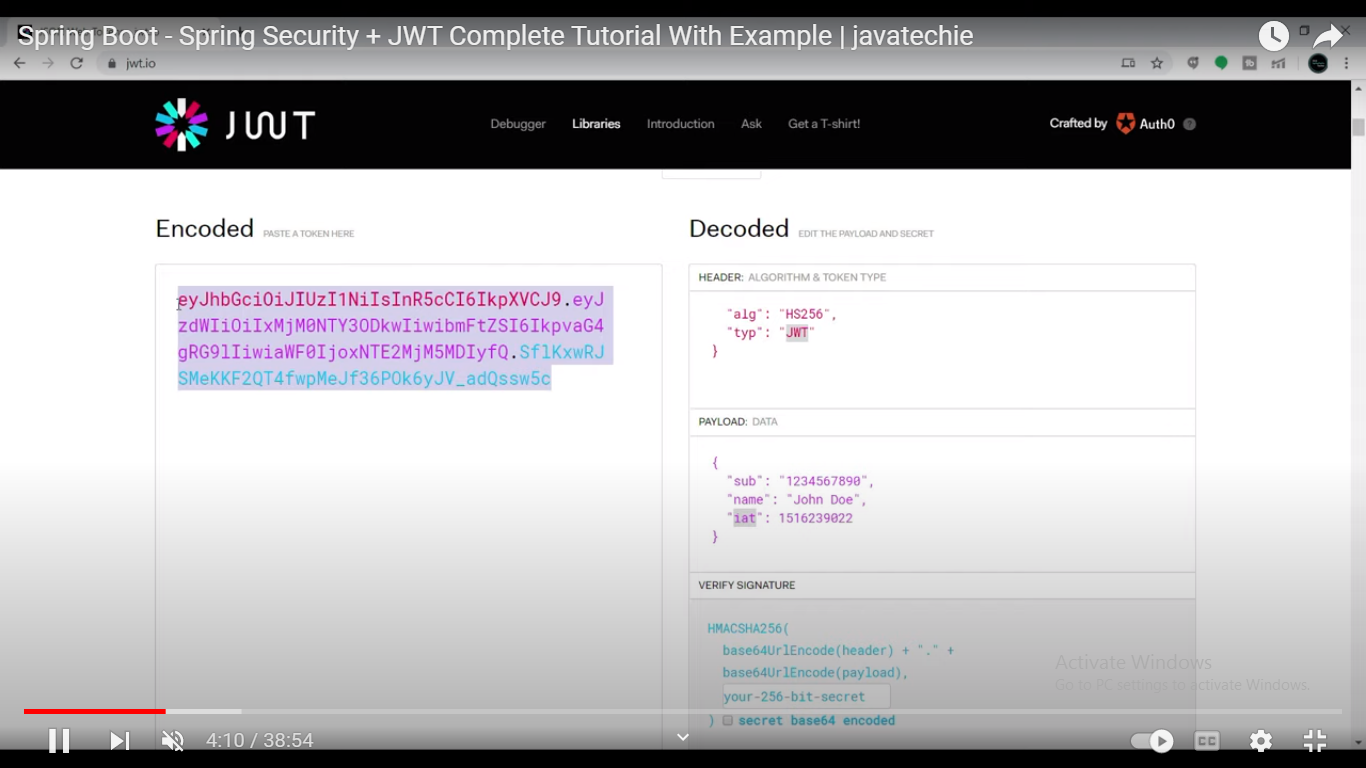




# Spring Boot - Spring Security + JWT Complete Tutorial With Example

[**spring-security-jwt-example**](https://github.com/Java-Techie-jt/spring-security-jwt-example)





package com.javatechie.jwt.api.entity;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Data

@AllArgsConstructor

@NoArgsConstructor

@Entity

@Table(name = "USER\_TBL")

public class User {

@Id

private int id;

private String userName;

private String password;

private String email;

}

package com.javatechie.jwt.api.entity;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class AuthRequest {

private String userName;

private String password;

}

package com.javatechie.jwt.api.repository;

import com.javatechie.jwt.api.entity.User;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User,Integer> {

User findByUserName(String username);

}

package com.javatechie.jwt.api.service;

import com.javatechie.jwt.api.entity.User;

import com.javatechie.jwt.api.repository.UserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

@Service

public class CustomUserDetailsService implements UserDetailsService {

@Autowired

private UserRepository repository;

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

User user = repository.findByUserName(username);

return new org.springframework.security.core.userdetails.User(user.getUserName(), user.getPassword(), new ArrayList<>());

}

}

package com.javatechie.jwt.api.util;

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Service;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

@Service

public class JwtUtil {

private String secret = "javatechie";

public String extractUsername(String token) {

return extractClaim(token, Claims::getSubject);

}

public Date extractExpiration(String token) {

return extractClaim(token, Claims::getExpiration);

}

public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {

final Claims claims = extractAllClaims(token);

return claimsResolver.apply(claims);

}

private Claims extractAllClaims(String token) {

return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

}

private Boolean isTokenExpired(String token) {

return extractExpiration(token).before(new Date());

}

public String generateToken(String username) {

Map<String, Object> claims = new HashMap<>();

return createToken(claims, username);

}

private String createToken(Map<String, Object> claims, String subject) {

return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10))

.signWith(SignatureAlgorithm.HS256, secret).compact();

}

public Boolean validateToken(String token, UserDetails userDetails) {

final String username = extractUsername(token);

return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

}

package com.javatechie.jwt.api.config;

import com.javatechie.jwt.api.filter.JwtFilter;

import com.javatechie.jwt.api.service.CustomUserDetailsService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.BeanIds;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.password.NoOpPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private CustomUserDetailsService userDetailsService;

@Autowired

private JwtFilter jwtFilter;

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService);

}

@Bean

public PasswordEncoder passwordEncoder(){

return NoOpPasswordEncoder.getInstance();

}

@Bean(name = BeanIds.AUTHENTICATION\_MANAGER)

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable().authorizeRequests().antMatchers("/authenticate")

.permitAll().anyRequest().authenticated()

.and().exceptionHandling().and().sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);;

}

}

package com.javatechie.jwt.api.filter;

import com.javatechie.jwt.api.service.CustomUserDetailsService;

import com.javatechie.jwt.api.util.JwtUtil;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@Component

public class JwtFilter extends OncePerRequestFilter {

@Autowired

private JwtUtil jwtUtil;

@Autowired

private CustomUserDetailsService service;

@Override

protected void doFilterInternal(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, FilterChain filterChain) throws ServletException, IOException {

String authorizationHeader = httpServletRequest.getHeader("Authorization");

String token = null;

String userName = null;

if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {

token = authorizationHeader.substring(7);

userName = jwtUtil.extractUsername(token);

}

if (userName != null && SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails = service.loadUserByUsername(userName);

if (jwtUtil.validateToken(token, userDetails)) {

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =

new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

usernamePasswordAuthenticationToken

.setDetails(new WebAuthenticationDetailsSource().buildDetails(httpServletRequest));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);

}

}

filterChain.doFilter(httpServletRequest, httpServletResponse);

}

}

package com.javatechie.jwt.api;

import com.javatechie.jwt.api.entity.User;

import com.javatechie.jwt.api.repository.UserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import javax.annotation.PostConstruct;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

@SpringBootApplication

public class SpringSecurityJwtExampleApplication {

@Autowired

private UserRepository repository;

@PostConstruct

public void initUsers() {

List<User> users = Stream.of(

new User(101, "javatechie", "password", "javatechie@gmail.com"),

new User(102, "user1", "pwd1", "user1@gmail.com"),

new User(103, "user2", "pwd2", "user2@gmail.com"),

new User(104, "user3", "pwd3", "user3@gmail.com")

).collect(Collectors.toList());

repository.saveAll(users);

}

public static void main(String[] args) {

SpringApplication.run(SpringSecurityJwtExampleApplication.class, args);

}

}

## ****application.yml****

spring:

h2:

console:

enabled: true

server:

port: 9192

**What is @pathparam,@requestparam annotation.**

**@PathVariable/@pathparam**

[**http://localhost:8080/springmvc/hello-world/100/Ramesh/**](http://localhost:8080/springmvc/hello-world/100/Ramesh/)

<http://localhost:8080/books/1234>

It is used to pass parameter along with the url, sometimes we need to pass parameters along with the url to get the data.

[**@RequestParam**](https://www.dineshonjava.com/requestparam-annotation-in-spring-mvc-with-example/) annotation, we can extract values from the query string.

[**http://localhost:8080/springmvc/hello-world?id=100&name=Ramesh**](http://localhost:8080/springmvc/hello-world?id=100&name=Ramesh)

the **@RequestParam** annotation is used to read  form data and bind it automatically to the parameter present in the provided method.

It is used to get the request parameters. @RequestParam automatically binds the request parameters to the arguments of your handler method.

http://localhost:8080/books?isbn=1234

**PUT:** it is used to set an entity's information completely. PUTting is similar to POSTing, except that it will overwrite the entity if already exists or create it otherwise.a

**PATCH:** it is used to update an existing entity with new information. You can't patch an entity that doesn't exist. You would use this when you have a simple update to perform, e.g. changing a user's name.

**What are RequestBody and ResponseBody in Rest Services?**

*@RequestBody* and *@ResponseBody* annotations are used to bind the HTTP request/response body with a domain object in method parameter or return type. Behind the scenes, these annotation uses HTTP Message converters to convert the body of HTTP request/response to domain objects.

**@RequestBody Annotation**

Annotation indicating a method parameter should be bound to the body of the web request.

**@ResponseBody Annotation.**

When you use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the HTTP response automatically.

**Spring Boot basic annotations**

In the example application, we have these Spring Boot annotations:

* @Bean - indicates that a method produces a bean to be managed by Spring.
* @Service - indicates that an annotated class is a service class.
* @Repository - indicates that an annotated class is a repository, which is an abstraction of data access and storage.
* @Configuration - indicates that a class is a configuration class that may contain bean definitions.
* @Controller - marks the class as web controller, capable of handling the requests.
* **@ConfigurationProperties(“ containt ”)**
* @RequestMapping - maps HTTP request with a path to a controller method.
* @Autowired - marks a constructor, field, or setter method to be autowired by Spring dependency injection.
* @SpringBootApplication - enables Spring Boot autoconfiguration and component scanning.

@Component is a generic stereotype for a Spring managed component. It turns the class into a Spring bean at the auto-scan time. Classes decorated with this annotation are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.@Repository, @Service, and @Controller are specializations of @Component for more specific use cases.

There are also Hibernate @Entity, @Table, @Id, and @GeneratedValue annotations in the example.

**How spring boot autoconfiguration work in details**

Spring Boot auto-configuration automatically configures the Spring application based on the jar dependencies that we have added.

For example, if the H2 database Jar is present in the classpath and we have not configured any beans related to the database manually.

the Spring Boot's auto-configuration feature automatically configures it in the project.

We can enable the auto-configuration feature by using the annotation **@EnableAutoConfiguration**. But this annotation does not use because it is wrapped inside the **@SpringBootApplication** annotation.

The annotation @SpringBootApplication is the combination of three annotations: **@ComponentScan, @EnableAutoConfiguration,** and **@Configuration**. However, we use @SpringBootApplication annotation instead of using @EnableAutoConfiguration.

**Types of design patterns**

Consequently, based on their purpose, design patterns are divided into three types of patterns **creational**, **structural,** and **behavioral**. Moreover, each of these design patterns has sub-types.

**Creational Design Pattern**

* Singleton Design Pattern
* Factory Pattern
* Absolute factory Pattern
* Builder Pattern
* Prototype Pattern

**Structural Design Pattern**

* Adapter Pattern
* Composite Pattern
* Proxy Pattern
* Flyweight Pattern
* Facade Pattern
* Bridge Pattern
* Decorator Pattern

**Behavioral Design Pattern**

* Template Method Pattern
* Mediator Pattern
* Chain of responsibility Pattern
* Observer Pattern
* Strategy Pattern
* Command Pattern
* State Pattern
* Visitor Pattern
* Interpreter Pattern
* Iterator Pattern
* Memento Pattern

**Spring Batch..**

**What is Spring Batch?**

**Spring Batch is a lightweight framework designed to facilitate batch processing**. It allows developers to create batch applications. In turn, these batch applications process the incoming data and transform it for further usage.

**What is Batch Processing?**

[Batch processing](https://betterjavacode.com/programming/handling-large-datasets-in-distributed-systems) is a data processing mode. It involves consuming all the data, processing that data, transforming it, and then sending it to another data source.

**Spring Batch Framework**

The following architecture shows the components of the Spring Batch framework.

Its convert csv file to database and  generate the report  for database to csv file.

**Write code for OneToMany and ManyToMany relationship**

**In the Category side:**

**private** Set<Product> products;

@OneToMany(mappedBy = "category", cascade = CascadeType.ALL)

**public** Set<Product> getProducts() {

**return** products;

}

**In the Product side:**

**private** Category category;

@ManyToOne

@JoinColumn(name = "CATEGORY\_ID")

**public** Category getCategory() {

**return** category;

}

-------------------------------------------------------------------------------

@Entity

@Table(name = "CATEGORY")

**public** **class** Category {

**private** **long** id;

**private** String name;

**private** Set<Product> products;

**public** Category() {

    }

**public** Category(String name) {

**this**.name = name;

    }

    @Id

    @Column(name = "CATEGORY\_ID")

    @GeneratedValue

**public** **long** getId() {

**return** id;

    }

    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL)

**public** Set<Product> getProducts() {

**return** products;

    }

    // other getters and setters...

}

@Entity

@Table(name = "PRODUCT")

**public** **class** Product {

**private** **long** id;

**private** String name;

**private** String description;

**private** **float** price;

**private** Category category;

**public** Product() {

    }

**public** Product(String name, String description, **float** price,

            Category category) {

**this**.name = name;

**this**.description = description;

**this**.price = price;

**this**.category = category;

    }

    @Id

    @Column(name = "PRODUCT\_ID")

    @GeneratedValue

**public** **long** getId() {

**return** id;

    }

    @ManyToOne

    @JoinColumn(name = "CATEGORY\_ID")

**public** Category getCategory() {

**return** category;

    }

    // other getters and setters...

}

**ManytoMany Association mapping.**

@Entity

@Table(name = "USERS")

**public** **class** User {

**private** **long** id;

**private** String username;

**private** String password;

**private** String email;

**private** Set<UserGroup> userGroups = **new** HashSet<UserGroup>();

    @OneToMany(mappedBy = "user")

**public** Set<UserGroup> getUserGroups() {

**return** userGroups;

    }

----------------------------------------------------------------------------------------

    @Id

    @GeneratedValue

    @Column(name = "USER\_ID")

**public** **long** getId() {

**return** id;

    }

    }

@Entity

@Table(name = "GROUPS")

**public** **class** Group {

**private** **long** id;

**private** String name;

**private** Set<UserGroup> userGroups = **new** HashSet<UserGroup>();

    @OneToMany(mappedBy = "group")

**public** Set<UserGroup> getUserGroups() {

**return** userGroups;

    }

    @Id

    @GeneratedValue

    @Column(name = "GROUP\_ID")

**public** **long** getId() {

**return** id;

    }

    }

**----------------------------------------------------------------------------------------------**

@Entity

@Table(name = "USERS\_GROUPS")

**public** **class** UserGroup {

**private** **long** id;

**private** User user;

**private** Group group;

    // additional fields

**private** **boolean** activated;

**private** Date registeredDate;

    @Id

    @GeneratedValue

    @Column(name = "USER\_GROUP\_ID")

**public** **long** getId() {

**return** id;

    }

    @ManyToOne(cascade = CascadeType.ALL)

    @JoinColumn(name = "USER\_ID")

**public** User getUser() {

**return** user;

    }

    @ManyToOne(cascade = CascadeType.ALL)

    @JoinColumn(name = "GROUP\_ID")

**public** Group getGroup() {

**return** group;

    }

    }

**SPRING, SPRINGBOOT AND MICROSERVICES::**

**Multiple Database Configurations in Spring Boot**

Hibernate.property file.

**spring.user.datasource.url=jdbc:mysql://localhost:3306/user**   
**spring.user.datasource.username=root**   
**spring.user.datasource.password=root**   
**spring.user.datasource.driver-class-name=com.mysql.jdbc.Driver**   
   
**spring.booking.datasource.url=jdbc:mysql://localhost:3306/booking**   
**spring.booking.datasource.username=root**   
**spring.booking.datasource.password=root**   
**spring.booking.datasource.driver-class-name=com.mysql.jdbc.Driver**

Configuring Booking Database

Following is the configuration to connect to booking database. We have configured the entitymanager required to query the booking DB as per JPA.@ConfigurationProperties(prefix = "spring.booking.datasource"). This will ensure that spring picks properties starting with spring.booking.datasource to create the datasource and utilise it while executing methods of BookingDao.java.basePackages = "com.devglan.booking.dao" will ensure that spring uses booking datasource while executing methods of BookingDao.java @Primary It tells spring to use this bean to use as a primary bean as we have multiple beans for same return type. To use other beans of same return type we require to use @Qualifier annotation.**BookingDBConfig.java**

Following is the configuration to connect to booking database. We have configured the entitymanager required to query the booking DB as per JPA.

@ConfigurationProperties(prefix = "spring.booking.datasource"). This will ensure that spring picks properties starting with spring.booking.datasource to create the datasource and utilise it while executing methods of BookingDao.java.

basePackages = "com.devglan.booking.dao" will ensure that spring uses booking datasource while executing methods of BookingDao.java

@PrimaryIt tells spring to use this bean to use as a primary bean as we have multiple beans for same return type. To use other beans of same return type we require to use @Qualifier annotation.

**Ther are three methoed are required.**

mysqlDataSource()

MysqlEntityManagerFactory()

MysqlTransactionManager()

**BookingDBConfig.java**

@Configuration   
@EnableTransactionManagement   
@EnableJpaRepositories(   
entityManagerFactoryRef = "bookingEntityManager",    
transactionManagerRef = "bookingTransactionManager",    
basePackages = "com.devglan.booking.dao"   
)   
public class BookingDBConfig {   
   
@Primary   
@Bean   
@ConfigurationProperties(prefix = "spring.booking.datasource")   
public DataSource mysqlDataSource() {   
return DataSourceBuilder   
.create()   
.build();   
}   
   
@Primary   
@Bean(name = "bookingEntityManager")   
public LocalContainerEntityManagerFactoryBean mysqlEntityManagerFactory(EntityManagerFactoryBuilder builder) {   
return builder   
.dataSource(mysqlDataSource())   
.properties(hibernateProperties())   
.packages(Booking.class)   
.persistenceUnit("bookingPU")   
.build();   
}   
   
@Primary   
@Bean(name = "bookingTransactionManager")   
public PlatformTransactionManager mysqlTransactionManager(@Qualifier("bookingEntityManager") EntityManagerFactory entityManagerFactory) {   
return new JpaTransactionManager(entityManagerFactory);   
}   
   
private Map hibernateProperties() {   
   
Resource resource = new ClassPathResource("hibernate.properties");   
   
try {   
Properties properties = PropertiesLoaderUtils.loadProperties(resource);   
   
return properties.entrySet().stream()   
.collect(Collectors.toMap( 

e -> e.getKey().toString(),   
e -> e.getValue())   
);   
} catch (IOException e) {   
return new HashMap();   
}   
}   
}

Configuring User DataBase

**Ther are three methoed are required.**

postgresqlDataSource()

postgresqlEntityManagerFactory()

postgresqlTransactionManager()

BookingDBConfig.java.

@Configuration   
@EnableTransactionManagement   
@EnableJpaRepositories(   
entityManagerFactoryRef = "userEntityManager",    
transactionManagerRef = "userTransactionManager",    
basePackages = "com.devglan.user.dao"   
)   
public class UserDBConfig {   
   
@Bean   
@ConfigurationProperties(prefix = "spring.user.datasource")   
public DataSource postgresqlDataSource() {   
return DataSourceBuilder   
.create()   
.build();   
}   
   
@Bean(name = "userEntityManager")   
public LocalContainerEntityManagerFactoryBean postgresqlEntityManagerFactory(EntityManagerFactoryBuilder builder) {   
return builder   
.dataSource(postgresqlDataSource())   
.properties(hibernateProperties())   
.packages(UserDetails.class)   
.persistenceUnit("userPU")   
.build();   
}   
   
@Bean(name = "userTransactionManager")   
public PlatformTransactionManager postgresqlTransactionManager(@Qualifier("userEntityManager") EntityManagerFactory entityManagerFactory) {   
return new JpaTransactionManager(entityManagerFactory);   
}   
   
private Map hibernateProperties() {   
   
Resource resource = new ClassPathResource("hibernate.properties");   
   
try {   
Properties properties = PropertiesLoaderUtils.loadProperties(resource);   
   
return properties.entrySet().stream()   
.collect(Collectors.toMap( 

e -> e.getKey().toString(),   
e -> e.getValue())   
);   
} catch (IOException e) {   
return new HashMap();   
}   
}   
} 

Difference between @RequestMapping() and @RequestMapping(Methpod= GET)?

@RequestMapping is a class level

@GetMapping is a method-level

**RequestMapping** can be used at class level: This annotation can be used both at the class and at the method level. while **GetMapping** only applies to method: Annotation for mapping HTTP GET requests onto specific handler methods.

**@Component Annotation**

@Component is a class-level annotation. It is used to denote a class as a Component. We can use @Component across the application to mark the beans as Spring’s managed components. A component is responsible for some operations. Spring framework provides three other specific annotations to be used when marking a class as a Component.

1. @Service
2. @Repository
3. @Controller

**1: @Service:** We specify a class with @Service to indicate that they’re holding the business logic. Besides being used in the service layer, there isn’t any other special use for this annotation. The utility classes can be marked as Service classes.

**2: @Repository:** We specify a class with @Repository to indicate that they’re dealing with **CRUD operations**, usually, it’s used with DAO (Data Access Object) or Repository implementations that deal with database tables.

**3: @Controller:** We specify a class with @Controller to indicate that they’re front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.

What are stereotype annotations? These annotations are used to create Spring beans automatically in the application context by scanning for anything of type @Component. Once Spring has your beans in the application context via its classpath @Component scanning, it can perform autowiring for you. Essentially, if your annotation is of type @Component, it gets picked up.

Our applications may be using many stereotype annotations to identify different boundaries/layers of our application. Here are some common suspects that you will comes across …

*Spring Stereotype annotations*

* **@Component** – Identifies a Java Class that is to be registered as a Spring Bean
* **@Controller** or **@RestController** – Web Layer services like mapping HTTP request to handler methods and processing response
* **@Repository** – Vendor neutral Exception Translation Service on DAO Classes – DataAcessException
* **@Configuration** – Java based configuration/code representing a factory to create Spring beans
* **@Service** – does nothing special, What? I’m confused!

At this point it should be clear that we could directly use @Component to wire up our Java Classes as Spring beans and inherit the magic of [Dependency Injection](http://mvpjava.com/brief-history-dependency-injection/) via annotations like @Autowired.

All these stereotype annotations were created to introduce some specialized functionality (they actually do something that @Component doesn’t). However, one annotation from this group does not do anything extra! None is more confusing than the @Service annotation. So why even use it?

What is Spring Boot?

Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services.

In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

**@SpringBootApplication**

A single @SpringBootApplication annotation is used to enable the following annotations:

* **@EnableAutoConfiguration:** It enables the Spring Boot auto-configuration mechanism.
* **@ComponentScan:** It scans the package where the application is located.
* **@Configuration:** It allows us to register extra beans in the context or import additional configuration classes.

The primary comparison between Spring and Spring Boot are discussed below:

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| To test the Spring project, we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |

Advantages

* It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.
* It provides opinionated '**starter**' POMs to simplify our Maven configuration.
* There is no requirement for **XML** configuration.
* It offers the number of **plug-ins**.
* It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
* It **increases productivity** and reduces development time.

Goals

Spring Boot is designed with the following goals −

* To avoid complex XML configuration in Spring
* To develop a production ready Spring applications in an easier way
* To reduce the development time and run the application independently
* Offer an easier way of getting started with the application

Why Spring Boot?

You can choose Spring Boot because of the features and benefits it offers as given here −

* It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
* It provides a powerful batch processing and manages REST endpoints.
* In Spring Boot, everything is auto configured; no manual configurations are needed.
* It offers annotation-based spring application
* Eases dependency management
* It includes Embedded Servlet Container

How does it work?

Spring Boot automatically configures your application based on the dependencies you have added to the project by using **@EnableAutoConfiguration** annotation.

For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

The entry point of the spring boot application is the class contains **@SpringBootApplication** annotation and the main method.

Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation.

**Q21.  What is the use of WebMvcTest annotation in Spring MVC applications?**

**@ WebMvcTest**

**WebMvcTest** annotation is used for unit testing Spring MVC Applications in cases where the test objective is to just focus on Spring MVC Components.

**What is the role of actuator in spring boot?**

A spring boot actuator is a project that provides restful web services to access the current state of an application that is running in production.

In addition, you can monitor and manage application usage in a production environment without having to code or configure any of the applications.

**What are starter dependences**

spring-boot-starter-web

spring-boot-starter-test

spring-boot-starter-jdbc

spring-boot-starter-jersey

spring-boot-starter-aop

spring-boot-starter-security

spring-boot-starter-data-jpa

spring-boot-starter-batch

spring-boot-starter-data-mongodb

spring-boot-starter-data-rest

**From where spring boot Execution is start**

**Different ways to run method after startup in spring boot**

1. Using CommandLineRunner interface.
2. With ApplicationRunner interface.
3. Spring boot Application events.
4. @Postconstruct annotation on a method.
5. The InitializingBean Interface.
6. Init attribute of @bean annotation.

**Write code for OneToMany relationship**

@Entity

@Table(name = "books")

public class Book implements Serializable {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

   @OneToMany(mappedBy = "book", fetch = FetchType.LAZY,

   cascade = CascadeType.ALL)

   private Set<Page> pages;

**}**

@Entity

@Table(name = "pages")

public class Page implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne(fetch = FetchType.LAZY, optional = false)

@JoinColumn(name = "book\_id", nullable = false)

private Book book;

}

**Write code for ManyToMany relationship**

@Entity

@Table(name = "students")

public class Student implements Serializable {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

   @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)

    @JoinTable(name = "students\_courses",

    joinColumns = {

    @JoinColumn(name = "student\_id", referencedColumnName = "id",

                nullable = false, updatable = false)},

   inverseJoinColumns = {

   @JoinColumn(name = "course\_id", referencedColumnName = "id",

               nullable = false, updatable = false)})

private Set<Course> courses = new HashSet<>();

**}**

@Entity

@Table(name = "courses")

public class Course implements Serializable {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @ManyToMany(mappedBy = "courses", fetch = FetchType.LAZY)

    private Set<Student> students = new HashSet<>();

**}**

How to connect to database in springboot.

               <dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

             <dependency>

<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc8</artifactId>

<scope>runtime</scope>

</dependency>

              <dependency>

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

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

</dependency>

**Application.property file**

server.port=8081

# Oracle settings

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe

spring.datasource.username=system

spring.datasource.password=system

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

**@PostConstruct and @PreDestroy annotations**

**@PostConstruct** : is called after the bean has been initialized and before this bean is returned to the requested object.

@PostConstruct

**public** **void** Initialize() {

        System.out.println("initializing the bean");

    }

**@PreDestroy** : is called just before the bean is removed from the container.

@PreDestroy

**public** **void** cleanUp() {

        System.out.println("cleaning up the resources");

    }

**@PostConstruct**

When we annotate a method in Spring Bean with @PostConstruct annotation, it gets executed after the spring bean is initialized. We can have only one method annotated with @PostConstruct annotation.

**@PreDestroy annotations**

When we annotate a Spring Bean method with PreDestroy annotation, it gets called when bean instance is getting removed from the context. This is a very important point to understand – if your [spring bean scope](https://www.journaldev.com/21039/spring-bean-scopes) is “prototype” then it’s not completely managed by the spring container and PreDestroy method won’t get called.

If there is a method named shutdown or close then spring container will try to automatically configure them as callback methods when bean is being destroyed.

**Explain spring bean life cycle.**

**Life cycle callback methods**

**Spring framework provides following 4 ways for controlling life cycle events of a bean:**

**InitializingBean and DisposableBean callback interfaces**

**@PostConstruct and @PreDestroy annotations**

**\*Aware interfaces for specific behavior**

**Custom init() and destroy() methods in bean configuration file**

**import org.springframework.beans.factory.DisposableBean;**

**import org.springframework.beans.factory.InitializingBean;**

**public class DemoBean implements InitializingBean, DisposableBean**

**{**

**//Other bean attributes and methods**

**@Override**

**public void afterPropertiesSet() throws Exception**

**{**

**//Bean initialization code**

**}**

**@Override**

**public void destroy() throws Exception**

**{**

**//Bean destruction code**

**}**

**}**

**23.Why Spring ?**

**11)Dependency Injection**

**9) Transaction Management**

**10)Container**

**2) Loose Coupling**

**3) Easy to test**

**4) Lightweight**

**5) Fast Development**

**8) Aspect-Oriented Programming (AOP)**

**1) Predefined Templats**s**es**

**6) Powerful abstraction**

**7) Declarative support**

**SQL injection?**

SQL injection is a [code injection](https://en.wikipedia.org/wiki/Code_injection) technique used to [attack](https://en.wikipedia.org/wiki/Attack_(computing)) data-driven applications,

in which malicious [SQL](https://en.wikipedia.org/wiki/SQL) statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).

**What is the use of @transaction annotation?**

**Connection** connection = dataSource.getConnection(); // **(1)**

**try** (connection) {

    connection.setAutoCommit(**false**); // **(2)**

*// execute some SQL statements...*

    connection.commit(); // **(3)**

} **catch** (**SQLException** e) {

    connection.rollback(); // **(4)**

}

Understanding the flow of Spring Web MVC

* As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
* The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
* The controller returns an object of ModelAndView.
* The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

* **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

Advantages of Spring MVC Framework

**Separate roles**

**Light-weight**

**Powerful Configuration**

**Reusable business code**

**Easy to test**

**Flexible Mapping.** 

**@ResponseStatus**

As the name suggests, @ResponseStatus allows us to modify the HTTP status of our response. It can be applied in the following places:

* On the exception class itself
* Along with the @ExceptionHandler annotation on methods
* Along with the @ControllerAdvice annotation on classes

**@ResponseStatus(value = HttpStatus.NOT\_FOUND)** 

**How to handle Exception in Spring MVC/ Springboot both.**

Controller Advice

The @ControllerAdvice is an annotation, to handle the exceptions globally.

Exception Handler

Using Spring Boot @ExceptionHandler Annotation: **@ExceptionHandler annotation provided by Spring Boot can be used to handle exceptions in particular Handler classes or Handler methods**.

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

package com.tutorialspoint.demo.exception;   
   
import org.springframework.web.bind.annotation.ControllerAdvice;   
   
@ControllerAdvice 

public class ProductExceptionController { 

}

Define a class that extends the RuntimeException class

package com.tutorialspoint.demo.exception;   
   
public class ProductNotfoundException extends RuntimeException { 

private static final long serialVersionUID = 1L; 

}

You can define the @ExceptionHandler method to handle the exceptions as shown. This method should be used for writing the Controller Advice class file.

@ExceptionHandler(value = ProductNotfoundException.class)   
   
public ResponseEntity<Object> exception(ProductNotfoundException exception) {

}

The complete code to handle the exception is given below. In this example, we used the PUT API to update the product. Here, while updating the product, if the product is not found, then return the response error message as “Product not found”. Note that the **ProductNotFoundException** exception class should extend the **RuntimeException**.

package com.tutorialspoint.demo.exception; 

public class ProductNotfoundException extends RuntimeException { 

  private static final long serialVersionUID = 1L; 

}

The Controller Advice class to handle the exception globally is given below. We can define any Exception Handler methods in this class file.

@ControllerAdvice   
public class ProductExceptionController { 

   @ExceptionHandler(value = ProductNotfoundException.class) 

 public ResponseEntity<Object> exception(ProductNotfoundException exception)       { 

return new ResponseEntity<>("Product not found", HttpStatus.NOT\_FOUND); 

  }   
}

The Product Service API controller file is given below to update the Product. If the Product is not found, then it throws the **ProductNotFoundException** class.

**How to modify the bean after initializing?**

A **bean post processor** allows for custom modification of new bean instances created by spring [bean factory](https://howtodoinjava.com/spring-core/different-spring-ioc-containers/). If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, we can plug in one or more **BeanPostProcessor** implementations.

**BeanPostProcessor** interface consists of exactly two callback methods

1. postProcessBeforeInitialization()

2. postProcessAfterInitialization().

**Explain spring bean life cycle**

The following image shows the process flow of the bean life cycle.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

*Bean Life Cycle Process Flow*

We can choose custom method name instead of **init()** and **destroy()**. Here, we will use init() method to execute all its code as the spring container starts up and the bean is instantiated, and destroy() method to execute all its code on closing the container.

**Use of @Order annotation?**

[**@Order**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/Order.html) **annotation** in Spring. The @Order annotation in Spring defines the sorting order of beans or components.

1. The @Order annotation in Spring framework is for the execution precedence.
2. The highest precedence advice runs first.
3. The lower number have the highest precedence (order annotation with 1 will run before 2)

**2. How to Use @Order**

To understand this annotation, let’s build the following example.

* We want to recommend a car to the potential customers.
* Toyota is the first recommendation.
* Honda is the second recommendation.
* UsedCard will be the lowest priority recommendation.

public interface Car {

 String getCarRecommendation();

 }

@Component

@Order(1)

public class Toyota implements Car {

@Override

public String getCarRecommendation()

{

 return "Toyota";

} }

@Component

@Order(2)

public class Honda implements Car

{

 @Override

public String getCarRecommendation()

 {

 return "Honda";

}

}

@Component

@Order(Ordered.LOWEST\_PRECEDENCE)

public class UsedCar implements Car {

 @Override

public String getCarRecommendation()

{

 return "Certified Car";

}

 }

**Dependency Injection**

Dependency Injection is a practice to pass dependent object to other objects. Spring has two types of Dependency Injection :

* Constructor based Injection -When container call the constructor of the class. It should be used for mandatory dependencies.

* Setter based Injection - It can be used by calling setter methods on your beans. It should be used for optional dependencies.

**IOC Container(Inversion Of Control.)**

Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and [Java POJO class](https://www.geeksforgeeks.org/pojo-vs-java-beans/). These objects are called Beans.

Spring to container do remember that Spring provides two types of Containers namely as follows:

1. BeanFactory Container
2. ApplicationContext Container

**BeanFactory**

**When using just the BeanFactory we can create one and read in some bean definitions in the XML format as follows:**

**BeanFactory factory = new XmlBeanFactory(resource);**

**ClassPathResource resource = new ClassPathResource("beans.xml");**

**BeanFactory factory = new XmlBeanFactory(resource);**

**2. ApplicationContext**

**ApplicationContext container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContext interface.**

**ApplicationContext context = new FileSystemXmlApplicationContext("beans.xml");**

**The ApplicationContext container includes all functionality of the BeanFactory container**

**Types of ApplicationContext**

**The most commonly used ApplicationContext implementations are:**

**1 .FileSystemXmlApplicationContext**

**2 .ClassPathXmlApplicationContext**

**3 .WebXmlApplicationContext**

**What is a circular dependency?**

In our day to day development life, while implementing the spring applications, we may have a requirement like class A depending on class B and class B also depending on class A,  though it is not a good practice sometimes you can’t avoid such types of dependencies.

Spring considers this type of scenario as a **circular dependency** and is also given some good practices to handle this. Here, I am going to show you the most popular way to handle this scenario.

**Where we get circular dependency problem :**

In spring we have different [types of dependency injections](https://www.onlinetutorialspoint.com/spring/dependency-injection-ioc-in-spring-with-example.html) if we inject the dependency through the constructor arguments <constructor-arg>, and the two classes are depending on each other then spring will throw an exception like [BeanCurrentlyInCreationException](https://docs.spring.io/spring/docs/1.2.9/api/org/springframework/beans/factory/BeanCurrentlyInCreationException.html), this is where we get circular dependency in spring. In the case of constructor injection, the dependencies are injected while creating the object, so that, by the time creating the object of A, there is no object of B and vice versa.

class A {   
    private B b;   
    public A(B b) {   
        this.b = b;   
    }   
}   
   
class B {   
    private A a;   
    public B(A a) {   
        this.a = a;   
    }   
}

**How to solve the circular dependency?**

In order to solve the circular dependency problem on one side, the dependency injection type must be changed to [setter injection](https://www.onlinetutorialspoint.com/spring/dependency-injection-ioc-in-spring-with-example.html). In the case of setter injection, the dependencies are injected after creating the object, not while creating the objects.

class A {   
    private B b;   
    public void setB(B b) {   
        this.b = b;   
    }   
}   
   
class B {   
    private A a;   
    public void setA(A a) {   
        this.a = a;   
    }   
}

In the above, the Spring container will inject objects of class B to A, like the following.

A a = new A();   
B b = new B();   
a.setB(b);

Explain about active profile.

In the following application, we have three profiles (local, dev, prod) and two profile-specific property files. We use the spring.profiles.active to *set* active profiles and SpringApplicationBuilder's profiles method to *add* new active profiles.

pom.xml   
src   
├── main   
│   ├── java   
│   │   └── com   
│   │       └── zetcode   
│   │           └── Application.java   
│   └── resources   
│       ├── application-dev.properties   
│       ├── application-prod.properties   
│       └── application.properties   
└── test   
    └── java

Then, we need to create three  application.properties:

1. application-dev.properties
2. application-test.properties
3. application-prod.properties

Of course, the application.properties will remain as a master properties file, but *if we override any key in the profile-specific file, the latter will gain precedence.*

I will now define DB configuration properties for in respective properties file and add code in DBConfiguration.class to pick the appropriate settings.

Here is the base  application.properties:

In DEV, we will use an in-memory database:

In TEST, we will be using a lower instance of RDS MySQL database, and in PROD, we will use a higher instance of the MySQL database. (It's the price that matters...)

Now, we are done with properties files. Let's configure in the DBConfiguration.class to pick the correct one.

We have used the @Profile("Dev")   to let the system know that this is the BEAN  that should be picked up when we set the application profile to DEV. The other two beans will not be created at all.

One last setting is how to let the system know that this is DEV, TEST, or PROD. But, how do we do this?

We will use the application.properties to use the key below:

1

spring.profiles.active=dev

From here, Spring Boot will know which profile to pick. Let's run the application now!

With the profile in DEV mode, and it should pick H2 DB.

Now, change the profile to PROD. We will see MySQL with High Config for

**Spring retry annotations**

* **@EnableRetry** – to enable spring retry in spring boot project
* **@Retryable** – to indicate any method to be a candidate of retry
* **@Recover** – to specify fallback method!

2. Enable *Spring Retry*

@EnableRetry annotation to the configuration class.

@EnableRetry   
@SpringBootApplication   
public class SpringBootApplication {   
*// ...*   
}

For non Spring Boot applications

@Configuration   
@EnableRetry   
public class Application {   
*// ...*   
}

2. *@Retryable* Annotation

As the next step to use the retry feature, we use @Retryable *annotation* on the method where we like to enable *retry* feature.

2.1 *@Retryable*

Let’s create our sample retry service to see @Retryable annotation in action.

2.2 *@Recover*

The @Recover annotation used to define a separate recovery method when a @Retryablemethod fails with a specified exception.

**what are different status codes in restful and in which situation you will consider which code.**

**200 OK**

**201 Created**

**202 Accepted**

**203 Non-authoritative Information**

**204 No Content**

**205 Reset Content**

**300 Multiple Choices**

**301 Moved Permanently**

**302 Found**

**303 See Other**

**304 Not Modified**

**305 Use Proxy**

**400 Bad Request**

**401 Unauthorized**

**402 Payment Required**

**403 Forbidden**

**404 Not Found**

**405 Method Not Allowed**

**406 Not Acceptable**

**407 Proxy Authentication Required**

**408 Request Timeout**

**409 Conflict**

**500 Internal Server Error**

**501 Not Implemented**

**502 Bad Gateway**

**503 Service Unavailable**

**504 Gateway Timeout**

**505 HTTP Version Not Supported**

<https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

**Type of session method.**

|  |  |
| --- | --- |
| **Sr.No.** | **Session Methods & Description** |
| 1 | **Transaction beginTransaction()**  Begin a unit of work and return the associated Transaction object. |
| 2 | **void cancelQuery()**  Cancel the execution of the current query. |
| 3 | **void clear()**  Completely clear the session. |
| 4 | **Connection close()**  End the session by releasing the JDBC connection and cleaning up. |
| 5 | **Criteria createCriteria(Class persistentClass)**  Create a new Criteria instance, for the given entity class, or a superclass of an entity class. |
| 6 | **Criteria createCriteria(String entityName)**  Create a new Criteria instance, for the given entity name. |
| 7 | **Serializable getIdentifier(Object object)**  Return the identifier value of the given entity as associated with this session. |
| 8 | **Query createFilter(Object collection, String queryString)**  Create a new instance of Query for the given collection and filter string. |
| 9 | **Query createQuery(String queryString)**  Create a new instance of Query for the given HQL query string. |
| 10 | **SQLQuery createSQLQuery(String queryString)**  Create a new instance of SQLQuery for the given SQL query string. |
| 11 | **void delete(Object object)**  Remove a persistent instance from the datastore. |
| 12 | **void delete(String entityName, Object object)**  Remove a persistent instance from the datastore. |
| 13 | **Session get(String entityName, Serializable id)**  Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance. |
| 14 | **SessionFactory getSessionFactory()**  Get the session factory which created this session. |
| 15 | **void refresh(Object object)**  Re-read the state of the given instance from the underlying database. |
| 16 | **Transaction getTransaction()**  Get the Transaction instance associated with this session. |
| 17 | **boolean isConnected()**  Check if the session is currently connected. |
| 18 | **boolean isDirty()**  Does this session contain any changes which must be synchronized with the database? |
| 19 | **boolean isOpen()**  Check if the session is still open. |
| 20 | **Serializable save(Object object)**  Persist the given transient instance, first assigning a generated identifier. |
| 21 | **void saveOrUpdate(Object object)**  Either save(Object) or update(Object) the given instance. |
| 22 | **void update(Object object)**  Update the persistent instance with the identifier of the given detached instance. |
| 23 | **void update(String entityName, Object object)**  Update the persistent instance with the identifier of the given detached instance. |

**scope of spring and explain**

* **singleton** – only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
* **prototype** – A new instance will be created every time the bean is requested from the spring container.
* **request** – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
* **session** – A new bean will be created for each HTTP session by the container.
* **global-session** – This is used to create global session beans for Portlet applications.

**explain about criteria interface.**

 Criteria Interface

The Criteria interface provides many methods to specify criteria. The object of    Criteria can be obtained by calling the **createCriteria()** method of Session interface.

**Syntax of createCriteria() method of Session interface**

**public** Criteria createCriteria(Class c)

**Hibernate lifecycle**

The Hibernate lifecycle contains the following states: -

* Transient state
* Persistent state
* Detached state

What is hibernate?

Hibernate is an open-source and lightweight ORM tool that is used to store, manipulate, and retrieve data from the database.

What is ORM?

ORM is an acronym for Object/Relational mapping. It is a programming strategy to map object with the data stored in the database. It simplifies data creation, data manipulation, and data access.

Mention some of the advantages of using ORM over JDBC.

ORM has the following advantages over JDBC:

* Application development is fast.
* Management of transaction.
* Generates key automatically.
* Details of SQL queries are hidden.

List the key components of Hibernate.

Key components of Hibernate are:

* Configuration
* Session
* SessionFactory
* Criteria
* Query
* Transaction

4) What are the core interfaces of Hibernate?

The core interfaces of Hibernate framework are:

* Configuration
* SessionFactory
* Session
* Query
* Criteria
* Transaction

**What is Microservices?**

Microservice Architecture is an architectural development style which builds an application as a collection of small autonomous services developed for a business domain.

* Maintainable and testable
* Loosely coupled
* Independently deployable
* Managed by a small team

**Explain microservices architecture**

**Name three commonly used tools for Microservices**

* • Wiremock, 2.) Docker and 3.) Hysrix are important Microservices tool.

**What are the advantages of microservices?**

Here, are some significant advantages of using Microservices:

* • Technology diversity, e., Microservices can mix easily with other frameworks, libraries, and databases
* • Fault isolation, e., a process failure should not bring the whole system down.
* • Greater support for smaller and parallel team
* • Independent deployment
* • Deployment time reduce

Maintainable and testable

Loosely coupled

Independently deployable

Managed by a small team

Disadvantages of Microservices

* Complete end-to-end testing is complex.
* There is a higher chance of failure during communication between different services.
* Deployment Challenges.
* The third-party applications are hard to control.
* Difficult to manage a large number of services.
* Microservices has all the associated complexities of the distributed system.
* Pre-planning is essential.
* Complex development.
* Requires a cultural shift.
* The developer needs to solve the problem, such as network latency and load balancing.
* Complex testing over a distributed environment.

**Micro service Anotation**

**@SpringbootApplication =  @Anableautoconfiguration + @configaration + @componentScans**

**@componentScans  = {("com.io.sba"),("com.io.sba.entity")}**

**@EnableAutoConfiguration**

**@configaration**

**@Bean**

**@Id**

**@Column**

**@NotNull**

**@Gmail**

**@GenrateValue**

**@NotEmpty**

**@NotBlank**

**@NoArgumentConstructor**

**@AllArgumentConstructor**

**@ToString**

**@Data**

**@Setter**

**@Getter**

**@OneToOne**

**@OnetTOMany**

**@ManyToMany**

**@Trasactional**

**@Quary**

**@Table**

**@EnableAutoGenrate**

**@Entity**

**@Repository**

**@service**

**@Controller**

**@RestController**

**@NativeQuery**

**@RequestMapping**

**@Async**

**@Transient**

**@PostMapping**

**@namedQuery**

**@putMappingh**

**@GetMapping**

**@deleteMapping**

**@NameQueries**

**@Component**

**@CrossOrigin**

**@CreatedBy,**

**@LastModifiedBy,**

**@CreatedDate,**

**@LastModifiedDate**

**@Override**

**@RequestBody**

**@ResponseBody**

**@PathVairable**

**@PathParam**

**@MatrixParam**

**@Value**

**@Valid**

**@Autowired**

**@ControllAdvise**

**@ExceptionHandler**

**@EnableWebSecurity**

**@EnableGlobalWebSecurity**

**@Primery**

**@EnableTrasactionManagement**

**@ConfigurationProperties**

**@EnableJpaRepository**

**@Qualifier**

**@ResponseStatus**

**@PostConsTrust**

**@PreDestroy**

**@Temporal**

**@PropertySource**

**@JsonProperty**

**@EgnoreJsonProperty**

**@Scheduled**

**@EgnoreJsonProperties**

**@InitBinder**

**@EnableGlobalMethoedSecurity**

**@EnabalAuthorizationServer**

**@EnabalResourceServer**

**@EnableEurekaClient**

**@EnableDiscoveryClient**

**@EnableConfigServer**

**@EnableConfigClient**

**@EnableEurekaServer**

**@EnableCircuitBreaker**

**@EnableHystrixDashboard**

**@HystrixCommand(fallbackMethod = "myFallbackMethod")**

**@EnableAdminServer.**

**@LoadBalanced.**

**@EnableFeignClients**

**@FeignClient(name="microservice-1", url="localhost:8000")**

**@ManageBean**

**@Local**

**@Patchmapping**

**@EJB**

**@Remote**

**@StateLess**

**@Statefull**

**@Resource**

**@TrasactionalAtribute**

@EnableEurekaClient - annotation is used to consider service as a eureka client.

@EnableDiscoveryClient

@EnableConfigServer

@EnableConfigClient

@EnableEurekaServer

@EnableCircuitBreaker

@EnableHystrixDashboard

@HystrixCommand(fallbackMethod = "myFallbackMethod")

@EnableAdminServer.

@LoadBalanced.

@EnableFeignClients

**@FeignClient(name="microservice-1", url="localhost:8000")**

@EnableDiscoveryClient or (@EnableEurekaClient)

you can use it to discover service instances form the Eureka Server

**How to invoke one mircoservice from another microservice (Feign) ?**

Lets call microservice-1 from microservice-2.   
If you remember our architecture its like below:

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

We can use RestTemplate for this like below:

HashMap uriVariables=new HashMap();

uriVariables.put("from",from);

uriVariables.put("to",to);

ResponseEntity<Microservice2> responseEntity=new RestTemplate().getForEntity("http://localhost:8000/microservice-1/from/USD/to/INR", Microservice2.class, uriVariables);

Microservice2 microservice2 = responseEntity.getBody();

But this is not the right approach. If there are multiple microservices which are calling each other, the code would be too cumbersome.

The alternate to this is – Feign.

In the above example we can replace it with Feign.

Lets follow the steps to enable Feign in microservice-2. You don’t have to do anything in the microservice-1.

Steps to add Feign:

1. Add below Feign dependency. Feign dependencies are not available in Spring Initializer. You need to manually add it.

<dependency>

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

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

</dependency>

2. In main class add this: @EnableFeignClients

3. Create a proxy interface. The name is the application you are supposed to call- this name you will find from application.properties. Give the signature and GetMapping parameters of the method same as of the calling method.

@FeignClient(name="microservice-1", url="localhost:8000")

public interface MyProxy {

@GetMapping("microservice-1/from/{from}/to/{to}")

public Microservice2 getValue(@PathVariable String from, @PathVariable String to);

}

4. In the controller add below method:

@GetMapping("microservice-2-feign/from/{from}/to/{to}/quantity/{quantity}")

public Microservice2 calculateFeign(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

Microservice2 microservice2 = proxy.getValue(from, to);

return new Microservice2(microservice2.getId(), from, to, microservice2.getConversionMultiple(),microservice2.getEnvironment(), quantity, quantity.multiply(microservice2.getConversionMultiple()) );

}

Thats it….

You can test by calling the URL:

<http://localhost:8100/microservice-2-feign/from/USD/to/INR/quantity/10>

The GitLab URL for both microservices are here:

<https://gitlab.com/heapsteep/microservice-1.git>

<https://gitlab.com/heapsteep/microservice-2.git>

for each microservice to interact with other microservices properly, there are 2 possible techniques, namely:

* Synchronous and
* Asynchronous.

**What are the main components of Microservices?**

Following is the list of main components of Microservices or Microservice architecture:

* Containers, Clustering.
* IaC (Infrastructure as Code Conception)
* Cloud Infrastructure
* API Gateway
* Enterprise Service Bus
* Service Delivery

**What are the use of Jenkins.**

CI --- Source, Build, test, code coverage, Docker image

 CD--- Deploy, push Docker image to docker hub.

Jenkins is an leading open source [continuous integration](http://seleniumeasy.com/jenkins-tutorials/what-is-continous-integration) server built with Java.

This [continuous integration](http://seleniumeasy.com/jenkins-tutorials/what-is-continous-integration)  server runs in servlet containers such as Apache Tomcat. Jenkins facilitates [continuous integration and continuous delivery](https://www.lambdatest.com/blog/what-is-continuous-integration-and-continuous-delivery/) in software projects by automating parts related to build, test, and deployment. This makes it easy for developers to continuously work on the betterment of the product by integrating changes to the project.

It is used to build and test software projects continuously making it easier to integrate changes to the project.

 It provides 985 plugins to support building and testing virtually any project.

**What is a CI/CD pipeline?**

Continuous integration/continuous delivery

A CI/CD pipeline automates the process of software delivery. It builds code, runs tests, and helps you to safely deploy a new version of the software. CI/CD pipeline reduces manual errors, provides feedback to developers, and allows fast product iterations.

**What is Continuous Integration, Continuous Delivery, and Continuous Deployment?**

**Continuous integration** is a software development method where members of the team can integrate their work at least once a day. In this method, every integration is checked by an automated build to search the error.

Build,test,code coverage,Docker image.

**Continuous delivery** is a software engineering method in which a team develops software products in a short cycle. It ensures that software can be easily released at any time.

**Continuous deployment** will d

Deploy,push Docker image to docker hub.

is a software engineering process in which product functionalities are delivered using automatic deployment. It helps testers to validate whether the codebase changes are correct, and it is stable or not.

**Stages of a CI/CD pipeline**

Here are the important Stages of CI/CD pipeline:

Stages of CI/CD pipeline

**Source Stage**

In the source stage, CI/CD pipeline is triggered by a code repository. Any change in the program triggers a notification to the CI/CD tool that runs an equivalent pipeline. Other common triggers include user-initiated workflows, automated schedules, and the results of other pipelines.

**Build Stage**

This is the second stage of the CI/CD Pipeline in which you merge the source code and its dependencies. It is done mainly to build a runnable instance of software that you can potentially ship to the end-user.

Programs that are written in languages like C++, Java, C, or Go language should be compiled. On the other hand, JavaScript, Python, and Ruby programs can work without the build stage.

Failure to pass the build stage means there is a fundamental project misconfiguration, so it is better that you address such issue immediately.

**Test Stage**

Test Stage includes the execution of automated tests to validate the correctness of code and the behaviour of the software. This stage prevents easily reproducible bugs from reaching the clients. It is the responsibility of developers to write automated tests.

**Deploy Stage**

This is the last stage where your product goes live. Once the build has successfully passed through all the required test scenarios, it is ready to deploy to live server.

**Example of CI/CD Pipeline**

Here is example of CI/CD pipeline:

* **Source Code Control:** Host code on GitHub as a private repository. This will help you to integrate your application with major services and software.
* **Continuous integration:** Use continuous integration and delivery platform CircleCI and commit every code. When the changes notify, this tool will pull the code available in GitHub and process to build and run the test.
* **Deploy code to UAT:** Configure CircleCI to deploy your code to AWS UAT server.
* **Deploy to production:** You have to reuse continuous integration steps for deploying code to UAT.

**Advantages of CI/CD pipelines**

Here are the pros/ benefits of CI/CD Pipeline:

* Builds and testing can be easily performed manually.
* It can improve the consistency and quality of code.
* Improves flexibility and has the ability to ship new functionalities.
* CI/CD pipeline can streamline communication.
* It can automate the process of software delivery.
* Helps you to achieve faster customer feedback.
* CI/CD pipeline helps you to increase your product visibility.
* It enables you to remove manual errors.
* Reduces costs and labour.
* CI/CD pipelines can make the software development lifecycle faster.
* It has automated pipeline deployment.
* A CD pipeline gives a rapid feedback loop starting from developer to client.

                                             What is AWS?

* AWS stands for **Amazon Web Services**.
* The AWS service is provided by the Amazon that uses distributed IT infrastructure to provide different IT resources available on demand. It provides different services such as infrastructure as a service (IaaS), platform as a service (PaaS) and packaged software as a service (SaaS).

                  Infrastructure service providing means pay as you go model

                  Network + storage + server  this data is going for virtualization form.

              Iaas  paas and saas divide in to three ways.

          Below image you can see that.

           Iaas -------infrastructure as a service.          ( Blue Color)

            Network +  storage +  server +  virtualization + Os

          hardware  service on the rent like part of  infratecture model.

         Network + storage + server  this data is going for virtualization form.

           Paas------ platform as a service.                     ( Yellow Color )

            Network +  storage +  server +  virtualization + Os  +  middleware + runtime.

             Middleware + runtime   as platform.

           SaaS------ software as a service.                      (green Color  )

      Network +  storage +  server +  virtualization + Os  +  middleware + runtime + data + application.

End to end service of the saas.

|  |
| --- |
| application |
| data |
| runtime |
| middleware |
| Os |
| virtualization |
| server |
| storage |
| Network |

Deployment model in cloud.

Public cloud.

Private cloud

Hybrid cloud.

Public cloud ---

Aws and Asure and google cloud all the public cloud .

We can run application  . We can use service and

Private cloud --  we can say  enterprise cloud.

Hybrid cloud = public + private cloud.

**EC2  instance**

1. **Amazon  EC**2  provide scalable computing capacity in aws  cloud.

       2.     you can use amazon EC2 to launch as many or as few virtual servers as you need.

              Configure security and networking and message store.

       3.  amazon EC2 enables you to scale up and scale down the reference.

       4. amazon EC2 has storage option that is EBS and instance store and preconfigure template

         Are available none as  amezon machine image.

**There are seven type of EC2 instance are there.**

   1. Genral purpose instance

   2. compute optimize instance

  3. memory optimize instance

  4. storage optimize instance

  5.Accelerated computing instance.

  6. high memory instance.

  7. previous generation instance.

**Genral purpose instance** -- Genral purpose instance provide a balance of compute memory and networking and can be use verity of workload.

It has three types of series.

Like A, M, T series.

A  series--  A1.

M series – m4, m5, m5a,m5ad,m5d.

T sesies -  T2, T3, T3a.

**compute optimize instance**  --  compute optimize instance  are ideal for compute bound application that benefit for high performance processor. We are using for beach processing.

In that we have  C  series.

C  Series.

             |

              C4, c5, c5n.

**memory optimize instance** --  memory optimize instance  are degine to develop fast performance for workload that recovere large data set in memory.

In that we have three series like

R ,X ,Z series.

R series -  R4,R5,R5N,R5ad,R5d.

X series – x1,x1e

Z series -  z1d

**storage optimize instance** -- storage optimize instance are deigned for workload that required high sequential read and write access to very large data set in are local storage.

There are three type series are there like

I , D , H  series.

**Accelerated computing instance** -- Accelerated computing  family use hardware accelerator of

Co-processor to perform some function such as floating point number calculation graphics processor or data  peturn matching more efficiently there possible in software running on CPU.

There are three type of series are there.

P , G , F series.

**high memory instance** -- high memory instance  purchage buit to run large in memory database including production development of SAP HANA in the cloud.

In that we have U series.

**Important AWS Services**

Amazon Web Services offers a wide range of different business purpose global cloud-based products. The products include storage, databases, analytics, networking, mobile, development tools, enterprise applications, with a pay-as-you-go pricing model.

Important AWS Services

**Applications of AWS services**

Amazon Web services are widely used for various computing purposes like:

* Web site hosting
* Application hosting/SaaS hosting
* Media Sharing (Image/ Video)
* Mobile and Social Applications
* Content delivery and Media Distribution
* Storage, backup, and disaster recovery
* Development and test environments
* Academic Computing
* Search Engines
* Social Networking

**Companies using AWS**

* Instagram
* Netflix
* Twitch
* LinkedIn
* Facebook
* Turner Broadcasting: $10 million
* Zoopla
* Smugmug
* Pinterest
* Dropbox

**Advantages of AWS**

Following are the pros of using AWS services:

1) Flexibility

2) Cost-effectiveness

3) Scalability/Elasticity

4) Security

**What are devOps.**

DevOps is methdollogy which is build in development + Operations.

Means development  in plan + code+ build + test

Operations  deploy + operation + Monitoring.

Which is there so many tools are there like

Git and Jenkins and Docker and Selenlum + Chef.

                                                                    WaterFowl Model

                                                                             |

                                                                         Agile Methodologgy

                                                                              |

                                                                           DevOps

Which are some of the most popular DevOps tools?

The most popular [DevOps tools](https://www.simplilearn.com/tutorials/devops-tutorial/devops-tools) include:

1. [Selenium](https://www.simplilearn.com/tutorials/selenium-tutorial/what-is-selenium)
2. [Puppet](https://www.simplilearn.com/puppet-tutorial-article)
3. [Chef](https://www.simplilearn.com/chef-tutorial-article)
4. [Git](https://www.simplilearn.com/tutorials/git-tutorial)
5. [Jenkins](https://www.simplilearn.com/tutorials/jenkins-tutorial/what-is-jenkins)
6. [Ansible](https://www.simplilearn.com/tutorials/ansible-tutorial)
7. [Docker](https://www.simplilearn.com/tutorials/docker-tutorial)

**Linex Commands.**

**17. su Command**

The [su](https://www.javatpoint.com/linux-su-commands) command provides administrative access to another user. In other words, it allows access of the Linux shell to another user.

**1. pwd Command**

The [pwd](https://www.javatpoint.com/linux-pwd) command is used to display the location of the current working directory.

**2. mkdir Command**

The [mkdir](https://www.javatpoint.com/linux-mkdir) command is used to create a new directory under any directory.

**3. rmdir Command**

The [rmdir](https://www.javatpoint.com/linux-rmdir) command is used to delete a directory.

**4. ls Command**

The [ls](https://www.javatpoint.com/linux-ls) command is used to display a list of content of a directory.

**5. cd Command**

The [cd](https://www.javatpoint.com/linux-cd) command is used to change the current directory.

**6. touch Command**

The [touch](https://www.javatpoint.com/linux-touch) command is used to create empty files. We can create multiple empty files by executing it once.

**7. cat Command**

The [cat](https://www.javatpoint.com/linux-cat) command is a multi-purpose utility in the Linux system. It can be used to create a file, display content of the file, copy the content of one file to another file, and more.

**8. rm Command**

The [rm](https://www.javatpoint.com/linux-rm) command is used to remove a file.

**9. cp Command**

The [cp](https://www.javatpoint.com/linux-cp) command is used to copy a file or directory.

**10. mv Command**

The [mv](https://www.javatpoint.com/linux-mv) command is used to move a file or a directory form one location to another location.

**12. head Command**

The [head](https://www.javatpoint.com/linux-head) command is used to display the content of a file. It displays the first 10 lines of a file.

**13. tail Command**

The [tail](https://www.javatpoint.com/linux-tail) command is similar to the head command. The difference between both commands is that it displays the last ten lines of the file content. It is useful for reading the error message.

**14. tac Command**

The [tac](https://www.javatpoint.com/linux-tac) command is the reverse of cat command, as its name specified. It displays the file content in reverse order (from the last line).

**18. id Command**

The [id](https://www.javatpoint.com/linux-id-command) command is used to display the user ID (UID) and group ID (GID).

**19. useradd Command**

The [useradd](https://www.javatpoint.com/linux-create-user) command is used to add or remove a user on a Linux server.

**20. passwd Command**

The [passwd](https://www.javatpoint.com/linux-user-password) command is used to create and change the password for a user.

**21. groupadd Command**

The [groupadd](https://www.javatpoint.com/linux-add-user-to-group) command is used to create a user group.

**22. cat Command  or  tac Command**

The [cat](https://www.javatpoint.com/linux-cat-filters) command is also used as a filter. To filter a file, it is used inside pipes.

**23. cut Command**

The [cut](https://www.javatpoint.com/linux-cut) command is used to select a specific column of a file. The '-d' option is used as a delimiter, and it can be a space (' '), a slash (/), a hyphen (-), or anything else. And, the '-f' option is used to specify a column number.

**24. grep Command**

The [grep](https://www.javatpoint.com/linux-grep) is the most powerful and used filter in a Linux system.

**27. tee command**

The [tee](https://www.javatpoint.com/linux-tee) command is quite similar to the cat command.

**30. wc Command**

The [wc](https://www.javatpoint.com/linux-wc) command is used to count the lines, words, and characters in a file.

**33. gzip Command**

The [gzip](https://www.javatpoint.com/linux-gzip) command is used to truncate the file size. It is a compressing tool. It replaces the original file by the compressed file having '.gz' extension.

**34. gunzip Command**

The [gunzip](https://www.javatpoint.com/linux-gzip) command is used to decompress a file. It is a reverse operation of gzip command.

**44. exit Command**

Linux [exit](http://javatpoint.com/linux-exit-command) command is used to exit from the current shell.

**What is GIT?**

Git is an open source distributed version control system and source code management (SCM) system.

It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.

with an insistence to control small and large projects with speed and efficiency.

Git Flow / Git Branching Model

It is referred to as **Branching Model** by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.

There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:

* **Master**
* **Develop**
* **Hotfixes**
* **Release branches**
* **Feature branches**

Git Master Branch

The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

Operations on Branches

We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

**The Operations that can be performed on a branch:**

Create Branch

List Branch

Delete Branch

Switch Branch

Rename Branch

Merge Branch

Basic Git Commands

Staging & commits

Git Init

Git Add

Git Commit

Git Clone

Git Repository

 Git Index

Git Pull

Git Push

Git Checkout

Git Rebase

Git Ignore

Git Head

Git Stash

 Git Ignore

Git Fork

Git Head

Git Origin Master

Git Remote

Git TagsUpstream & Downstream

Collaborating

   Git Fetch

   Git Pull

   Git Push

   Branching & Merging

Git BranchMerge & Merge Conflict

       Git Rebase

        Git Squash

 Undoing changes

Git Checkout

Git Revert

Git Reset

Git Rm

Git Cherry-pick

**How to download Git and Install Git?**

**Step1**

To download the Git installer, visit the Git's official site and go to download page. The link for the download page is <https://git-scm.com/downloads>. The page looks like as

56.9M

1.1K

C++ vs Java

Click on the package given on the page as **download 2.23.0 for windows**. The download will start after selecting the package.

Now, the Git installer package has been downloaded.

Install Git

**Step2**

Click on the downloaded installer file and select **yes** to continue. After the selecting **yes** the installation begins, and the screen will look like as

Click on **next** to continue.

**Step3**

Default components are automatically selected in this step. You can also choose your required part.

Click next to continue.

**Step4**

The default Git command-line options are selected automatically. You can choose your preferred choice. Click **next** to continue.

**Step5**

The default transport backend options are selected in this step. Click **next** to continue.

**Step6**

Select your required line ending option and click next to continue.

**Step7**

Select preferred terminal emulator clicks on the **next** to continue.

**Step8**

This is the last step that provides some extra features like system caching, credential management and symbolic link. Select the required features and click on the **next** option.

**Step9**

The files are being extracted in this step.

Therefore, The Git installation is completed. Now you can access the **Git Gui** and **Git Bash**.

The **Git Gui** looks like as

It facilitates with three features.

* Create New Repository
* Clone Existing Repository
* Open Existing Repository

The **Git Bash** looks like as

Challenges of Microservices Architecture

* Microservices require a heavy investment.
* It cost a lot in staff selection and maintenance.
* We need excessive planning for managing operations overhead.

**The main challenges from a technical point of view:**

* The components in microservices always rely on each other, so; it requires communication between them in the application.
* There are also a lot of challenges in deployment.
* Testing and Debugging are very challenging.
* It requires complete component automation and application maintenance.
* It receives heavy operations overhead.
* It requires skilled professionals to support heterogeneously distributed microservices.
* Bounded Context
* **Configuration Management.**
* **Debugging.**
* Dynamic Scale up and Scale Down
* Monitoring
* Fault Tolerance
* Cyclic dependencies
* DevOps Culture

**5) What is Spring Cloud?**

Spring cloud is an Integration software that integrates with external systems. It allows microservices framework to build applications which perform restricted amounts of data processing.

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, leadership election, distributed sessions, cluster state)

**Spring Cloud**: In Microservices, the Spring cloud is a system that integrates with external systems. This is a short-lived framework designed to build applications quickly. It contributes significantly to microservice architecture due to its association with finite amounts of data processing. Some of the features of spring cloud are shown below:

**10. What issues are generally solved by spring clouds?**

The following problems can be solved with spring cloud:

* **Complicated issues caused by distributed systems:** This includes network issues, latency problems, bandwidth problems, and security issues.
* **Service Discovery issues:** Service discovery allows processes and services to communicate and locate each other within a cluster.
* **Redundancy issues:** Distributed systems can often have redundancy issues.
* **Load balancing issues:** Optimize the distribution of workloads among multiple computing resources, including computer clusters, central processing units, and network links.
* **Reduces performance issues:** Reduces performance issues caused by various operational overheads

**What are the advantages of using Spring Cloud**

When developing distributed microservices with Spring Boot we face the following issues-

* **Complexity associated with distributed systems-**   
  This overhead includes network issues, Latency overhead, Bandwidth issues, security issues.
* **Service Discovery-**   
  Service discovery tools manage how processes and services in a cluster can find and talk to one another. It involves a directory of services, registering services in that directory, and then being able to lookup and connect to services in that directory.
* **Redundancy-**   
  Redundancy issues in distributed systems.
* **Loadbalancing-**   
  Load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives.
* **Performance issues-**   
  Performance issues due to various operational overheads.
* **Deployment complexities-**   
  Requirement of Devops skills.

**20) What is a CDC?**

CDC is Consumer-Driven Contract. It is a pattern for developing Microservices so that external systems can use them.

As the name implies, CDC (Consumer-Driven Contract) basically ensures service communication compatibility by establishing an agreement between consumers and service providers regarding the format of the data exchanged between them. An agreement like this is called a contract. Basically, it is a pattern used to develop Microservices so that they can be efficiently used by external systems.

**19. Name some famous companies that use Microservice architecture.**

Microservices architecture has replaced monolithic architecture for most large-scale websites like:

* Twitter
* Netflix
* Amazon, etc

**Explain the term Eureka in Microservices.**

Eureka Server, also referred to as Netflix Service Discovery Server, is an application that keeps track of all client-service applications. As every Microservice registers to Eureka Server, Eureka Server knows all the client applications running on the different ports and IP addresses. It generally uses Spring Cloud and is not heavy on the application development process

What are the main differences between Microservices and Monolithic Architecture?

The main differences between Microservices and Monolithic Architecture:

|  |  |
| --- | --- |
| **Microservices** | **Monolithic Architecture** |
| The service startup is fast in Microservices. | The service startup takes time as it is slow in Monolithic Architecture. |
| It is a loosely coupled architecture. | It is primarily a tightly coupled architecture. |
| In Microservices, if you make changes in a single data model, it does not affect others. | In Monolithic Architecture, any changes in the data model affect the entire database. |
| It mainly focuses on products, not projects. | It mainly focuses on the whole project. |

**7) What are main differences between Microservices and Monolithic Architecture?**

|  |  |
| --- | --- |
| **Microservices** | **Monolithic Architecture** |
| Service Startup is fast | Service startup takes time |
| Microservices are loosely coupled architecture. | Monolithic architecture is mostly tightly coupled. |
| Changes done in a single data model does not affect other Microservices. | Any changes in the data model affect the entire database |
| Microservices focuses on products, not projects | Monolithic put emphasize over the whole project |

**What do you mean by Distributed Transaction?**

Distributed transactions are an outdated approach in today's microservice architecture that leaves the developer with severe scalability issues. Transactions are distributed to several services that are called to complete the transaction in sequence. With so many moving parts, it is very complex and prone to failure.

**Explain type of tests mostly used in Microservices.**

While working with microservices, testing becomes quite complex as there are multiple microservices working together. So, tests are divided into different levels.

As there are multiple microservices working together, microservice testing becomes quite complex when working with microservices. Consequently, tests are categorized according to their level:

**Botton-level tests:** The bottom-level tests are those that deal with technology, such as unit tests and performance tests. This is a completely automated process.     
**Middle-level tests:** In the middle, we have exploratory tests such as stress tests and usability tests.      
**Top-level tests:** In the top-level testing, we have a limited number of acceptance tests. The acceptance tests help stakeholders understand and verify the software features.

 At the **bottom level**, we have **technology-facing tests** like- unit tests and performance tests. These are completely automated.

* At the **middle level**, we have tests for **exploratory testing** like the stress tests and usability tests.
* At the **top level,**we have **acceptance tests** that are few in number. These acceptance tests help stakeholders in understanding and verifying software features.

**Explain Container in Microservices.**

Containers are useful technologies for allocating and sharing resources. It is considered the most effective and easiest method for managing microservice-based applications to develop and deploy them individually. Using Docker, you may also encapsulate a microservice along with its dependencies in a container image, which can then be used to roll on-demand instances of the microservice without any additional work.

What is the method to override a Spring Boot project's default properties?

We can do it by specifying the properties in application.properties. The Spring MVC applications need the suffix and the prefix to be specified. This can be done by:

* **For suffix:**mvc.view.suffix: .jsp
* **For prefix:**mvc.view.prefix: /WEB-INF/

**19) What is the meaning of Semantic monitoring in Microservices architecture?**

Semantic monitoring combines automated tests with monitoring of the application. It allows you to find out reasons why your business is not getting more profits.

**How independent micro-services communicate with each other?**

It depends upon your project needs. However, in most cases, developers use HTTP/REST with JSON or Binary protocol. However, they can use any communication protocol.

**principles microservice architecture**

* Scalability
* Availability
* Resiliency
* Flexibility
* Independent, autonomous
* Decentralized governance
* Failure isolation
* Auto-Provisioning
* Continuous delivery through DevOps

What are the different Microservices Design Patterns

The different Microservices Design Patterns are -

* Aggregator Microservice Design Pattern
* API Gateway Design Pattern
* Chain of Responsibility Design Pattern
* Branch Microservice Design Pattern
* Circuit Breaker Design Pattern
* Asynchronous Messaging Design Pattern

                             Saga pattern

**Chained Microservice Pattern**

There will be multiple dependencies of for single services or microservice eg: Sale microservice has dependency products microservice and order microservice. Chained microservice design pattern will help you to provide the consolidated outcome to your request.

The request received by a microservice: 1, which is then communicating with microservice-2 and it may be communicating with microservice-3. All these services are synchronous calls.

**Branch Pattern**

Branch microservice pattern is a mix of Aggregator and Chain design patterns and allows simultaneous request/response processing from two or more microservices. The invoked microservice can be chains of microservices. Branch pattern can also be used to invoke different chains of microservices, or a single chain, based on your business needs

**Circuit Breaker Pattern**

🌟 The Circuit Breaker design pattern is used to stop the request and response process if a service is not working, as the name suggests.

Figure 1: [Circuit Breaker Pattern](https://www.edureka.co/blog/microservices-design-patterns#CircuitBreaker)

🌟 As an example, assume a consumer sends a request to get data from multiple services. But, one of the services is unavailable due to technical issues. There are mainly two issues you will face now.

* First, because the consumer will be unaware that a particular service is unavailable (failed), so the requests will be sent to that service continuously.
* The second issue is that network resources will be exhausted with low performance and user experience.

🌟 You can leverage the Circuit Breaker Design Pattern to avoid such issues. The consumer will use this pattern to invoke a remote service using a proxy. This proxy will behave as a circuit barrier.

🌟 When the number of failures reaches a certain threshold, the circuit breaker trips for a defined duration of time.

🌟 During this timeout period, any requests to the offline server will fail. When that time period is up, the circuit breaker will allow a limited number of tests to pass, and if those requests are successful, the circuit breaker will return to normal operation. If there is a failure, the time out period will start again.

Figure 14: [States of Circuit Breaker](https://martinfowler.com/bliki/CircuitBreaker.html)

The Circuit Breaker pattern has 3 states.

1. **Open.**
2. **Closed.**
3. **Half-Open.**

🌟 The circuit breaker works (pass requests through the service) normally when it is in the “**closed**” state. But when the failures exceed the threshold limit, the circuit breaker trips. As seen in the above diagram, this “**opens**” the circuit (state switches to “open”).

🌟 When the circuit is “**open**” incoming requests will return with error without any attempt to execute the real operation

🌟 After a duration of time, the circuit breaker goes into the “**half-open**” state. In this state, the circuit breaker will allow a limited number of test requests to pass through and if the requests succeed, the circuit breaker resets and returns to the “**closed**” state and the traffic will go through as usual. If this request fails, the circuit breaker returns to the open state until another timeout.

**API Gateway Pattern**

The **API gateway pattern** is recommended if you want to design and build **complex large microservices**-**based applications** with multiple client applications. The pattern is similar to the **facade pattern** from object-oriented design,

We said that It is similar to the **facade pattern** of Object-Oriented Design, so it provides a **single entry point** to the APIs with encapsulating the underlying system architecture.

the **API gateway** locate between the client apps and the internal microservices. It is working as a **reverse proxy** and routing requests from clients to backend services. It is also provide **cross**-**cutting concerns** like **authentication**, **SSL termination**, and cache.

You can see the image that is collect client request in **single entrypoint** and route request to internal microservices.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

We are going to iterate our **e-commerce architecture** with adding API Gateway pattern.

As you can see that we have added to **single API Gateway** in our application.

This will handle the client requests and route the internal microservices, also **aggregate multiple internal microservices** into a single client request and performs **cross-cutting concerns** like **Authentication** and **authorization**, Rate limiting and throttling and so on..

**Kubectl** is the command line configuration tool to interact with Kubernetes clusters using Kubernetes API server. kubectl allows users to create, inspect, update, and delete Kubernetes objects.

**Kubectl Charctersitisc**

* Kubectl can be pronounced as “cube CTL”, “kube control”, “cube cuttle”
* It is a is a robust CLI that runs commands against the Kubernetes cluster and controls the cluster manager
* kubectl is known as the swiss army knife of container orchestration and management
* kubectl is designed to make this process more comfortable and straightforward
* kubectl allows users to create, inspect, update, and delete Kubernetes objects
* Every Kubernetes command has an API endpoint, and kubectl’s primary purpose is to carry out HTTP requests to the API.

**How Kubectl works?**

Every time when we run a command with kubectl, it builds an HTTP REST API request underneath and sends this request to the Kubernetes API server, and then retrieves the result and displays it on your terminal. In fact, when we want to execute any Kubernetes operation, we can simply make an HTTP request to its corresponding API endpoint.

Kuberents can be controlled with a tool like curl by manually issuing HTTP requests to the Kubernetes API but  Kubectl just makes it easier for us to use the Kubernetes API.

**Most Common Kubectl Commands:**

* **Cluster Management:** A Kubernetes cluster is a set of nodes that run containerized applications. It allows containers to run across multiple machines and environments: virtual, physical, cloud-based, and on-premises. Following kubectl commands can be used to manage a cluster
* **kubectl cluster-info** : Display endpoint information about the master and services in the cluster
* **kubectl version** : Display the Kubernetes version running on the client and server
* **kubectl config view** : Get the configuration of the cluster
* **kubectl api-resource** : List the API resources that are available
* **kubectl api-versions** : List the API versions that are available
* **kubectl get all –all -namespaces :** List everything
* **Listing Resources:** Kubernets resources also known as Kubernetes objects associated to a specific namespace, you can either use individual kubectl get command to list down each resource one by one, or you can list down all the resources in a Kubernetes namespace by running a single command. Following are the list of commands to get the resources information.
* **kubectl get namespaces** : Generate a plain-text list of all namespaces:
* **kubectl get pods** : Generate a plain-text list of all pods
* **kubectl get pods -o wide** : Generate a detailed plain-text list of all pods
* **kubectl get pods–field-selector=spec. nodeName=[server-name]** : Generate a list of all pods running on a particular node server
* **kubectl get replicationcontroller [replication-controller-name]** : List a specific replication controller in plain text
* **kubectl get replicationcontroller, services :** Generate a plain-text list of all replication controllers and services
* **Daemonsets :** A Daemonset ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a **DaemonSet** will clean up the Pods it created.
* **kubectl get daemonset** : List one or more daemonsets
* **kubectl edit daemonset <daemonset\_name**> : Edit and update the definition of one or more daemonset
* **kubectl delete daemonset <daemonset\_name>** : Delete a daemonset
* **kubectl create daemonset <daemonset\_name>** : Create a new daemonset
* **kubectl rollout daemonset** : Manage the rollout of a daemonset
* **kubectl describe ds <daemonset\_name> -n <namespace\_name>** : Display the detailed state of daemonsets within a namespace
* **Deployments :** A **Kubernetes Deployment** is used to tell **Kubernetes** how to create or modify instances of the pods that hold a containerized application. **Deployments** can scale the number of replica pods, enable rollout of updated code in a controlled manner, or roll back to an earlier **deployment** version if necessary.
* **kubectl get deployment** : List one or more deployments
* **kubectl describe deployment <deployment\_name>** : Display the detailed state of one or more deployments
* **kubectl edit deployment <deployment\_name>** : Edit and update the definition of one or more deployment on the server
* **kubectl create deployment <deployment\_name>** : Create one a new deployment
* **kubectl delete deployment <deployment\_name>** : Delete deployments
* **kubectl rollout status deployment <deployment\_name>** : See the rollout status of a deployment
* **Events:** Kubernetes **events** are objects that show you what is happening inside a cluster, such as what decisions were made by the scheduler or why some pods were evicted from the node. **Events** are the first thing to look at for application, as well as infrastructure operations when something is not working as expected. Following are the kubectl commands to get the events.
* **kubectl get events** : List recent events for all resources in the system
* **kubectl get events –field-selector type=Warning** : List Warnings only
* **kubectl get events –field-selector involvedObject.kind!=Pod** : List events but exclude Pod events
* **kubectl get events –field-selector involvedObject.kind=Node, involvedObject.name=<node\_name>** : Pull events for a single node with a specific name
* **kubectl get events –field-selector type!=Normal** : Filter out normal events from a list of events
* **Logs :** Kubernets logs commands can be used to monitor, logging and debugging the pods.
* **kubectl logs <pod\_name>** : Print the logs for a pod
* **kubectl logs –since=1h <pod\_name>** : Print the logs for the last hour for a pod
* **kubectl logs –tail=20 <pod\_name>** : Get the most recent 20 lines of logs
* **kubectl logs -f <service\_name> [-c <$container>]** : Get logs from a service and optionally select which container
* **kubectl logs -f <pod\_name>** : Print the logs for a pod and follow new logs
* **kubectl logs -c <container\_name> <pod\_name>** : Print the logs for a container in a pod
* **kubectl logs <pod\_name> pod.log** : Output the logs for a pod into a file named ‘pod.log’
* **kubectl logs –previous <pod\_name>** : View the logs for a previously failed pod
* **Namespaces :** Namespaces are **Kubernetes objects** which partition a single Kubernetes cluster into multiple **virtual clusters**. Each **Kubernetes namespace** provides the scope for Kubernetes Names it contains; which means that using the combination of an object name and a Namespace, each object gets an **unique identity** across the cluster.
* **kubectl create namespace <namespace\_name>** : Create namespace <name>
* **kubectl get namespace <namespace\_name>** : List one or more namespaces
* **kubectl describe namespace <namespace\_name>** : Display the detailed state of one or more namespace
* **kubectl delete namespace <namespace\_name>** : Delete a namespace
* **kubectl edit namespace <namespace\_name>** : Edit and update the definition of a namespace
* **kubectl top namespace <namespace\_name>** : Display Resource (CPU/Memory/Storage) usage for a namespace
* **Node Operations:** A Node is a **worker machine** in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each **Node** is managed by the control plane. A Node can have **multiple pods**, and the Kubernetes control plane automatically handles scheduling the pods across the Nodes in the **cluster**. Following commands can be used for Node Operations.
* **kubectl taint node <node\_name>** : Update the taints on one or more nodes
* **kubectl get node** : List one or more nodes
* **kubectl delete node <node\_name>** : Delete a node or multiple nodes
* **kubectl top node** : Display Resource usage (CPU/Memory/Storage) for nodes
* **kubectl describe nodes | grep Allocated -A 5** : Resource allocation per node
* **kubectl get pods -o wide | grep <node\_name>** : Pods running on a node
* **kubectl annotate node <node\_name>** : Annotate a node
* **kubectl cordon node <node\_name>** : Mark a node as unschedulable
* **kubectl uncordon node <node\_name>** : Mark node as schedulable
* **kubectl drain node <node\_name>** : Drain a node in preparation for maintenance
* **kubectl label node** : Add or update the labels of one or more nodes
* **Pods :** Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, it creates Pods with containers inside them. Each Pod is tied to the Node where it is scheduled, and remains there until termination or deletion or restarted.  Following kubectl command can be used for Pods Operations.
* **kubectl get pod** : List one or more pods
* **kubectl delete pod <pod\_name>** : Delete a pod
* **kubectl create pod <pod\_name>** : Create a pod
* **kubectl exec <pod\_name> -c <container\_name> <command>** : Execute a command against a container in a pod
* **kubectl exec -it <pod\_name> /bin/sh** : Get interactive shell on a a single-container pod
* **kubectl top pod** : Display Resource usage (CPU/Memory/Storage) for pods
* **kubectl describe pod <pod\_name>** : Display the detailed state of a pods
* **kubectl annotate pod <pod\_name> <annotation>** : Add or update the annotations of a pod
* **kubectl label pod <pod\_name>** : Add or update the label of a pod
* **Replication Controllers and ReplicaSets**
* **kubectl get rc** : List the replication controllers
* **kubectl get rc –namespace=”<namespace\_name>”** : List the replication controllers by namespace
* **kubectl get replicasets** : List ReplicaSets
* **kubectl describe replicasets <replicaset\_name>** : Display the detailed state of one or more ReplicaSets
* **kubectl scale –replicas=[x]** : Scale a ReplicaSet
* **Secrets:** A **Kubernets Secret** is an object that contains a small amount of sensitive data such as a **password**, a token, or **a key**. Such information might otherwise be put in a Pod specification or in an image. Users can create Secrets and the system also creates some Secrets using following kubectl commands.
* **kubectl create secret** : Create a secret
* **kubectl get secrets** : List secrets
* **kubectl describe secrets** : List details about secrets
* **kubectl delete secret <secret\_name>** : Delete a secret
* **Services and Service Accounts:** A **Kubernetes service** is a logical abstraction for a deployed group of pods in a cluster (which all perform the same function) and Service accounts are used to provide an identity for pods. Pods that want to interact with the API server will authenticate with a particular service account.
* **kubectl get services :** List one or more services
* **kubectl describe services :** Display the detailed state of a service
* **kubectl expose deployment [deployment\_name] :** Expose a replication controller, service, deployment or pod as a new Kubernetes service
* **kubectl edit services :** Edit and update the definition of one or more services
* **kubectl get serviceaccounts :** List service accounts
* **kubectl describe serviceaccounts :** Display the detailed state of one or more service accounts
* **kubectl replace serviceaccount :** Replace a service account
* **kubectl delete serviceaccount <service\_account\_name> :** Delete a service account

**21) What is the use of Docker?**

What is Docker?

Docker is an **open-source centralized platform designed** to create, deploy, and run applications.

create springboot app

create a project jar file

create docker file

create a image with help of docker file.

Run the docker image in Docker Container.

Dockerfile, Docker Images & Docker Containers are three important terms that you

need to understand while using Docker.

Docker uses **container** on the host's operating system to run applications.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

**Docker  commands .**

* [**docker –version**](https://www.edureka.co/blog/docker-commands/#version)
* [**docker push**](https://www.edureka.co/blog/docker-commands/#push)
* [**docker pull**](https://www.edureka.co/blog/docker-commands/#pull)
* [**docker images**](https://www.edureka.co/blog/docker-commands/#images)
* [**docker commit**](https://www.edureka.co/blog/docker-commands/#commit)
* [**docker build**](https://www.edureka.co/blog/docker-commands/#build)
* [**docker run**](https://www.edureka.co/blog/docker-commands/#run)
* [**docker stop**](https://www.edureka.co/blog/docker-commands/#stop)
* [**docker login**](https://www.edureka.co/blog/docker-commands/#login)
* [**docker ps**](https://www.edureka.co/blog/docker-commands/#ps)
* [**docker ps -a**](https://www.edureka.co/blog/docker-commands/#psa)
* [**docker exec**](https://www.edureka.co/blog/docker-commands/#exec)
* [**docker kill**](https://www.edureka.co/blog/docker-commands/#kill)
* [**docker rm**](https://www.edureka.co/blog/docker-commands/#rm)
* [**docker rmi**](https://www.edureka.co/blog/docker-commands/#rmi)

Containers ensure that our application works in any environment like development, test, or production.

48.1M

885

HTML Tutorial

**Docker includes components such as**

**Docker client,**

**Docker server,**

**Docker machine,**

**Docker hub,**

**Docker composes,** etc.

**Docker Containers**

Docker containers are the **lightweight** alternatives of the virtual machine.

It allows developers to package up the application with all its libraries and dependencies, and ship it as a single package.

The advantage of using a docker container is that you don't need to allocate any RAM and disk space for the applications.

It automatically generates storage and space according to the application requirement.

Advantages of Docker

There are the following advantages of Docker -

* It runs the container in seconds instead of minutes.
* It uses less memory.
* It provides lightweight virtualization.
* It does not a require full operating system to run applications.
* It uses application dependencies to reduce the risk.
* Docker allows you to use a remote repository to share your container with others.
* It provides continuous deployment and testing environment.

Disadvantages of Docker

There are the following disadvantages of Docker -

* It increases complexity due to an additional layer.
* In Docker, it is difficult to manage large amount of containers.
* Some features such as container self -registration, containers self-inspects, copying files form host to the container, and more are missing in the Docker.
* Docker is not a good solution for applications that require rich graphical interface.

**Dockerfile, Images & Containers**

Dockerfile, Docker Images & Docker Containers are three important terms that you need to understand while using Docker.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

As you can see in the above diagram when the Dockerfile is built, it becomes a Docker Image and when we run the Docker Image then it finally becomes a Docker Container.

Refer below to understand all the three terms.

**Dockerfile:** A Dockerfile is a text document which contains all the commands that a user can call on the command line to assemble an image. So, Docker can build images automatically by reading the instructions from a Dockerfile. You can use docker build to create an automated build to execute several command-line instructions in succession.

**Docker Image:** In layman terms, Docker Image can be compared to a template which is used to create Docker Containers. So, these read-only templates are the building blocks of a Container. You can use docker run to run the image and create a container.

Docker Images are stored in the Docker Registry. It can be either a user’s local repository or a public repository like a Docker Hub which allows multiple users to collaborate in building an application.

**Docker Container:** It is a running instance of a Docker Image as they hold the entire package needed to run the application. So, these are basically the ready applications created from Docker Images which is the ultimate utility of Docker.

**Docker  commands .**

* [**docker –version**](https://www.edureka.co/blog/docker-commands/#version)
* [**docker push**](https://www.edureka.co/blog/docker-commands/#push)
* [**docker pull**](https://www.edureka.co/blog/docker-commands/#pull)
* [**docker images**](https://www.edureka.co/blog/docker-commands/#images)
* [**docker commit**](https://www.edureka.co/blog/docker-commands/#commit)
* [**docker build**](https://www.edureka.co/blog/docker-commands/#build)
* [**docker run**](https://www.edureka.co/blog/docker-commands/#run)
* [**docker stop**](https://www.edureka.co/blog/docker-commands/#stop)
* [**docker login**](https://www.edureka.co/blog/docker-commands/#login)
* [**docker ps**](https://www.edureka.co/blog/docker-commands/#ps)
* [**docker ps -a**](https://www.edureka.co/blog/docker-commands/#psa)
* [**docker exec**](https://www.edureka.co/blog/docker-commands/#exec)
* [**docker kill**](https://www.edureka.co/blog/docker-commands/#kill)
* [**docker rm**](https://www.edureka.co/blog/docker-commands/#rm)
* [**docker rmi**](https://www.edureka.co/blog/docker-commands/#rmi)

So, let’s get started:

**Docker Commands**

1. **docker –version**

This command is used to get the currently installed version of docker

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

2. **docker pull**

**Usage: docker pull <image name>**

This command is used to pull images from the **docker repository**(hub.docker.com)

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
3. **docker run**

**Usage: docker run -it -d <image name>**

This command is used to create a container from an image

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
4. **docker ps**

This command is used to list the running containers

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
5. **docker ps -a**

This command is used to show all the running and exited containers

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
6. **docker exec**

**Usage: docker exec -it <container id> bash**

This command is used to access the running container

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
7. **docker stop**

**Usage: docker stop <container id>**

This command stops a running container

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
8. **docker kill**

**Usage: docker kill <container id>**

This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
9. **docker commit**

**Usage: docker commit <conatainer id> <username/imagename>**

This command creates a new image of an edited container on the local system

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
10. **docker login**

This command is used to login to the docker hub repository

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
11. **docker push**

**Usage: docker push <username/image name>**

This command is used to push an image to the docker hub repository

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
12. **docker images**

This command lists all the locally stored docker images

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
13. **docker rm**

**Usage: docker rm <container id>**

This command is used to delete a stopped container

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
14. **docker rmi**

**Usage: docker rmi <image-id>**

This command is used to delete an image from local storage

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif   
   
15. **docker build**

**Usage: docker build <path to docker file>**

This command is used to build an image from a specified docker file

[**Docker Certification Training Course**](https://www.edureka.co/docker-training)

[Weekday / Weekend Batches**See Batch Details**](https://www.edureka.co/docker-training)

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

**Microservices Example – Demo**

To demonstrate the concepts of microservices, I have created 3 Maven Projects called as Doctor\_Microservice\_Edureka, Diagnosis\_Microservice\_Edureka, and Patient\_Microservice\_Edureka using Spring Boot.

Refer to the snapshot below.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

Before you understand how these 3 projects interact with each other. Let me brief you on the files of these projects.

To explain this I will consider the project  **Patient\_Microservice\_Edureka** and list down its basic files.

Refer to the snapshot below.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

**Pom.xml** – Dependencies are added for the creation of REST services.

**Application.java** – Identical class in all three projects. Acts as an initiator to Spring Boot.

**ApplicationConfiguration.java** – Research configuration class responsible for exposing REST services for application users.

**Patient.java** – A simple class consisting of input such as the patient’s name, id, email.

**PatientRest.java** – Starts the implementation of the REST services in the project.

In this way, similar files are created for the other 2 projects with some additional files in Doctor\_Microservice.

REST services are thus created to search patients and the diagnosis. Keys of patient and diagnosis are passed as a parameter to a method(PatientDetails) in Doctor\_Microservice.This method gets the data of the patients and diseases.

https://c1h-word-edit-15.cdn.office.net/we/s/hA3596C17DAD9A003_resources/1033/progress.gif

**JUnit Testing...**

What is the Unit Test Case?

A Unit Test Case is the combination of input data and expected output result. It is defined to test the functionality of a unit.

What is the use of @Test annotation?

The @Test annotation is used to mark the method as the test method.

What are the important JUnit annotations?

The test runner is used to execute the test cases.

* @Test
* @BeforeClass
* @Before
* @After
* @AfterClass
* @Ignore
* @RunWith and @Suite
* @Suite
* @InjectMocks
* @Mock
* @runwith(mockitojunitrunner.class)

* @mock dataservice dataservicemock; - create a mock for dataservice.
* @injectmocks somebusinessimpl businessimpl; - inject the mocks as dependencies into businessimpl.
* @runwith(mockitojunitrunner.class) - the junit runner which causes all the initialization magic with @mock and @injectmocks to happen before the tests are run.

**notes**

* @runwith(mockitojunitrunner.class) public class businessservicesmocktest - the junit runner which causes all the initialization magic with @mock and @injectmocks to happen before the tests are run.
* @mock dataservice dataservicemock - create a mock for dataservice.
* @injectmocks businessservice businessimpl - inject the mocks as dependencies into businessservice.
* there are three test methods testing three different scenarios: multiple values, one value and no value passed in.

**notes**

* @runwith(springrunner.class) - spring runner is used to launch up a spring context in unit tests.
* @springboottest - this annotation indicates that the context under test is a @springbootapplication . the complete springboottutorialbasicsapplication is launched up during the unit test.
* @mockbean dataservice dataservicemock - @mockbean annotation creates a mock for dataservice. this mock is used in the spring context instead of the real dataservice.
* @autowired businessservice businessimpl - pick the business service from the spring context and autowire it in.

spring boot test starter brings in a wide range of dependencies for unit testing.

* basic test framework - junit
* mocking - mockito
* assertion - assertj, hamcrest
* spring unit test framework - spring test

< dependency >

<groupid > junit < /groupid>

<artifactid > junit < /artifactid>

<version > 4.12 < /version>

<scope > compile < /scope>

</dependency>

<dependency>

<groupid > org.assertj < /groupid>

<artifactid > assertj - core < /artifactid>

<version > 3.8 .0 < /version>

<scope > compile < /scope>

</dependency>

<dependency >

 <groupid > org.mockito < /groupid>

<artifactid > mockito - core < /artifactid>

<version > 2.11 .0 < /version>

<scope> compile < /scope>

</dependency>

< dependency >

 <groupid > org.springframework < /groupid>

<artifactid > spring - test < /artifactid>

<version > 5.0 .1.release < /version>

<scope > compile < /scope>

</dependency>

Assert Class

This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of Assert class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)**  Checks that two primitives/objects are equal. |
| 2 | **void assertFalse(boolean condition)**  Checks that a condition is false. |
| 3 | **void assertNotNull(Object object)**  Checks that an object isn't null. |
| 4 | **void assertNull(Object object)**  Checks that an object is null. |
| 5 | **void assertTrue(boolean condition)**  Checks that a condition is true. |
| 6 | **void fail()**  Fails a test with no message. |

Assertion

Some of the important methods of Assert class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void assertEquals(boolean expected, boolean actual)**  Checks that two primitives/objects are equal. |
| 2 | **void assertTrue(boolean condition)**  Checks that a condition is true. |
| 3 | **void assertFalse(boolean condition)**  Checks that a condition is false. |
| 4 | **void assertNotNull(Object object)**  Checks that an object isn't null. |
| 5 | **void assertNull(Object object)**  Checks that an object is null. |
| 6 | **void assertSame(object1, object2)**  The assertSame() method tests if two object references point to the same object. |
| 7 | **void assertNotSame(object1, object2)**  The assertNotSame() method tests if two object references do not point to the same object. |
| 8 | **void assertArrayEquals(expectedArray, resultArray);**  The assertArrayEquals() method will test whether two arrays are equal to each other. |

TestCase Class

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **int countTestCases()**  Counts the number of test cases executed by run(TestResult result). |
| 2 | **TestResult createResult()**  Creates a default TestResult object. |
| 3 | **String getName()**  Gets the name of a TestCase. |
| 4 | **TestResult run()**  A convenience method to run this test, collecting the results with a default TestResult object. |
| 5 | **void run(TestResult result)**  Runs the test case and collects the results in TestResult. |
| 6 | **void setName(String name)**  Sets the name of a TestCase. |
| 7 | **void setUp()**  Sets up the fixture, for example, open a network connection. |
| 8 | **void tearDown()**  Tears down the fixture, for example, close a network connection. |
| 9 | **String toString()**  Returns a string representation of the test case. |

TestResult Class

|  |  |
| --- | --- |
| 1 | **void addError(Test test, Throwable t)**  Adds an error to the list of errors. |
| 2 | **void addFailure(Test test, AssertionFailedError t)**  Adds a failure to the list of failures. |
| 3 | **void endTest(Test test)**  Informs the result that a test was completed. |
| 4 | **int errorCount()**  Gets the number of detected errors. |
| 5 | **Enumeration<TestFailure> errors()**  Returns an Enumeration for the errors. |
| 6 | **int failureCount()**  Gets the number of detected failures. |
| 7 | **void run(TestCase test)**  Runs a TestCase. |
| 8 | **int runCount()**  Gets the number of run tests. |
| 9 | **void startTest(Test test)**  Informs the result that a test will be started. |
| 10 | **void stop()**  Marks that the test run should stop. |

TestSuite Class

important methods of **TestSuite** class are as follows −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **void addTest(Test test)**  Adds a test to the suite. |
| 2 | **void addTestSuite(Class<? extends TestCase> testClass)**  Adds the tests from the given class to the suite. |
| 3 | **int countTestCases()**  Counts the number of test cases that will be run by this test. |
| 4 | **String getName()**  Returns the name of the suite. |
| 5 | **void run(TestResult result)**  Runs the tests and collects their result in a TestResult. |
| 6 | **void setName(String name)**  Sets the name of the suite. |
| 7 | **Test testAt(int index)**  Returns the test at the given index. |
| 8 | **int testCount()**  Returns the number of tests in this suite. |
| 9 | **static Test warning(String message)**  Returns a test which will fail and log a warning message. |

Execution Procedure

the execution procedure of methods in JUnit, which defines the order of the methods called

import org.junit.After;   
import org.junit.AfterClass;   
   
import org.junit.Before;   
import org.junit.BeforeClass;   
   
import org.junit.Ignore;   
import org.junit.Test;   
   
public class ExecutionProcedureJunit {   
   
   //execute only once, in the starting    
   @BeforeClass   
   public static void beforeClass() {   
      System.out.println("in before class");   
   }   
   
   //execute only once, in the end   
   @AfterClass   
   public static void  afterClass() {   
      System.out.println("in after class");   
   }   
   
   //execute for each test, before executing test   
   @Before   
   public void before() {   
      System.out.println("in before");   
   }   
   
   //execute for each test, after executing test   
   @After   
   public void after() {   
      System.out.println("in after");   
   }   
   
   //test case 1   
   @Test   
   public void testCase1() {   
      System.out.println("in test case 1");   
   }   
   
   //test case 2   
   @Test   
   public void testCase2() {   
      System.out.println("in test case 2");   
   }   
}

Next, create a java class file named **TestRunner.java** in C:\>JUNIT\_WORKSPACE to execute annotations.

import org.junit.runner.JUnitCore;   
import org.junit.runner.Result;   
import org.junit.runner.notification.Failure;   
   
public class TestRunner {   
   public static void main(String[] args) {   
      Result result = JUnitCore.runClasses(ExecutionProcedureJunit.class);   
   
      for (Failure failure : result.getFailures()) {   
         System.out.println(failure.toString());   
      }   
   
      System.out.println(result.wasSuccessful());   
   }   
}

Verify the output.

in before class   
in before   
in test case 1   
in after   
in before   
in test case 2   
in after   
in after class 

What is a Unit Test Case ?

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected.

What is a Mock Object?

mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.

Explain unit testing using Mock Objects.

The common coding style for testing with mock objects is to −

* Create instances of mock objects.
* Set state and expectations in the mock objects.
* Invoke domain code with mock objects as parameters.
* Verify consistency in the mock objects.

How do you test a "private" method?

When a method is declared as "private", it can only be accessed within the same class. So there is no way to test a "private" method of a target class from any test class. Hence you need to perform unit testing manually. Or you have to change your method from "private" to "protected".

How do you test a "protected" method?

When a method is declared as "protected", it can only be accessed within the same package where the class is defined. Hence to test a "protected" method of a target class, define your test class in the same package as the target class.

The contents of **log4j.properties** file are as follows −

# Define the root logger with appender file   
log = /usr/home/log4j   
log4j.rootLogger = DEBUG, FILE   
   
# Define the file appender   
log4j.appender.FILE=org.apache.log4j.FileAppender   
log4j.appender.FILE.File=${log}/log.out   
   
# Define the layout for file appender   
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout   
log4j.appender.FILE.layout.conversionPattern=%m%n

import org.apache.log4j.Logger;   
   
import java.io.\*;   
import java.sql.SQLException;   
import java.util.\*;   
   
public class log4jExample{   
   
   /\* Get actual class name to be printed on \*/   
   static Logger log = Logger.getLogger(log4jExample.class.getName());   
      
   public static void main(String[] args)throws IOException,SQLException{   
      log.debug("Hello this is a debug message");   
      log.info("Hello this is an info message");   
   }   
}

Logging Methods

|  |  |
| --- | --- |
| **#** | **Methods and Description** |
| 1 | **public void debug(Object message)**  It prints messages with the level Level.DEBUG. |
| 2 | **public void error(Object message)**  It prints messages with the level Level.ERROR. |
| 3 | **public void fatal(Object message)**  It prints messages with the level Level.FATAL. |
| 4 | **public void info(Object message)**  It prints messages with the level Level.INFO. |
| 5 | **public void warn(Object message)**  It prints messages with the level Level.WARN. |
| 6 | **public void trace(Object message)**  It prints messages with the level Level.TRACE. |

import org.apache.log4j.Logger;   
   
public class LogClass {   
private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);   
      
   public static void main(String[] args) {   
      
      log.trace("Trace Message!");   
      log.debug("Debug Message!");   
      log.info("Info Message!");   
      log.warn("Warn Message!");   
      log.error("Error Message!");   
      log.fatal("Fatal Message!");   
   }   
}

Debug Message!   
Info Message!   
Warn Message!   
Error Message!   
Fatal Message!

The **org.apache.log4j.Level** levels. You can also define your custom levels by sub-classing the **Level** class.

|  |  |
| --- | --- |
| **Level** | **Description** |
| ALL | All levels including custom levels. |
| DEBUG | Designates fine-grained informational events that are most useful to debug an application. |
| INFO | Designates informational messages that highlight the progress of the application at coarse-grained level. |
| WARN | Designates potentially harmful situations. |
| ERROR | Designates error events that might still allow the application to continue running. |
| FATAL | Designates very severe error events that will presumably lead the application to abort. |
| OFF | The highest possible rank and is intended to turn off logging. |
| TRACE | Designates finer-grained informational events than the DEBUG. |

How do Levels Works?

For the standard levels, we have ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.

import org.apache.log4j.\*;   
   
public class LogClass {   
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);   
      
   public static void main(String[] args) {   
      log.setLevel(Level.WARN);   
   
      log.trace("Trace Message!");   
      log.debug("Debug Message!");   
      log.info("Info Message!");   
      log.warn("Warn Message!");   
      log.error("Error Message!");   
      log.fatal("Fatal Message!");   
   }   
}

Warn Message!   
Error Message!   
Fatal Message!

Setting Levels using Configuration File

log4j provides you configuration file based level setting which sets you free from changing the source code when you want to change the debugging level.

Following is an example configuration file which would perform the same task as we did using the **log.setLevel(Level.WARN)** method in the above example.

# Define the root logger with appender file   
log = /usr/home/log4j   
log4j.rootLogger = WARN, FILE   
   
# Define the file appender   
log4j.appender.FILE=org.apache.log4j.FileAppender   
log4j.appender.FILE.File=${log}/log.out   
   
# Define the layout for file appender   
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout   
log4j.appender.FILE.layout.conversionPattern=%m%n 

Let us now use our following program −

import org.apache.log4j.\*;   
   
public class LogClass {   
   
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);   
      
   public static void main(String[] args) {   
      
      log.trace("Trace Message!");   
      log.debug("Debug Message!");   
      log.info("Info Message!");   
      log.warn("Warn Message!");   
      log.error("Error Message!");   
      log.fatal("Fatal Message!");   
   }   
}

Now compile and run the above program and you would get following result in **/usr/home/log4j/log.out** file −

Warn Message!   
Error Message!   
Fatal Message! 