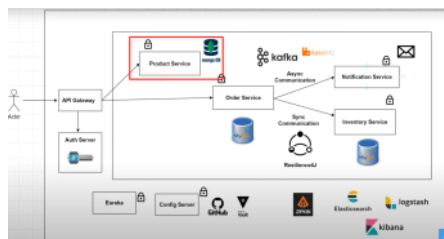


05 August 2024 10:16 AM

- **Product Service** - Create and View Products, acts as Product Catalog.
- **Order Service** - Can Order Products.
- **Inventory Service** - Can check if product is in stock or not.
- **Notification Service** - Can send notifications, after order is placed.
- **Order Service, Inventory Service and Notification Service** are going to interact with each other.



**Varun Gupta**  
Software Developer | Industry  
Relations  
Kansas City – Innovations 1024,  
Building 1, Floor 7  
8779 Hillcrest Road, Kansas City,  
MO 64138

### Explanation

- **@Document(collection = "users"):** This annotation tells Spring Data MongoDB that instances of the User class should be stored in the MongoDB collection named users. If you don't specify a collection name, Spring Data MongoDB will use the class name (in this case, User) as the default collection name.
- **@Id:** The @Id annotation is used to mark the field that will be used as the identifier for documents in MongoDB. This field will be mapped to the `_id` field in MongoDB.

1. **Mapping:** The `@Document` annotation helps map your Java class to a MongoDB collection.
2. **Collection Name:** You can specify the collection name using the collection attribute in the `@Document` annotation.
3. **CRUD Operations:** With Spring Data MongoDB, you can perform Create, Read, Update, and Delete operations on documents easily.

- **@RequestBody** is used in Spring MVC to bind the HTTP request body to a method parameter in a controller.
- It automatically deserializes JSON or XML responses into Java objects.
- **Usage:** Apply it to method parameters to handle POST or PUT requests when data sent in the request body by the client.

- 1 First create a product request class (dto) which is responsible for handling data sent by client for creating a product.
- 2 On controller use @RequestBody along with class product request which will map client data with that class.
- 3 Now create a service class for it's implementation.
- 4 On service class build a object of Product class (model) by taking data from dto product request class.
- 5 Use constructor injection to inject product repository interface
- 6 use Productrepo.save() method to save Product object in DB.
- 7 Use slf4j for the logging.
- 8 Now in controller method call the service class method which you have created. (also do constructor injection for service class).

**CascadeType.PERSIST**: When an entity is persisted (saved), this cascade type also persists all associated entities.  
**CascadeType.MERGE**: When an entity is merged (updated), this cascade type also merges all associated entities.  
**CascadeType.REMOVE**: When an entity is removed (deleted), this cascade type also removes all associated entities.  
**CascadeType.REFRESH**: When an entity is refreshed (refreshed from the database), this cascade type also refreshes all associated entities.  
**CascadeType.DETACH**: When an entity is detached (from the persistence context), this cascade type also detaches all associated entities.  
**CascadeType.ALL**: For all Operations above

- **AUTO:** Suitable for most cases, especially if you are unsure which generation strategy to use or if you want the JPA provider to handle it for you.
- **IDENTITY:** Preferred when working with databases that support auto-increment columns (e.g., MySQL, PostgreSQL). It's a straightforward approach but can have performance implications in some scenarios (e.g., high insert throughput).
- **SEQUENCE:** Ideal for databases that support sequences (e.g., Oracle, PostgreSQL). It can offer better performance and flexibility, especially for high-volume insert operations.
- **TABLE:** Used when sequences or identity columns are not available. It's a more complex and less efficient approach compared to sequences but provides an alternative where other methods are not feasible.

## Key Concepts

- 1. Transaction Management:**
  - A transaction is a sequence of operations performed as a single logical unit of work. Transactions ensure data integrity and consistency by making sure that a group of operations either all succeed or all fail.
- 2. Declarative Transactions:**
  - Using `@Transactional`, you can manage transactions declaratively, which means you can specify transaction management rules using annotations rather than programmatic code.

- **Class-Level:** The transaction settings apply to all public methods within the class.

New Section 1 Page 1

```
}
```

## 2. Transactional Attributes:

- o **propagation:** Defines how transactions should behave in relation to existing transactions. Common values include:
  - **REQUIRED:** The default. If a transaction exists, the method will join it. If not, a new transaction will be started.
  - **REQUIRES\_NEW:** Always starts a new transaction, suspending any existing transaction.
  - **NESTED:** Executes within a nested transaction if an existing transaction exists.

```
java
Copy code
@Transactional(propagation = Propagation.REQUIRES_NEW)publicvoidperformTask(){
    // New transaction will be started
}
```

- o **isolation:** Defines the level of isolation for the transaction, which affects how transaction changes are visible to other transactions. Common levels include:
  - **READ\_COMMITTED:** Ensures that any data read is committed at the moment it is read.
  - **REPEATABLE\_READ:** Ensures that if data is read multiple times, the results will be the same.
  - **SERIALIZABLE:** Ensures that transactions are executed in a serial manner, avoiding any possible anomalies.

```
java
Copy code
@Transactional(isolation = Isolation.SERIALIZABLE)publicvoidperformTask(){
    // Transaction with serializable isolation level
}
```

- o **timeout:** Specifies the timeout for the transaction. If the transaction takes longer than the specified timeout, it will be rolled back.

```
java
Copy code
@Transactional(timeout = 30)publicvoidperformTask(){
    // Transaction will be rolled back if it takes longer than 30 seconds
}
```

- o **readOnly:** Indicates whether the transaction is read-only. This can help optimize performance by allowing the transaction manager to perform optimizations.

```
java
Copy code
@Transactional(readOnly = true)publicList<User> getUsers(){
    // Read-only transaction
}
```

- o **rollbackFor:** Specifies which exceptions should trigger a rollback. By default, transactions are rolled back for unchecked exceptions.

```
java
Copy code
@Transactional(rollbackFor = { SQLException.class })publicvoidperformTask(){
    // Rollback for SQLExceptions
}
```

- o **noRollbackFor:** Specifies which exceptions should not trigger a rollback.

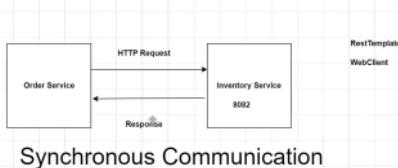
```
java
Copy code
@Transactional(noRollbackFor = { CustomException.class })publicvoidperformTask(){
    // No rollback for CustomException
}
```

## Key Points

- **Automatic Rollback:** By default, transactions are rolled back in case of unchecked exceptions (subclasses of RuntimeException). You can customize this behavior using `rollbackFor` and `noRollbackFor`.
- **Transaction Propagation:** The propagation setting determines how transactions are managed when a method is called within another transaction.
- **Isolation Levels:** Different isolation levels control the visibility of changes made by one transaction to other concurrent transactions, impacting consistency and performance.
- **Read-Only Transactions:** Using `readOnly = true` can optimize database operations, particularly for queries that should not modify data.
- **Transaction Management in Spring Boot:** Spring Boot applications automatically configure transaction management, and you can use `@Transactional` with various data access strategies (JPA, JDBC, etc.).

Using `@Transactional` helps in managing complex transaction requirements declaratively, ensuring data consistency and integrity while keeping the code clean and maintainable.

## Inter Process Communication



## Synchronous Communication

## @Bean

In Spring Boot, the `@Bean` annotation is used to indicate that a method produces a bean to be managed by the Spring container. Beans are objects that Spring manages, which are often used for dependency injection.

Here's a simple explanation:

1. **Defining Beans:** You use the `@Bean` annotation inside a `@Configuration` class to define beans. The method annotated with `@Bean` returns an object that Spring will treat as a bean.
2. **Managing Beans:** Once defined, Spring will manage the lifecycle of this bean. This includes creating, initializing, and injecting it where needed.
3. **Dependency Injection:** Other components (beans) can have this bean injected into them automatically by Spring.

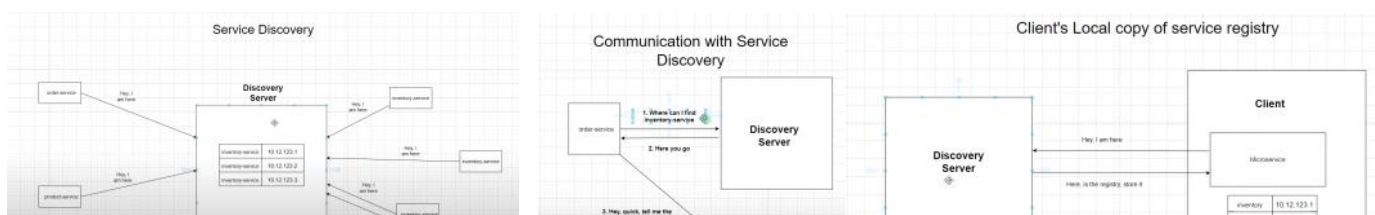
`@Bean`

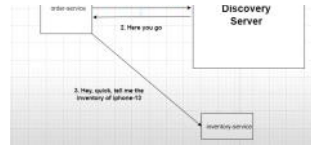
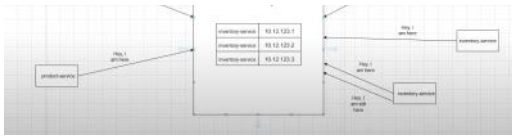
```
public MyService myService() {           //Now In our complete application if we want to use Myservice object then we can just use myService.
    return new MyService();
}
```

In Spring Boot, **WebClient** is a non-blocking, reactive client used for making HTTP requests. It's part of the Spring WebFlux module and provides a more modern and flexible way to perform HTTP operations compared to the older `RestTemplate`. It provides a fluent API and supports advanced features like streaming and error handling.

```
List<OrderItems> orderItemsList = orderRequest.getOrderItemsDtoList()
    .stream().map(orderItemsDto -> mapToDto(orderItemsDto))
    .collect(Collectors.toList());
```

In brief, this code snippet converts a list of `OrderItemsDto` objects into a list of `OrderItems` objects using Java Streams. Each `OrderItemsDto` is transformed into an `OrderItems` object by the `mapToDto` method, and the results are gathered into a new list.





Client holds the copy of services from discovery server to it's local server so that even if the Discover server is down client can connect with services locally.

## Discovery Server:

Using Spring cloud Netflix eureka dependency to enable our Discovery Server.

## What is Eureka Server?

Eureka Server is a service registry that allows microservices to register themselves and discover other services. It's used in microservices architectures for service discovery, enabling services to find and communicate with each other without hardcoding IP addresses or URLs.

@EnableEurekaServer to turn your Spring Boot application into a Eureka service registry, enabling service discovery for microservices. Ensure you configure it correctly and add the necessary dependencies to get it running.

Default port for Eureka Server: 8761

// this we use it for our client services like orderService or InventoryService

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

//This is for Discovery or Eureka Server

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

The @EnableEurekaClient annotation in Spring Boot is used to make a Spring Boot application act as a client to a Eureka service registry. This means that the application will register itself with a Eureka Server and be able to discover other services registered with the same Eureka Server.

//Have to use this for Load Balancing:

For using the WebClient with Discovery, declare following bean in configuration.

```
@Bean
@LoadBalanced
public WebClient.Builder loadBalancedWebClientBuilder() {
    return WebClient.builder();
}
```

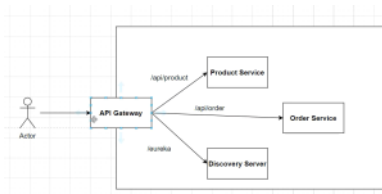
Autowire this in class where you are calling other service,

```
@Autowired
private WebClient.Builder webClientBuilder;
```

And, instead of webClient.get(), use webClientBuilder.build().get()

## API Gateway:

An **API Gateway** is a server that acts as an entry point for requests to a system of services. It sits between clients and backend services and handles tasks like routing requests, load balancing, authentication, authorization, and more. API Gateways are commonly used in microservices architectures to provide a unified entry point for client requests and to manage various cross-cutting concerns.



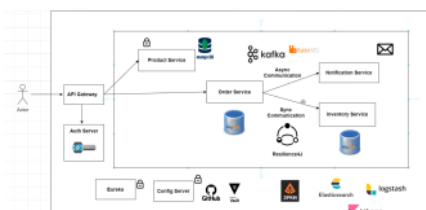
API Gateway act as a gatekeeper to route the user to a microservice. Also take cares of authorization, authentication and load balancing.

Once we have configured API Gateway we don't need to remember the ports on which each service is running. Just have to use the single port which will be of Api Gateway along with correct url of the service which we have to call..

## Logging Levels

In Spring Boot, logging levels determine the amount of detail logged by the application. The commonly used logging levels, from the most detailed to the least detailed, are:

1. **TRACE:** The most granular level of logging. It provides detailed information, typically used for debugging complex issues. This level is generally not enabled in production environments due to the high volume of logs it generates.
2. **DEBUG:** Provides more detailed information than INFO, often used for diagnosing problems and understanding the flow of the application. This level is useful for debugging purposes but might be too verbose for production.
3. **INFO:** Used for informational messages that highlight the progress of the application at a high level. This level provides a balance between verbosity and usefulness and is often used in production environments to track the application's general operation.
4. **WARN:** Indicates potential issues or important warnings that do not necessarily disrupt the application's functionality but might require attention. It helps in identifying potential problems before they become critical.
5. **ERROR:** Indicates serious problems that have caused or might cause the application to fail. This level is used for logging error messages and exceptions that need immediate attention.
6. **FATAL** (not commonly used in Spring Boot but present in other logging frameworks): Represents severe errors that will lead to application termination. In many logging systems, ERROR is sufficient for this purpose.

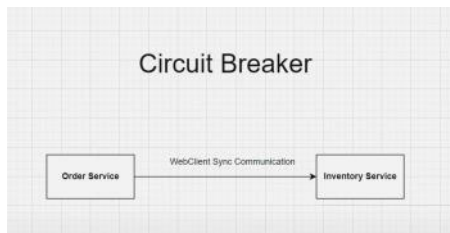


## KeyCloak:

Keycloak is an open-source identity and access management solution by Red Hat. It simplifies authentication and authorization for applications.

### Key Features:

- **Single Sign-On (SSO):** Access multiple applications with one login.
- **User Federation:** Integrates with existing LDAP/Active Directory systems.
- **Identity Brokering:** Connects to external identity providers (e.g., Google, Facebook).
- **Social Login:** Allows users to log in with social media accounts.
- **User Management:** Admins can manage users, roles, and permissions.
- **Authorization:** Supports role-based and attribute-based access control.
- **Multi-Factor Authentication (MFA):** Enhances security with additional verification steps.
- **Customizable Pages:** Tailor login and registration pages to your brand



In Spring Boot, a Circuit Breaker is a design pattern used to handle failures in distributed systems and microservices architectures. It helps improve the resilience and stability of applications by preventing failures in one part of the system from cascading to other parts. The Circuit Breaker pattern is commonly used in conjunction with other resilience strategies like retries and timeouts.

Aspect	Synchronous Communication	Asynchronous Communication
Definition	Real-time interaction; both parties are engaged simultaneously.	Interaction does not happen at the same time; sender and receiver operate independently.
Interaction	Real-time; both parties must be present.	Not real-time; parties can engage at different times.
Blocking	Blocking; the sender waits for a response before proceeding.	Non-blocking; the sender can continue with other tasks while awaiting a response.
Examples	Phone calls, video calls, live chat, synchronous API calls.	Emails, SMS, forum posts, asynchronous API calls.
Pros	Immediate feedback; ideal for real-time needs.	Flexibility; better for non-urgent tasks and high loads.
Cons	Requires both parties to be available simultaneously; potential for idle waiting.	Response times can be unpredictable; requires additional mechanisms for managing delays.
Use Case	Live customer support, instant messaging, real-time data updates.	Project updates, email communication, forum discussions.
Complexity	Simpler for real-time interactions; can become complex with high concurrency.	Can handle delays and high loads better; may require more complex systems for managing responses.

### Circuit Breaker Pattern:

- **Closed State:** The circuit breaker is "closed" when all operations are functioning normally. Requests pass through and are executed as usual.
- **Open State:** If the failure rate exceeds a certain threshold, the circuit breaker transitions to the "open" state. In this state, requests are automatically failed without being executed, allowing the system time to recover.
- **Half-Open State:** After a defined period, the circuit breaker transitions to the "half-open" state. A limited number of test requests are allowed through to check if the issue has been resolved. If these requests succeed, the circuit breaker moves back to the "closed" state; otherwise, it returns to the "open" state.

Using Resilience4j for Circuit breaker:

@CircuitBreaker is an annotation used to implement the Circuit Breaker pattern, which helps manage failures in distributed systems and improves resilience. This annotation is provided by libraries like Resilience4j and previously Hystrix (though Hystrix is now deprecated).

Using @TimeLimiter in Spring Boot with Resilience4j allows you to easily manage long-running tasks and improve your application's resilience. You can customize the timeout duration and handle any exceptions that may occur when a method exceeds the specified time limit.

```
actuator:
  management:
    health:
      circuitbreakers:
        enabled: true
    endpoints:
      web:
        exposure:
          include: "*"
        health:
          show-details: always

#resilience4j Properties
resilience4j.circuitbreaker.instances.inventory.registerHealthIndicator=true
resilience4j.circuitbreaker.instances.inventory.event-consumer-buffer-size=10
resilience4j.circuitbreaker.instances.inventory.slidingWindowSize=COUNT_BASED
resilience4j.circuitbreaker.instances.inventory.slidingWindowMaxSize=5
resilience4j.circuitbreaker.instances.inventory.failureRateThreshold=50
resilience4j.circuitbreaker.instances.inventory.waitBeforeIngestingStats=5s
resilience4j.circuitbreaker.instances.inventory.permittedNumberOfCallsInOpenState=3
resilience4j.circuitbreaker.instances.inventory.automaticTransitionFromOpenToHalfOpenEnabled=true
```

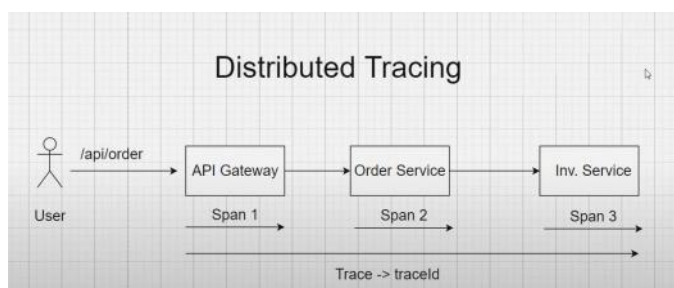
We have to configure Circuit Breaker and Actuators on application.properties.

## Spring Boot Actuators Quick Note

**Purpose:** Actuators in Spring Boot provide built-in endpoints to monitor and manage application health, performance, and configuration.

### Key Features:

- **Health Checks:** /actuator/health - Monitors application health status.
- **Metrics:** /actuator/metrics - Provides performance metrics like JVM memory usage.
- **Environment Info:** /actuator/env - Displays configuration properties and environment variables.
- **Application Info:** /actuator/info - Shows application metadata (e.g., version).
- **Beans:** /actuator/beans - Lists Spring beans in the application context.
- **Loggers:** /actuator/loggers - Allows dynamic modification of logging levels.
- **Thread Dump:** /actuator/heapdump - Provides a thread dump for diagnosing performance issues.
- **HTTP Traces:** /actuator/httptrace - Displays recent HTTP request/response traces.

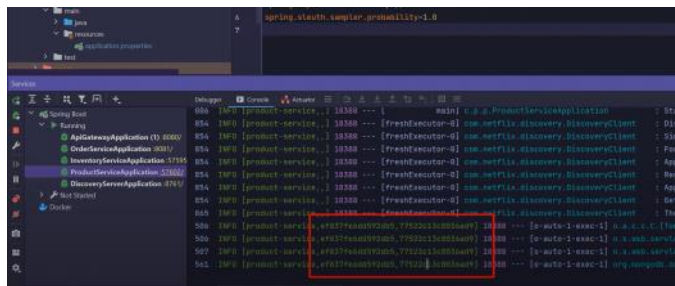


Distributed tracing is a critical aspect of microservices architecture that helps developers understand and monitor the flow of requests across different services. It allows you to trace requests as they propagate through various services, providing visibility into performance bottlenecks, errors, and overall system behavior. It captures timing data, which helps identify latency issues and trace the execution path.

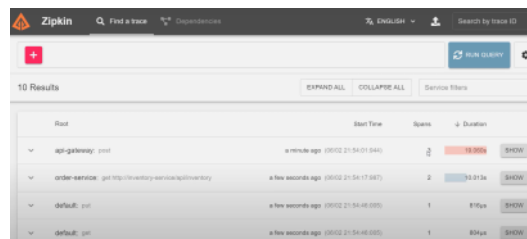
## Why Use Distributed Tracing?

- **Performance Monitoring:** Identify slow services and performance bottlenecks.
- **Error Tracking:** Quickly find the source of errors in a complex system.
- **Dependency Analysis:** Understand how services interact with each other.
- **Latency Optimization:** Optimize individual service performance by understanding their role in the request flow.

In a Spring Boot application, you can implement distributed tracing using tools like **Spring Cloud Sleuth** and **Zipkin** or **OpenTelemetry**.



The screenshot shows an IDE with a log viewer displaying application logs. A specific log entry is highlighted with a red box, showing the following details: `[INFO] [product-service, 1] 16380 --- [freshExecutor-8] com.netflix.discovery.DiscoveryClient : Sta`. The log entry includes the service name, span ID, and trace ID, which are essential for distributed tracing.



The screenshot shows the Zipkin web interface with a list of traces. The table displays the following information:

Trace	Start Time	Spans	Duration
api-gateway: post	a minute ago (2022-11-04 01:54:01)	0	14.00s
order-service: get http://inventory-service/api/inventory	a few seconds ago (2022-11-04 01:54:07)	2	14.01s
default: post	a few seconds ago (2022-11-04 01:54:08)	1	815us
default: post	a few seconds ago (2022-11-04 01:54:08)	1	804us

We can see the additional logs [service\_name, span\_id, trace\_id].

# Kafka

16 August 2024 06:50 AM

**Apache Kafka**, it's a distributed streaming platform used for building real-time data pipelines and streaming applications. Kafka is designed to handle high throughput and low-latency data processing. Here are some key concepts and features: More from [What is Kafka, and How Does it Work? A Tutorial for Beginners \(confluent.io\)](#)

## Key Concepts

- Producer:** An application that publishes messages to Kafka topics. Producers send data to Kafka brokers.
- Consumer:** An application that subscribes to topics and processes messages. Consumers read data from Kafka brokers.
- Topic:** A category or feed name to which records are sent. Topics are split into partitions for parallel processing. Producers write data to topics, and consumers read data from topics. Topics are central to Kafka's publish/subscribe model and are used to organize and manage the flow of messages.
- Partition:** A topic is divided into partitions. Each partition is an ordered, immutable sequence of records. Partitions allow Kafka to scale horizontally.
- Broker:** A Kafka server that stores data and serves clients. Kafka brokers form a Kafka cluster, and data is replicated across multiple brokers for reliability.
- ZooKeeper:** Kafka uses ZooKeeper to manage distributed brokers and maintain metadata. However, recent versions of Kafka are moving towards KRaft mode, which aims to eliminate the need for ZooKeeper.
- Offset:** A unique identifier for each record within a partition. Consumers use offsets to keep track of their position within a topic.

## Use Cases

- Real-time Analytics:** Kafka is used to build systems that process and analyze data in real-time.
- Event Sourcing:** It can be used as the backbone of an event-driven architecture, where events are published and consumed across different services.
- Log Aggregation:** Kafka is often used to collect and aggregate logs from various sources.

## Integration with Spring Boot

If you're using Spring Boot, you can integrate Kafka using the Spring Kafka project. It provides a set of abstractions and configuration options to work with Kafka easily, including:

- KafkaTemplate:** For sending messages to Kafka topics.
- @KafkaListener:** For consuming messages from Kafka topics.
- Kafka Streams:** For building stream processing applications with Kafka

## Integrating Apache Kafka with Spring Boot on Windows (No Docker)

### 1. Set Up Kafka

- Download and Extract Kafka:**
- Start ZooKeeper:**
- Start Kafka Broker:**
- Create a Kafka Topic:**  
`bin\windows\kafka-topics.bat --create --topic my-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1`

### 2. Integrate Kafka with Spring Boot

- Add Kafka Dependencies:**
  - Maven:** Add spring-kafka to pom.xml.
  - Gradle:** Add spring-kafka to build.gradle.
- Configure Kafka Properties:**
  - In application.yml, set Kafka bootstrap-servers, consumer, and producer settings.
- Create Kafka Producer and Consumer Services:**
  - Producer:** Implement a service to send messages to Kafka.
  - Consumer:** Implement a service to listen and process messages from Kafka.

### 3. Test Your Setup

- Run Your Spring Boot Application.**
- Send Test Messages Using the Producer Service.**
- Verify Message Processing with the Consumer Service.**

### 4. Troubleshooting and Monitoring

- Check Logs:** For any errors in Kafka and ZooKeeper.
- Verify Configuration:** Ensure correct setup and running status.
- Check Network:** Confirm no issues with connectivity.

**Kafka Template Class:** Using this class Producers (Order Service) send messages to consumers (Notification Service). It simplifies the process of producing messages and provides a higher-level abstraction over the Kafka producer API.

## What is KafkaTemplate?

KafkaTemplate is a high-level abstraction provided by Spring Kafka that simplifies sending messages to Kafka topics. It encapsulates the Kafka producer API and provides a more straightforward way to publish messages to Kafka.

## Key Features of KafkaTemplate:

- Message Sending:** It allows you to send messages to Kafka topics with various methods, including synchronous and asynchronous sending.
- Configuration:** KafkaTemplate is highly configurable through properties, making it adaptable to different Kafka configurations.
- Type Safety:** It provides type-safe methods for sending messages, which can include key-value pairs where both key and value types are specified.
- Error Handling:** It supports error handling for cases where message sending fails, which can be configured to retry or log errors.

## Key Methods:

- send:**
  - send(String topic, V data):** Sends a message with a value to a specified topic.
  - send(String topic, K key, V data):** Sends a message with a key and value to a specified topic.
  - send(String topic, Integer partition, K key, V data):** Sends a message to a specific partition.
- sendAndReceive:**
  - sendAndReceive(String topic, V data):** Sends a message and waits for a reply (requires additional configuration for request-response patterns).

**@KafkaListener** is an annotation in Spring Kafka that simplifies the process of consuming messages from Kafka topics. It is used to create Kafka consumers in a Spring Boot application.

It takes argument as a topic and will listen to that topic.

## Code Flow:

Whenever we receive an order via orderService (Producer) and order gets placed successfully then orderService will raise an event i.e OrderPlacedEvent and we can place the object of OrderPlacedEvent as a message inside the Kafka Broker. Notification Service (Consumer) will consume the message and process it accordingly.

- Add docker compose file to your project so that broker and zookeeper will run via docker.
- Add spring-kafka dependency in order service
- Add Kafka properties in application.properties of order-service

```
spring.kafka.bootstrap.servers=localhost:9092
spring.kafka.consumer.default.topic=notificationTopic
spring.kafka.producer.key.serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value.serializer=org.springframework.kafka.support.serializer.JsonSerializer
```

- In order to call Kafka Cluster via OrderService inject Kafka Template class in your OrderService. OrderPlacedEvent is just a POJO having orderNumber

```
@Service
@Transactional
public class OrderService {

    private final OrderRepository orderRepository;
    private final MongoClient mongoClientBuilder;
    private final Tracer tracer;
    private final KafkaTemplate<String, OrderPlacedEvent> kafkaTemplate;
```

- Using KafkaTemplate.send(): It is defined to send order Number to a topic NotificationTopic.

```
if (allProductsInStock) {
    orderRepository.save(order);
    kafkaTemplate.send("notificationTopic", order.getOrderNumber());
    return "Order Placed Successfully";
} else {
    throw new IllegalArgumentException("Product is not in stock, please try again");
}

finally {
    inventoryService.checkout.and();
}
```

- Creating NotificationService Module:

```
package com.programming.techlab;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.kafka.annotation.KafkaListener;

@SpringBootApplication
public class NotificationServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(NotificationServiceApplication.class, args);
    }

    @KafkaListener(topics = "notificationTopic")
    public void handleNotification(OrderPlacedEvent orderPlacedEvent) {
    }
}
```

We can write our business logic inside this handleNotification function as this is The main method.

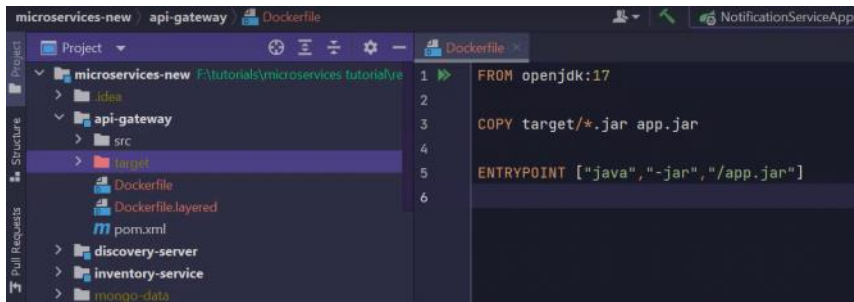
- Application.properties for Notification Service:





# Docker

18 August 2024 12:47 PM

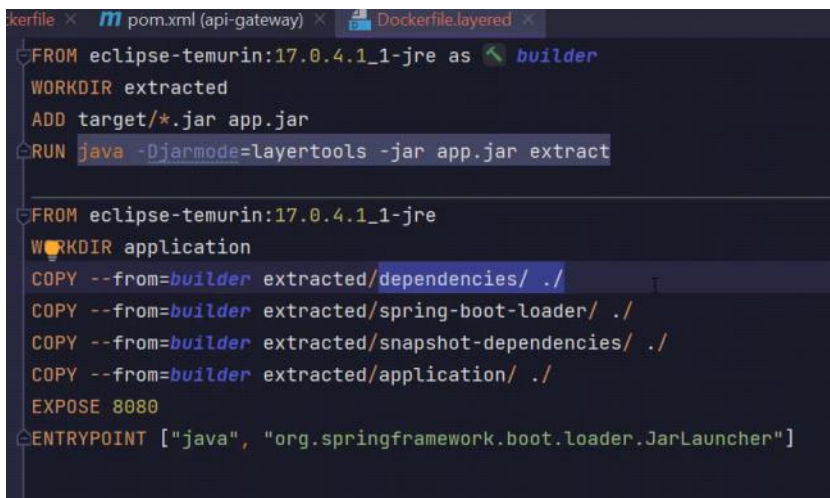


Creating docker file.



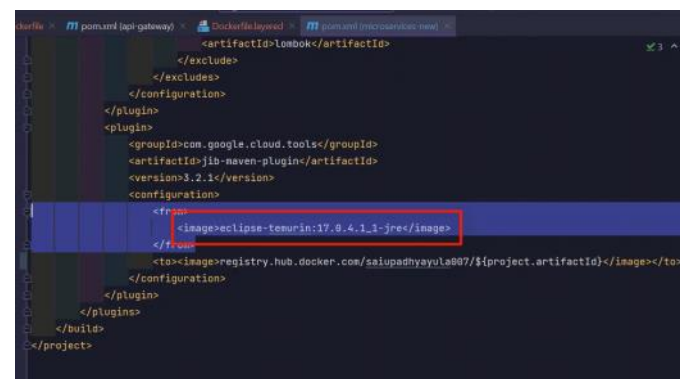
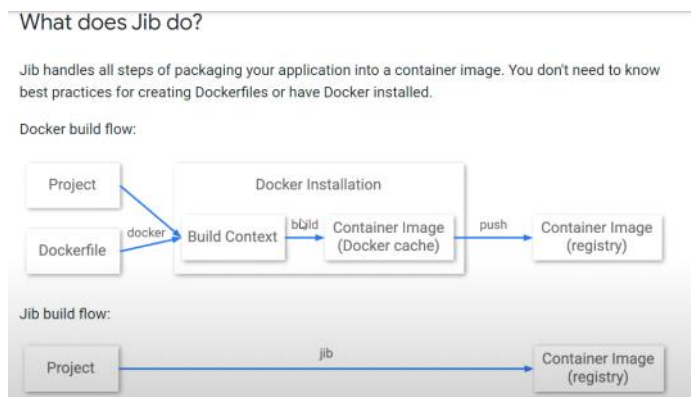
Running build command. -t stands for tag of the image

Docker images: Shows all the images in the system.

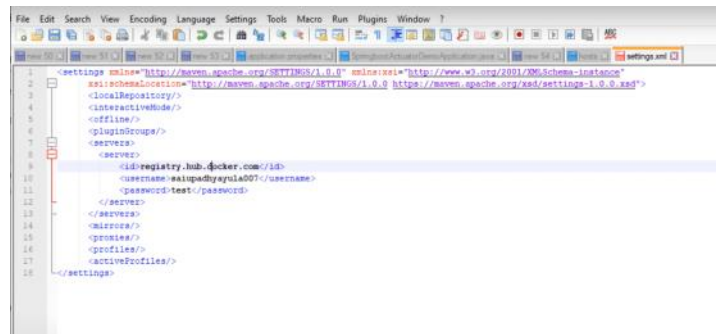
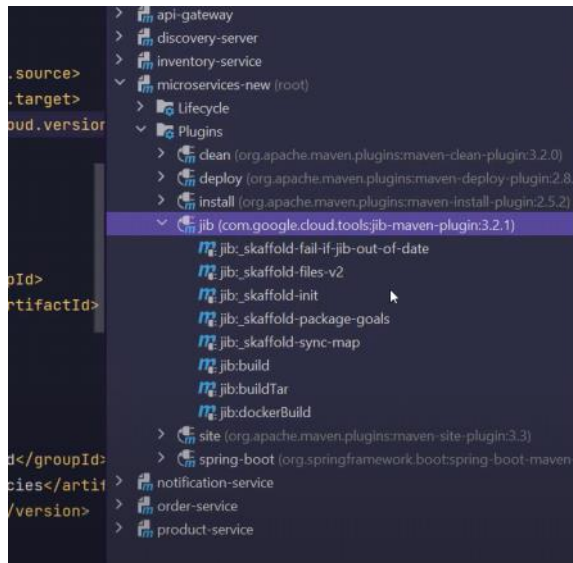


Layered Docker file (New way)

JIB is the library by Google which is used to build images without using docker file.







Not my cup of Tea. Learn Docker commands and interview Questions.