**Kafka Cluster**: Kafka is a distributed system, it acts as agent it consist of set of Broker / software, for production cluster must have 3 Broker.



**Producer**: Produce messages any in format. (JSON, XML, Text etc.) And send to Kafka cluster.

**Consumer**: Consume messages from Kafka broker.

**Kafka Broker**: it’s a Kafka server or s/w, the producer and Consumer don’t interact directly, they use Kafka broker / agent to exchange messages.

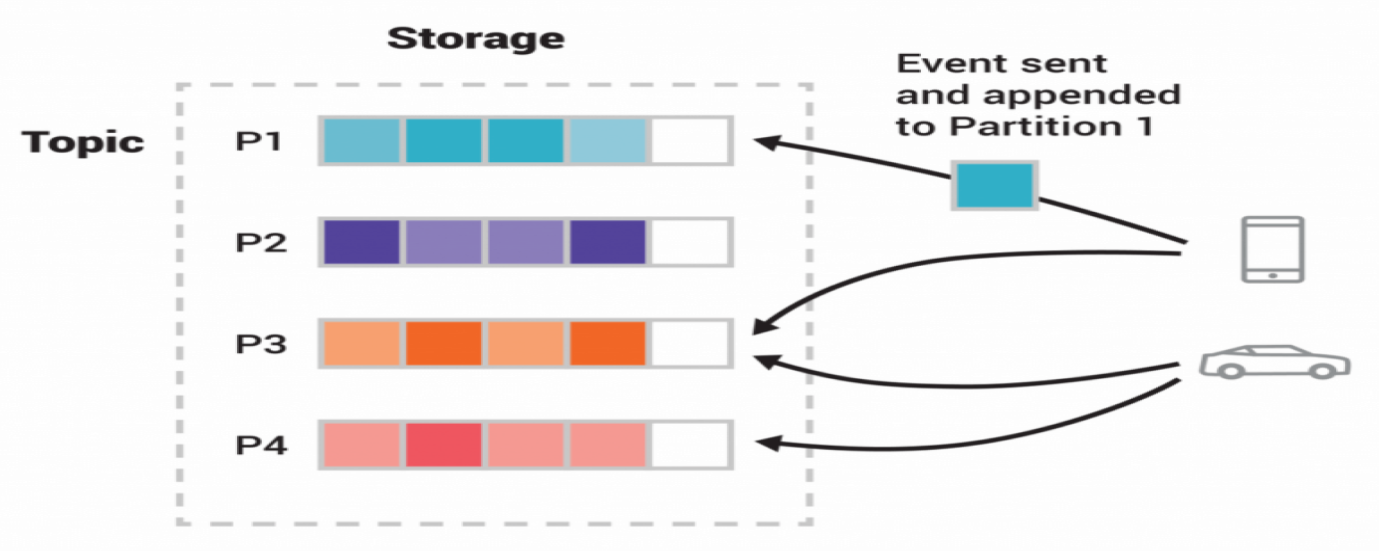
**Zookeeper**: Manages the state of the all Kafka broker in the Kafka cluster. It also maintains the configuration of all topics of producer and consumer.

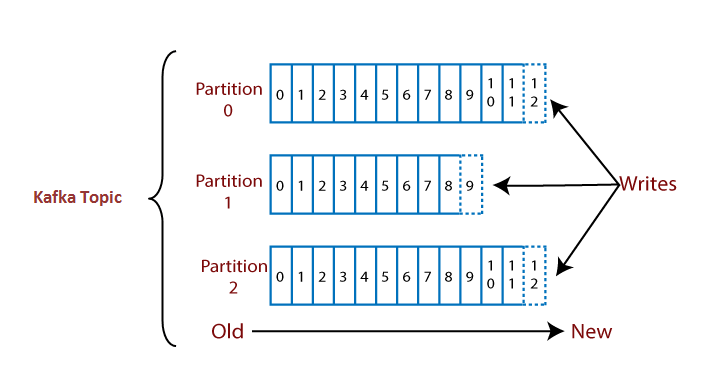
**Fault Tolerance**: If any broker gets down then automatically another broker handles the traffic as data is duplicated across the Kafka brokers in the cluster.

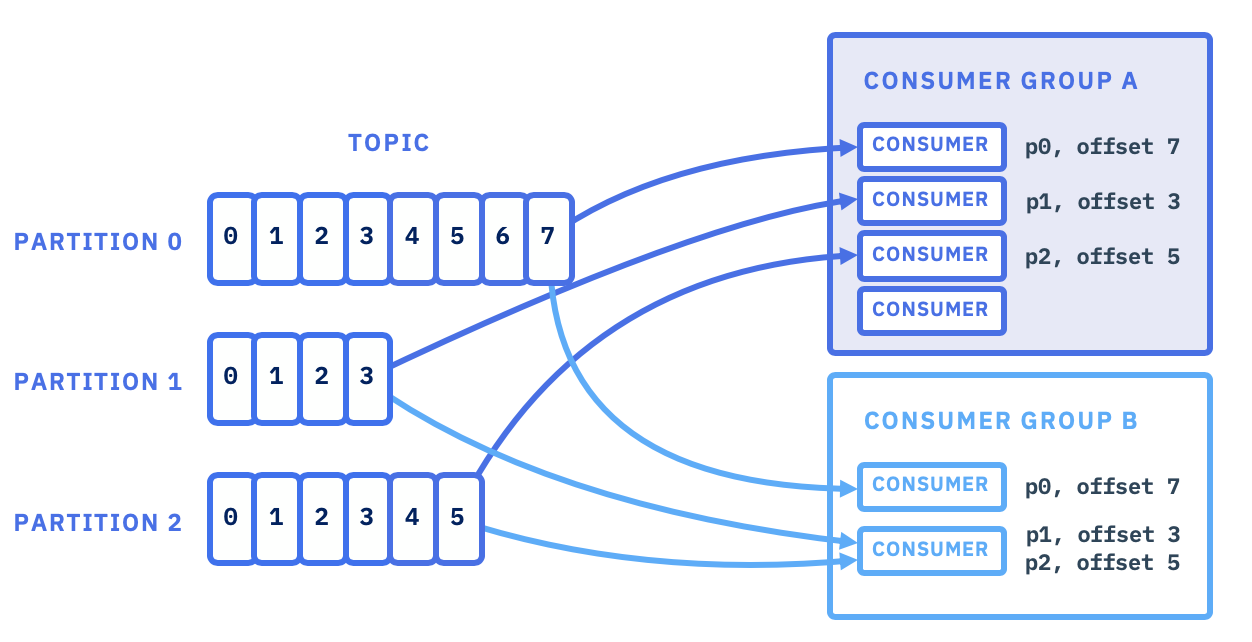
**Reference**: <https://kafka.apache.org/documentation/>

**Kafka Topic**: Is like Table in database or folder in file system, and is identified by name we can have any number of Topics in Kafka broker. Consumer subscribes the topic.

**Kafka Partition**: Topic has multiple partitions.



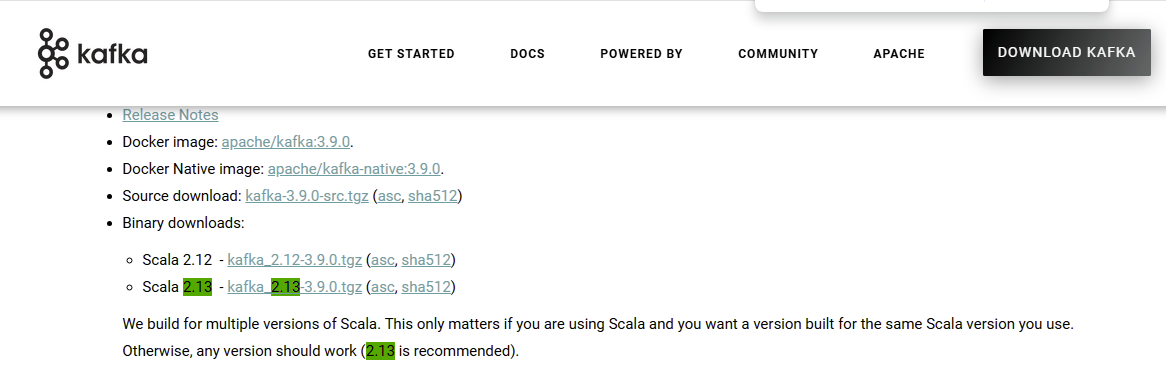
**Offset**: Is a sequence of Ids given to the messages at arrive position. Once the offset is assigned it will never changes.

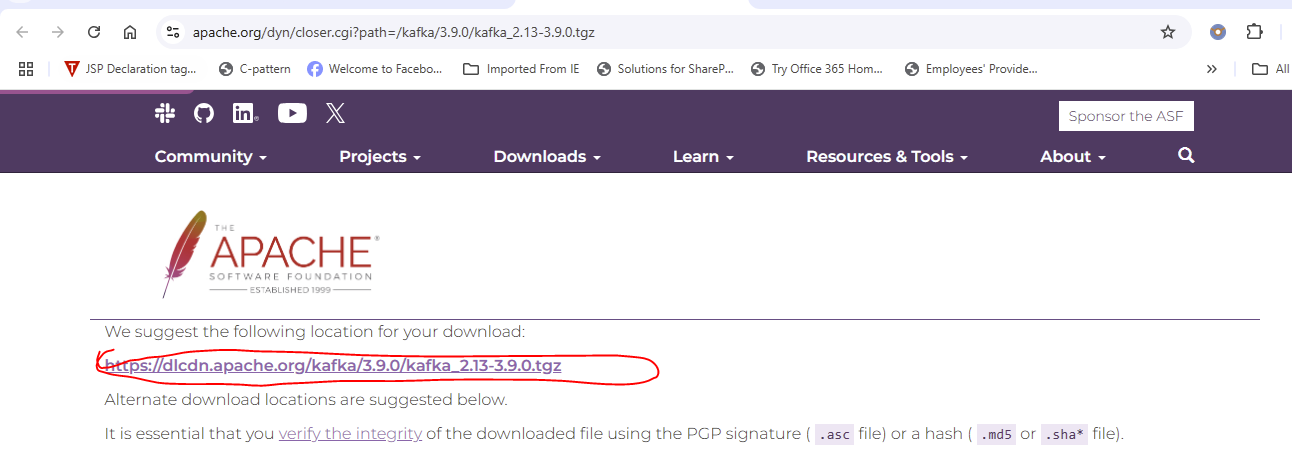
**Consumer Group**: A consumer group contains a one or more consumer’s working together to process the messages. 

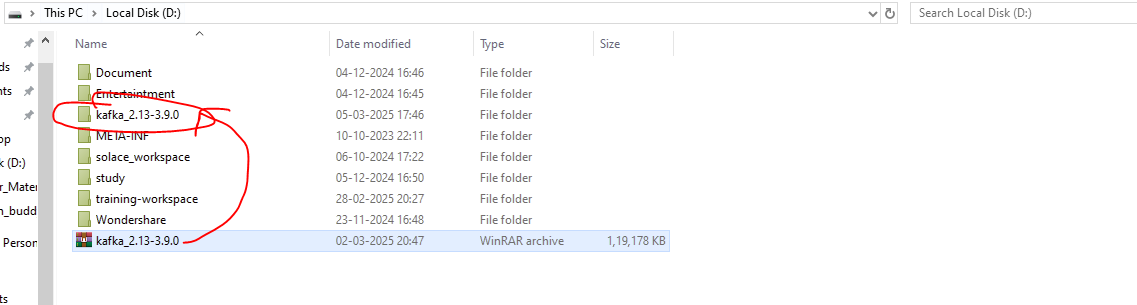
Can multiple consumers consume messages from same partition?

|  |
| --- |
| Within same group: **NO**  Two consumers (Consumer 1, 2) within the same group (Group 1) CAN NOT consume the same message from partition (Partition 0).  Across different groups: **YES**  Two consumers in two groups (Consumer 1 from Group 1, Consumer 1 from Group2) CAN consume the same message from partition (Partition 0). |

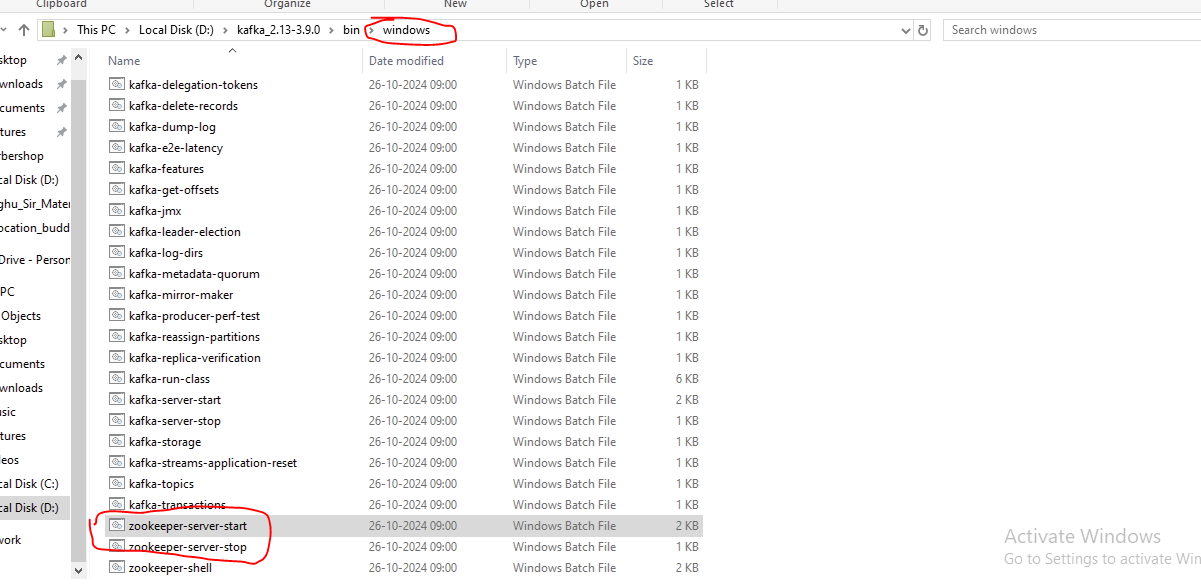
**How to install Kafka**: <https://kafka.apache.org/downloads>





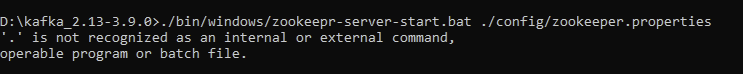
Extract Here

You will get multiple windows batch file as utility D:\kafka\_2.13-3.9.0\bin\windows



**Start Zookeeper:**

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties



**Check status if Zookeeper started**

.\bin\windows\zookeeper-shell.bat status or find log binding to port 0.0.0.0/0.0.0.0:2181

**Start Kafka broker**:

.\bin\windows\kafka-server-start.bat .\config\server.properties



**Create Topic**

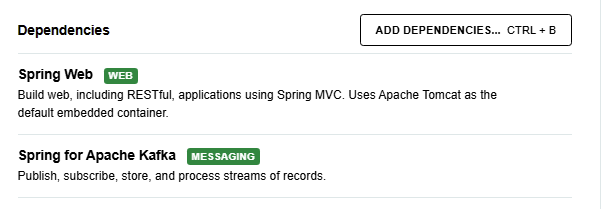
.\bin\windows\kafka-topics.bat --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

**List Topic**

.\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092

**List messages from topic**

.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic json\_topic --from-beginning

**Create Spring Application**

**Create consumer and producer Application**

application.properties

|  |
| --- |
| #specify the Kafka broker(s) that the consumer will connect to for consuming message  spring.kafka.consumer.bootstrap-servers: localhost:9092 OR  spring.kafka.consumer.bootstrap-servers: broker1:9092,broker2:9092,broker3:9092 spring.kafka.consumer.group-id: myGroup spring.kafka.consumer.auto-offset-reset:earliest  #Deserilise Key into Message spring.kafka.consumer.key-deserializer: org.apache.kafka.common.serialization.StringDeserializer  #Deserilise value into Message spring.kafka.consumer.value-deserializer: org.apache.kafka.common.serialization.StringDeserializer  spring.kafka.producer.bootstrap-servers: localhost:9092 spring.kafka.producer.key-serializer: org.apache.kafka.common.serialization.StringSerializer spring.kafka.producer.value-serializer: org.apache.kafka.common.serialization.StringSerializer |

When a producer sends a message to Kafka, it sends:

* A key (optional)
* A value (the message content)
* Both are sent as byte arrays

Kafka doesn't know or care what your data is—it just stores and transports bytes.

**Why deserializers?**

When a consumer reads data from Kafka, it gets:

* A byte array for the key
* A byte array for the value

To turn these bytes into something useful in your code (like a String, a JSON object, or a custom Java class), you need to **deserialize** them.

**Create Kafka Topic**

|  |
| --- |
| @Configuration public class KafkaConfig {  @Bean  public NewTopic rlbinTopic() {  return TopicBuilder.name("rlbIn").build();  }} |

**List Kafka Topic**:

.\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092

**Create Producer**

|  |
| --- |
| @Service public class KafkaProducer {  private KafkaTemplate<String, String> kafkaTemplate;   public KafkaProducer(KafkaTemplate<String, String> kafkaTemplate) {  this.kafkaTemplate = kafkaTemplate;  }   public void send(String message) {  kafkaTemplate.send("rlbIn", message);  } } |

**Create Rest API to publish Message**

|  |
| --- |
| @RestController @RequestMapping("/kafka/v1") public class KafkaController {   private KafkaProducer kafkaProducer;   public KafkaController(KafkaProducer kafkaProducer) {  this.kafkaProducer = kafkaProducer;  }   @RequestMapping("/publish")  public ResponseEntity<String> publish(@RequestParam("message") String msg) {  kafkaProducer.send(msg);  return new ResponseEntity<>("success", HttpStatus.*CREATED*);  } } |

**List Message:** .\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic rlbIn --from-beginning**Create Consumer**

|  |
| --- |
| @Service public class KafkaConsumer {  private static final Logger *LOGGER* = LogManager.*getLogger*(KafkaConsumer.class);   @KafkaListener(topics = "rlbIn", groupId = "myGroup")  public void consume(String msg) {  *LOGGER*.info(String.*format*("Message received : %s",msg));  } } |

**JSON Deserializer and JSON Serializer**

Spring Kafka created JsonSerializer and JsonDeserializer which can use to convert Java Object to and from JSON.

Spring Boot auto-configures a lot of Kafka properties for you, using sensible defaults. So unless you’re doing something custom, you often don’t need to set

spring.kafka.producer.value-serializer manually.

**Default Behavior**

If you’re using KafkaTemplate<String, String>, Spring Boot sees you're dealing with Strings, so it auto-applies:

|  |
| --- |
| spring.kafka.producer.key-serializer = org.apache.kafka.common.serialization.StringSerializer  spring.kafka.producer.value-serializer = org.apache.kafka.common.serialization.StringSerializer |

**When You Do Need to Set It?**

You do need to set spring.kafka.producer.value-serializer when:

* You’re sending custom Java objects
* You're using a format like JSON, Avro, or Protobuf
* You have multiple producers with different data formats

Example: Sending custom Java objects as JSON

|  |
| --- |
| spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer  spring.kafka.producer.properties.spring.json.type.mapping=yourTypeId:com.example.YourClass |

**spring.kafka.producer.properties.spring.json.type.mapping**=yourTypeId:com.example.YourClass

This is used with JsonSerializer and JsonDeserializer to help Spring Kafka map a type ID (like a string) to a specific Java class during serialization/deserialization

When Kafka messages are serialized as **JSON**, Spring Kafka’s JsonSerializer adds metadata (like the type info) into the message headers so the JsonDeserializer on the **consumer** side knows what class to turn the JSON back into.

**How it works**

Suppose you’re sending a message like:

kafkaTemplate.send("my-topic", new User("Alice", "admin")); Where User is a custom class.

|  |
| --- |
| spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer  spring.kafka.producer.properties.spring.json.type.mapping=user:com.example.User |

Now Spring knows when the type ID in headers is "user", it corresponds to com.example.User

Similarly, on the consumer side:

|  |
| --- |
| spring.kafka.consumer.value.deserializer=org.springframework.kafka.support.serializer.JsonDeserializer  spring.kafka.consumer.properties.spring.json.type.mapping=user:com.example.User  spring.kafka.consumer.properties.spring.json.trusted.packages=\* |

**spring.kafka.consumer.properties.spring.json.trusted.packages=\***

This property is used with JsonDeserializer in Spring Kafka to control what packages are allowed when deserializing JSON into Java objects.

**Why does this exist?**

Deserializing JSON can be dangerous if someone sends a malicious payload. For example, a bad actor could try to deserialize something into a class that could execute code — security risk!

To prevent that, Spring Kafka’s JsonDeserializer only allows classes from trusted packages — unless you explicitly say otherwise.

**What does \* mean?**

spring.kafka.consumer.properties.spring.json.trusted.packages=\*

"Trust **all packages**. Go ahead and deserialize any Java class if it's in the type headers."

**Is \* safe?**

Technically yes — but **use it with caution**.

* ✅ Okay in internal, tightly controlled systems
* ❌ Risky if you're consuming messages from untrusted sources or multi-tenant environments

If you're building something more secure or external-facing, it's better to be explicit, like:

spring.kafka.consumer.properties.spring.json.trusted.packages=com.example.model

This limits deserialization to classes in the com.example.model package only.

application.properties

|  |
| --- |
| spring.kafka.consumer.value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer  spring.kafka.consumer.properties.spring.json.trusted.packages=\*  spring.kafka.producer.value-serializer: org.springframework.kafka.support.serializer.JsonSerializer |

**Example of JSON Serializer and JSON Desteriliser**

**Create payload**

|  |
| --- |
| public class User {  private Integer id;  private String firstName;  private String lastName;  //getter setter  } |

**Create JsonProducer**

|  |
| --- |
| @Service public class JsonProducer {  private static final Logger *LOGGER* = LoggerFactory.*getLogger*(JsonProducer.class);   KafkaTemplate<String, User> kafkaTemplate;   public void sendMsg(User data) {  *LOGGER*.info(String.*format*("Message Send -> %s ", data.toString()));  Message<User> message = MessageBuilder.*withPayload*(data).setHeader(KafkaHeaders.*TOPIC*, "rlbIn").build();  kafkaTemplate.send(message);  } } |

**Create RestAPI to send Message**

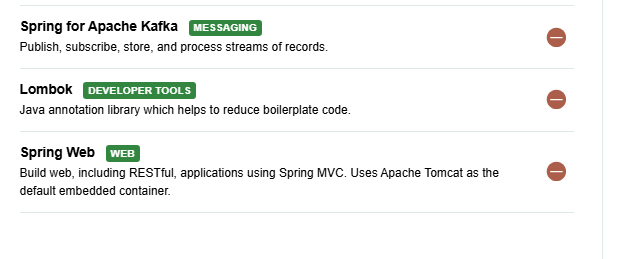
|  |
| --- |
| @RestController @RequestMapping("/kafka/v1") public class JsonController {  private static final Logger *LOGGER* = LoggerFactory.*getLogger*(JsonController.class);  private JsonProducer jsonProducer;   public JsonController(JsonProducer jsonProducer) {  this.jsonProducer = jsonProducer;  }   @PostMapping("/publish/json")  public ResponseEntity<String> publish(@RequestBody User user) {  *LOGGER*.info(String.*format*("Message Sent- > %s", user.toString()));  jsonProducer.sendMsg(user);  return new ResponseEntity<>("JSON Message sent", HttpStatus.*CREATED*);  } } |

**JSON Consumer**

|  |
| --- |
| @Service public class JsonConsumer {  private static final Logger LOGGER = LoggerFactory.getLogger(JsonConsumer.class);   @KafkaListener(topics = "rlbIn", groupId = "myGroup")  public void jsonConsume(User data) {  LOGGER.info("JSON ");  LOGGER.info(String.format("JSON Message received : %s", data.toString()));  } } |

**Multimodal Project**

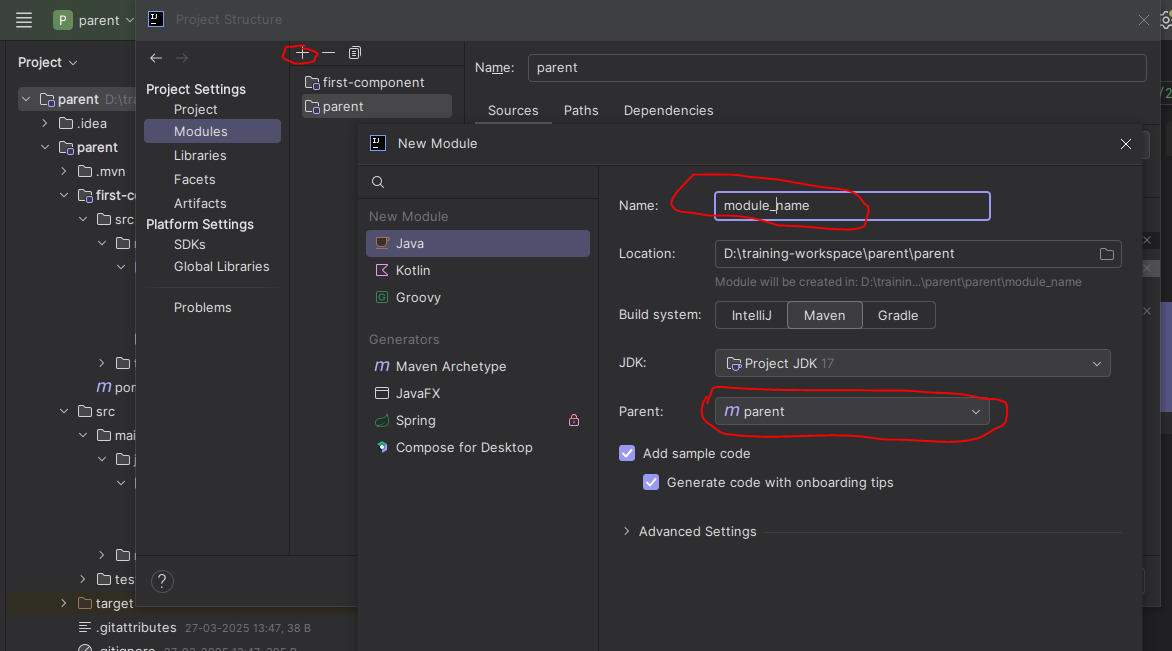
<https://start.spring.io/>



To make this project as Parent in pom.xml the packaging is set to **pom**

|  |
| --- |
| <name>parent</name> <description>Demo project for Spring Boot</description> <packaging>pom</packaging> |

Create child project->Right click on parent project -> Open Module Setting



**Load maven then in parent pom file the child module will be added**

|  |
| --- |
| <name>parent</name> <description>Demo project for Spring Boot</description> <modules>  <module>first-component</module> </modules> <packaging>pom</packaging> |

**In child pom file add packaging to jar.**

|  |
| --- |
| <parent>  <groupId>com.avish</groupId>  <artifactId>parent</artifactId>  <version>0.0.1-SNAPSHOT</version> </parent> <packaging>jar</packaging> <artifactId>first-component</artifactId> <properties>  <maven.compiler.source>17</maven.compiler.source>  <maven.compiler.target>17</maven.compiler.target>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> </properties> |

**Create Starter class into Child Project**

|  |
| --- |
| @SpringBootApplication public class FirstComponentApplication {  public static void main(String[] args) {  SpringApplication.run(FirstComponentApplication.class);  } } |

**Create application.properties under child project**

|  |
| --- |
| spring.kafka.producer.bootstrap-servers: localhost:9092 spring.kafka.producer.key-serializer: org.apache.kafka.common.serialization.StringSerializer spring.kafka.producer.value-serializer: org.apache.kafka.common.serialization.StringSerializer |

**Create kafka Topic**

|  |
| --- |
| @Configuration public class KafkaConfig {  @Bean  public NewTopic streamTopic() {  return TopicBuilder.name("stream").build();  } } |

**Reuse Parent dependency into child module.**

If you want to **inherit dependencies from a parent project into child modules**, Maven makes this super clean with **inheritance via the parent tag**

**Project Structure Example**

my-multi-module-project/

├── pom.xml ← parent project

├── common-lib/

│ └── pom.xml ← child module

├── service-api/

│ └── pom.xml ← child module

Parent pom.xml This parent acts as the **dependency manager** for all your child modules

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>my-springboot-parent</artifactId>  <version>1.0.0</version>  <packaging>pom</packaging>  <!-- This allows Spring Boot dependency management -->  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>3.1.0</version>  <relativePath/>  </parent>  <modules>  <module>common-lib</module>  <module>user-service</module>  </modules>  <dependencyManagement>  <dependencies>  <!-- Define versions centrally -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <version>3.1.0</version>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <!-- Common dependency for all child modules -->  <dependency>  <groupId>com.google.guava</groupId>  <artifactId>guava</artifactId>  <version>32.0.0-jre</version>  </dependency>  </dependencies>  </project> |

Each child module includes the parent in its pom.xml like this:

The <dependencyManagement> section **only defines versions** — it does **not automatically add** the dependency to child projects.

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>com.example</groupId>  <artifactId>my-springboot-parent</artifactId>  <version>1.0.0</version>  <relativePath>../pom.xml</relativePath>  </parent>  <artifactId>user-service</artifactId>  <dependencies>  <!-- Dependency defined in dependencyManagement -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- Dependency inherited directly from parent -->  <dependency>  <groupId>com.google.guava</groupId>  <artifactId>guava</artifactId>  </dependency>  </dependencies>  </project> |

| **Inheritance Type** | **Location** | **Purpose** |
| --- | --- | --- |
| parent tag | Child POM | Links child to parent |
| <dependencies> | Parent POM | Included automatically in child |
| <dependencyManagement> | Parent POM | Versions are inherited, but you still need to declare the dependency in child |

**use a sibling module (rlb-core) inside another module(rlb-oc)**

Update this part in rlb-oc/pom.xml

|  |
| --- |
| <dependency>  <groupId>com.rlb</groupId>  <artifactId>rlb-core</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency> |

**Publish Event with Key**

|  |
| --- |
| public void publishJSOSNpPayload(User payload) {  LOGGER.info("Event sent ->{}", payload.toString());  ~~//Message<User> message = MessageBuilder.withPayload(payload).setHeader(KafkaHeaders.TOPIC, "json\_topic").build();  // kafkaTemplate.send("json\_topic",payload.getId().toString(),payload);~~  ProducerRecord<String, User> record = new ProducerRecord<>(  "json\_topic",  payload.getId().toString(),  payload  );  kafkaTemplate.send(record); } |

**Consumer with concurrency**

|  |
| --- |
| @Service public class JSONConsumer {  private static Logger LOGGER = LoggerFactory.getLogger(JSONConsumer.class);   @KafkaListener(topics = "json\_topic", groupId = "rlbGroup", concurrency = "2")  public void JsonPayloadConsume(User payload) {  String threadName = Thread.currentThread().getName();  LOGGER.info("JSON payload received ->{} by Thread {}", payload,threadName);  } } |

**Generate different id to assign different partition**

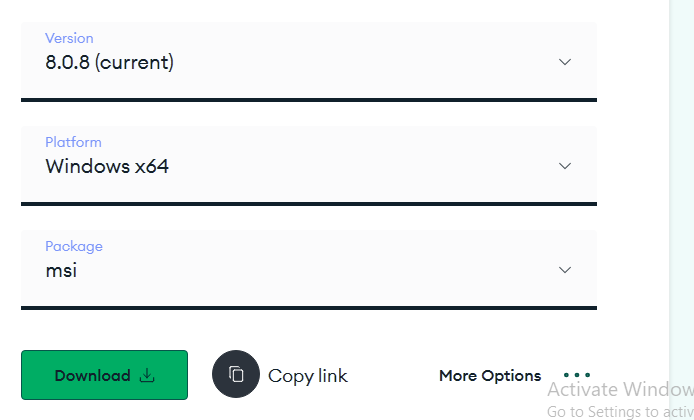
|  |
| --- |
| Random rand = new Random(); for (int i=0;i<100;i++){  user.setId(rand.nextInt(100));  jsonProducer.publishKeyEvent(user); } |

**Mongo DB installation**

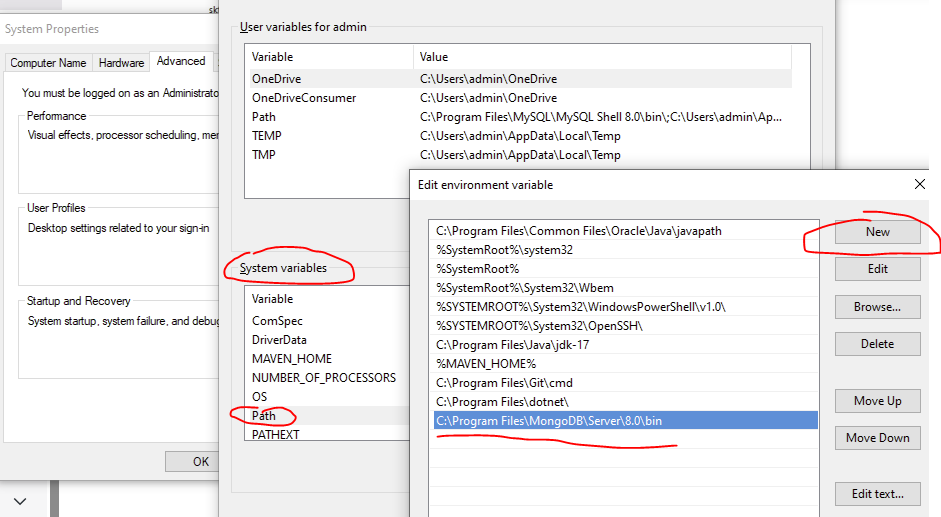
[MongoDB](https://www.geeksforgeeks.org/mongodb-an-introduction/), the most popular NoSQL database, is an open-source document-oriented database. The term ‘NoSQL’ means ‘non-relational’. It means that MongoDB isn’t based on the table-like relational database structure but provides an altogether different mechanism for the storage and retrieval of data. This format of storage is called [BSON](https://www.geeksforgeeks.org/difference-between-json-and-bson/)( similar to JSON format). A simple MongoDB document Structure:

|  |
| --- |
| {  title: rlb,  by: 'Avinash Khadsan,  url: 'https://www.rlb.com',  type: 'NoSQL'  } |

**Download MongoDB**: <https://www.mongodb.com/try/download/community>



 Set Environment Variables: C:\Program Files\MongoDB\Server\8.0\bin

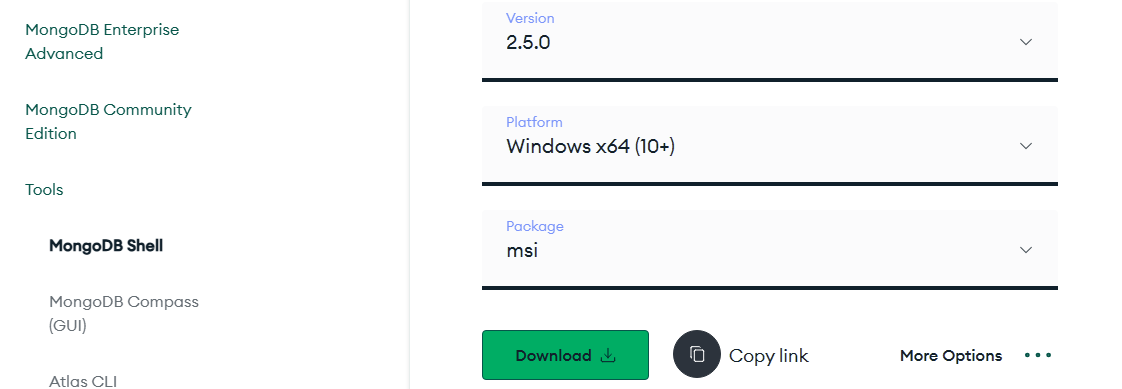


Create Folder:  C:/data/db/

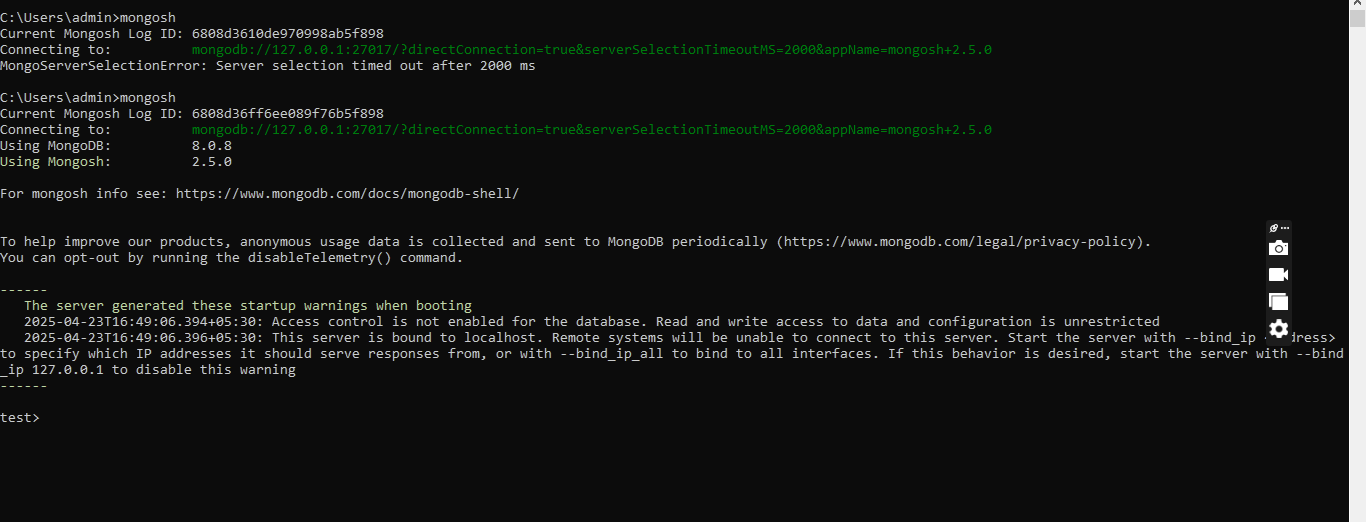
**Start mongo DB: C:\Users\admin>mongod**

The recommended shell for interacting with MongoDB databases is now mongosh, which provides improved functionality, better syntax, and full compatibility with the latest MongoDB features.

Download mongosh: <https://www.mongodb.com/try/download/database-tools>



**Start mongosh: C:\Users\admin>mongosh**



**Create a Database**

Now we can make a new **database**, **collections**, and **documents**in our shell. Use the following command within the mongosh shell to create a new database:

use database\_name

The use **Database\_name**command makes a new [database](https://www.geeksforgeeks.org/what-is-database/) in the system if it does not exist, if the database exists it uses that database:

use rlb

**Add Data to a Collection**

Insert a document into a collection using:

db.collection\_name.insertOne({field: value})

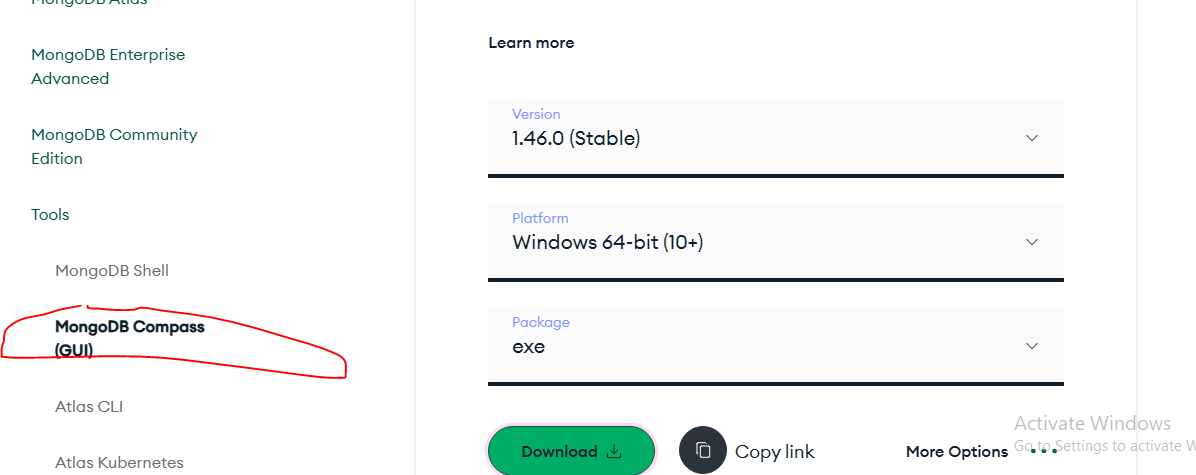
The**db.Collection\_name** command makes a new collection in the rlb database and the [insertOne()](https://www.geeksforgeeks.org/mongodb-insertone-method-db-collection-insertone/) method inserts the document in the **user** collection:

db.user.insertOne({name:'avinash'})

**Retrieve document from collection**

db.user.find()

**Download compass**: <https://www.mongodb.com/try/download/compass>



[**MongoDB Compass**](https://www.geeksforgeeks.org/install-mongodb-compass-on-windows/) is a powerful GUI for querying, aggregating, and analyzing your MongoDB data in a visual environment. Compass is free to use and source available and can be run on macOS, Windows, and Linux

Use default PORT: **27017** to connect mongodb from compass.

Reference : <https://www.geeksforgeeks.org/how-to-install-mongodb-on-windows/>

**Spring Boot + Mongo DB**

[**application.properties**](https://www.geeksforgeeks.org/spring-boot-application-properties/?ref=rp)

|  |
| --- |
| # Configuration for MongoDB  spring.data.mongodb.host=localhost  spring.data.mongodb.port=27017  spring.data.mongodb.database=rlb |

MongoRepository is an interface provided by Spring Data in the package  org.springframework.data.mongodb.repository. MongoRepository extends the PagingAndSortingRepository and QueryByExampleExecutor interfaces that further extend the CrudRepository interface. MongoRepository provides all the necessary methods which help to create a CRUD application and it also supports the custom derived query methods.

public interface MongoRepository<T,ID>

extends PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>

|  |
| --- |
| <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-data-mongodb</artifactId>  </dependency> |

|  |
| --- |
| @Document(collection = "user") public class User {  @Id  private Integer id;  private String userName;  private String surname;   public User(Integer id, String userName, String surname) {  this.id = id;  this.userName = userName;  this.surname = surname;  } } |

|  |
| --- |
| @Repository public interface UserRepository extends MongoRepository<User, Integer> { } |

**Primary key in MongoDB collection.**

MongoDB can use Integer (or Long) as the ID in theory, but it's not recommended by default.

* MongoDB natively expects the \_id to be unique and usually uses a special type called ObjectId.
* If you use an Integer for the @Id, you must manually handle ID generation yourself — MongoDB won't auto-generate integers for you.
* If you don't manage it carefully, you can easily get duplicate key errors.
* To generates IDs automatically for your entities use @Id and the field type is String or ObjectId.
* If you don't set the id when saving a new document, Spring Data MongoDB will generate a unique ID (using MongoDB's ObjectId). If you manually set an id, it will use your provided value instead.
* If the type is String, Spring will generate an ObjectId and convert it to a string automatically.
* If the type is org.bson.types.ObjectId, it just keeps it as an ObjectId.

**Use UUID instead of ObjectId**

Sometimes people prefer **UUIDs** over MongoDB's ObjectId (for readability, integration with other systems, etc).

|  |
| --- |
| @Data  @Document(collection = "user")  public class User {  @Id  private String id;  private String name;  private int age;  public User() {  this.id = UUID.randomUUID().toString();  }  } |

### Why ****use UUID**** instead of MongoDB's ****ObjectId****?

**Consistency across systems**:

If your system uses UUIDs elsewhere (Postgres, Kafka keys, Redis, etc.), it’s easier to keep everything using UUIDs instead of mixing ObjectId and UUID

**Client-side generation:**

UUIDs can be generated **before** saving to MongoDB. (This can be useful if you need an ID immediately without touching the database.)

**Readability:**

UUIDs (e.g., f47ac10b-58cc-4372-a567-0e02b2c3d479) are often seen as more "standard" and recognizable in APIs than ObjectIds (e.g., 650fdc44eb2a1d23f8b9aa2e).

**Globally Unique:**

UUIDs (e.g., f47ac10b-58cc-4372-a567-0e02b2c3d479) are often seen as more "standard" and recognizable in APIs than ObjectIds (e.g., 650fdc44eb2a1d23f8b9aa2e).

### But why stick with ****ObjectId**** usually?

**Smaller size:**

ObjectId = 12 bytes; UUID = 16 bytes. (Saves a little space.)

**Built-in features:**

ObjectId includes a timestamp inside it (first 4 bytes), which can be useful for sorting by creation time without an extra createdAt field.

**Optimized for Mongo:**

MongoDB indexes and handles ObjectIds very efficiently.

**Default behaviour:**

Less code to maintain — Mongo handles ID generation automatically

### Real-world guideline:

* If you're **only** using MongoDB → **stick with ObjectId**.
* If you need **cross-system compatibility** or **client-side ID generation** → consider **UUID**.

**Example scenario**:  
Imagine you have a microservices system where:

* Kafka events use UUIDs
* PostgreSQL uses UUIDs
* Redis keys are UUID-based  
  Then you'd prefer to use UUIDs **everywhere**, including Mongo, to avoid confusion.

UUIDs are random whereas ObjectID values always increase over time

An attacker cannot deduce the ID of another object if they have the ID of one object in case of UUID.

**Relationships**

In MongoDB, relationships are different than in SQL. You have two main styles:

* **Embedding**: You copy data inside documents (denormalization).
* **Referencing**: You store just IDs (like foreign keys).

**1: Referencing documents using @DBRef**

Suppose you have a User and a Role document.

|  |
| --- |
| import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.DBRef;  import org.springframework.data.mongodb.core.mapping.Document;  import java.util.List;  @Document(collection = "users")  public class User {  @Id  private String id;  private String username;    @DBRef  private List<Role> roles; // Reference to Role documents  // getters and setters  } |

|  |
| --- |
| import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  @Document(collection = "roles")  public class Role {  @Id  private String id;  private String name;  // getters and setters  } |

**Important about @DBRef:**

* It saves only the id of the referenced document.
* By default, Spring Data loads the references *lazily* unless you configure otherwise.

**2: Embedding documents inside another document**

If you want to **embed** full documents (not just references):

|  |
| --- |
| @Document(collection = "users")  public class User {  @Id  private String id;  private String username;  private List<Role> roles; // Embedded documents  // getters and setters  } |

Now Role will be stored *inside* the User document.

**Which one should you use?**

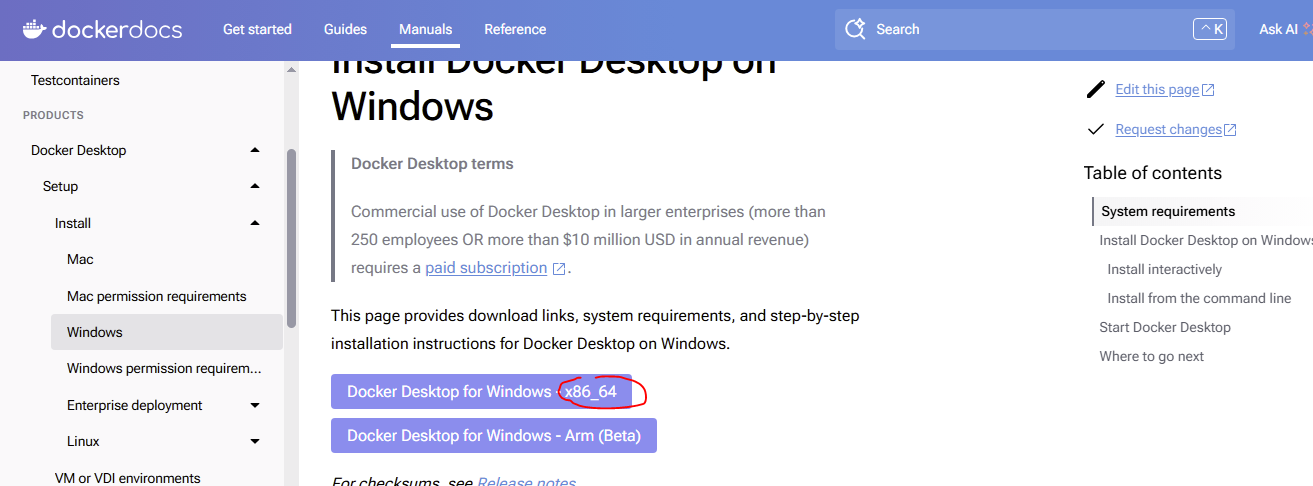
| **Embedding** | **Referencing (@DBRef)** |
| --- | --- |
| Small, static data | Big, reusable, changing data |
| Fast reads | More normalized, less duplication |
| Hard to update just part of embedded | Easier to update separate documents |

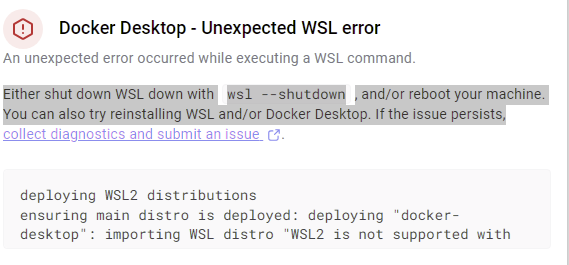
**Heads up**:  
MongoDB relationships don't enforce integrity like SQL (FOREIGN KEY), so if you delete a Role, the User still might reference it unless you handle that manually.

**What is Docker?**

[Docker](https://www.geeksforgeeks.org/introduction-to-docker)is an open-source container platform software tool, where you run your applications in the form of containers. [Docker container](https://www.geeksforgeeks.org/virtualisation-with-docker-containers)comes with light weighted softwares having all the dependencies and configurations so we can run them across different computing environments. It facilitates the developers to package their application with all its dependencies into a single entity in the form of images. These can be portable easily or sharable with other developers without worrying about the underlying OS

<https://docs.docker.com/desktop/setup/install/windows-install/>





To Enable virtualization press Windows +R and search optionalfeatures.exe

* Mark- Hyper-V, Windows subsystem for Linux and Virtualization Machine platform

And restart system and start installing Docker.

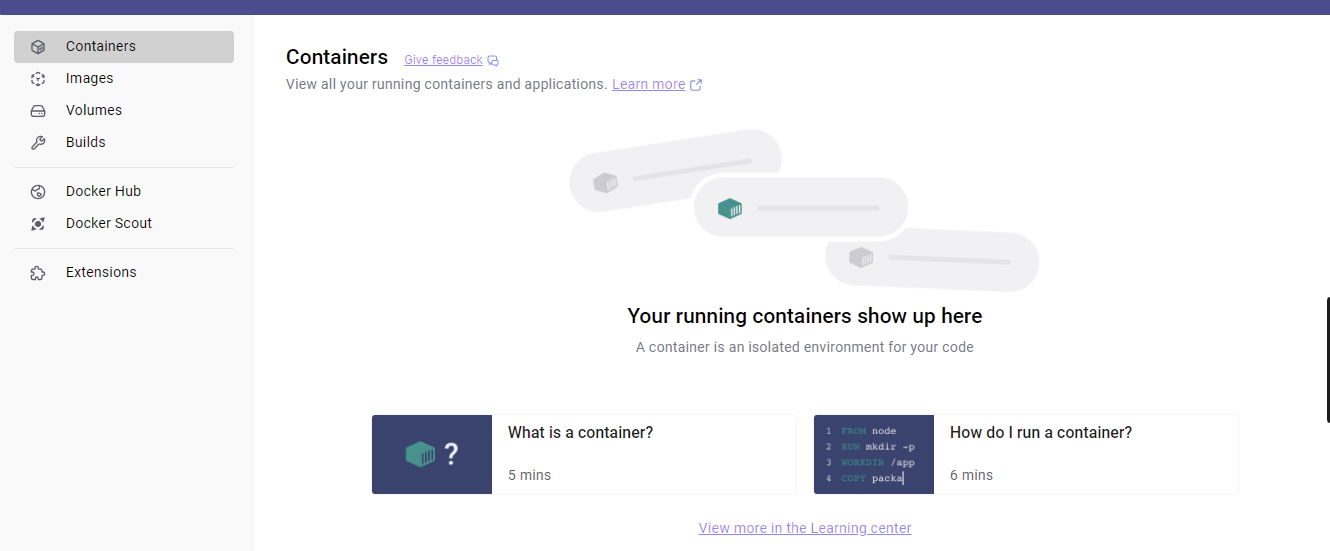


Image is actual configuration of application. And once you run image in local it is running under environment it is container. Image is part of container runtime.

What is container?

* A way to package applications with all the necessary dependencies and configuration.
* Portable artefact easily shared and moves around in between development team and operation team.
* Make development and deployment more efficient.
* Container lives in container repository public or private repository.

Layer of the Linux base image, application image with configuration



Before container

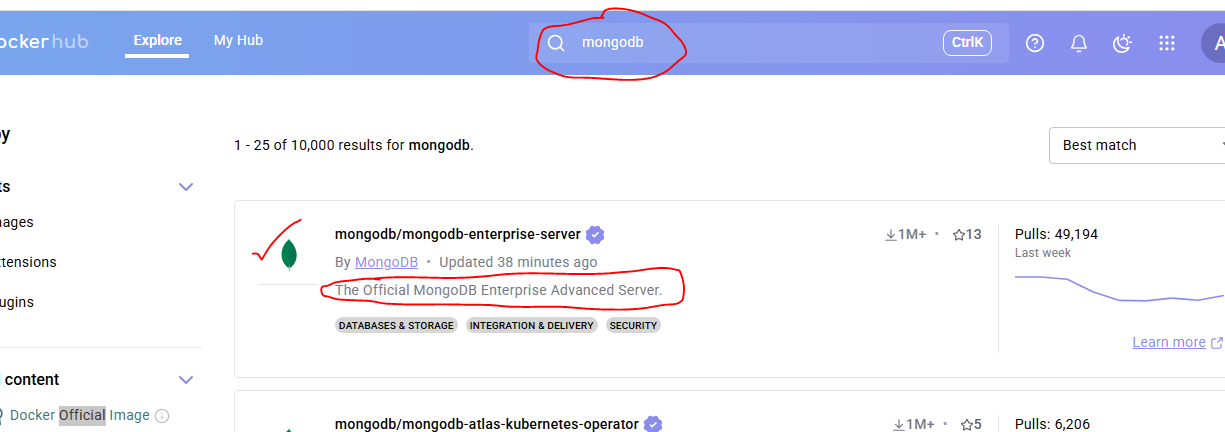
* Each developer needs to install the application specific version.
* Installation process is different on each OS environment.
* Many steps where something cloud get wrong

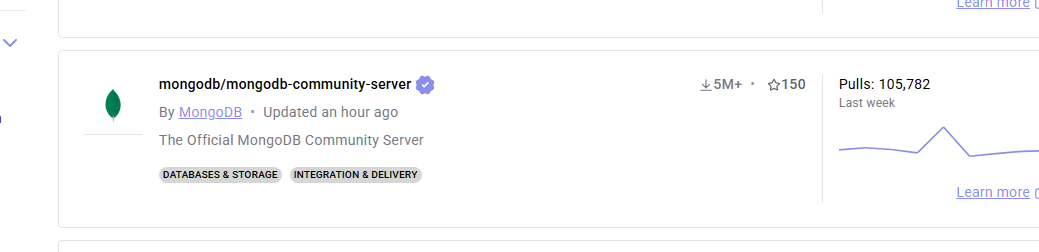
After container

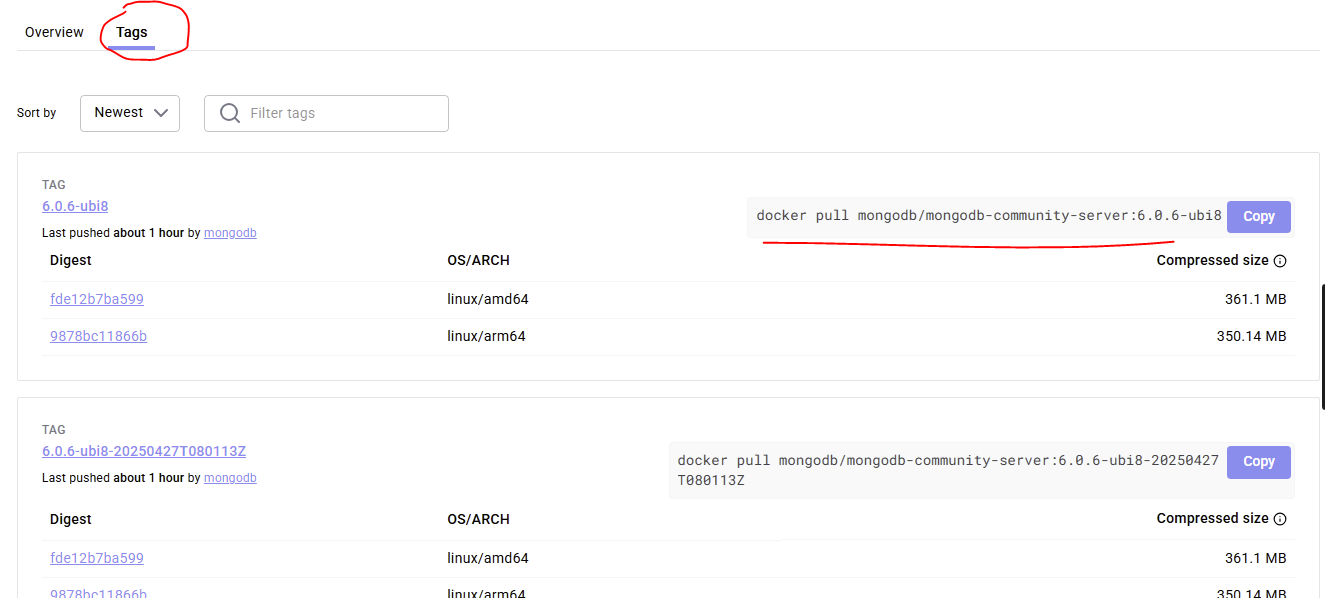
* Package with all needed configuration.
* One command to install to application.
* Run the same with 2 different versions.

Container is having its own isolated OS and Container is a running environment of an image.

**How to run image in Docker**







Pull the image

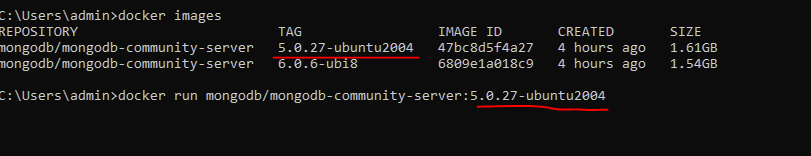
|  |
| --- |
| C:\Users\admin>docker pull mongodb/mongodb-community-server:6.0.6-ubi8 |

Run the image

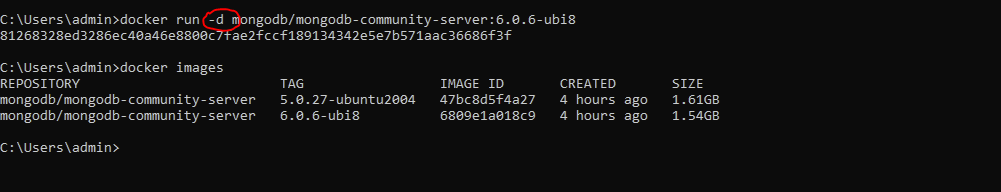
docker run mongodb/mongodb-community-server:6.0.6-ubi8



**List images**

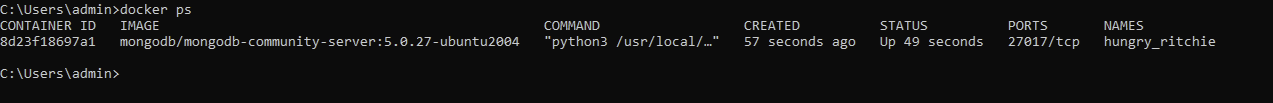
C:\Users\admin>docker images

**Start container**: docker run image:tag / docker run –d image:tag

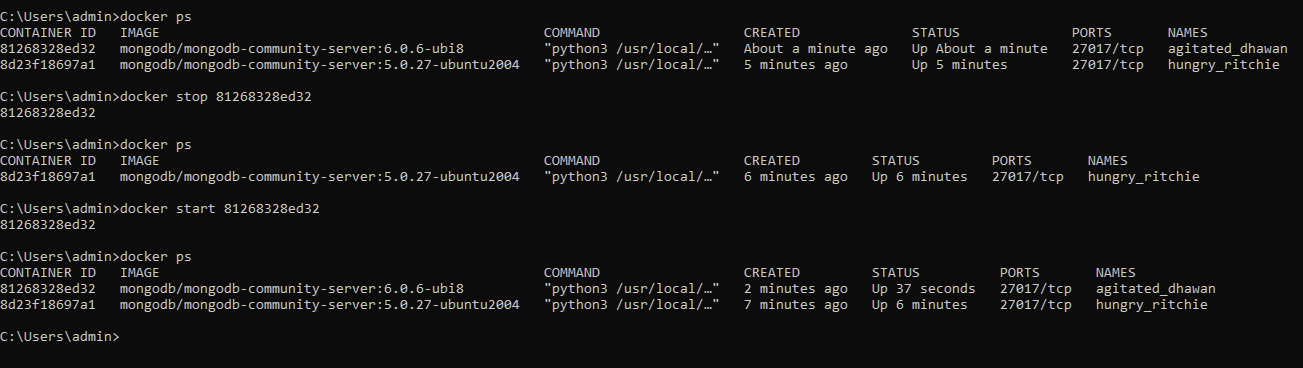


Docker run image:tag : if image not found then it will download and start.

**List state of container**

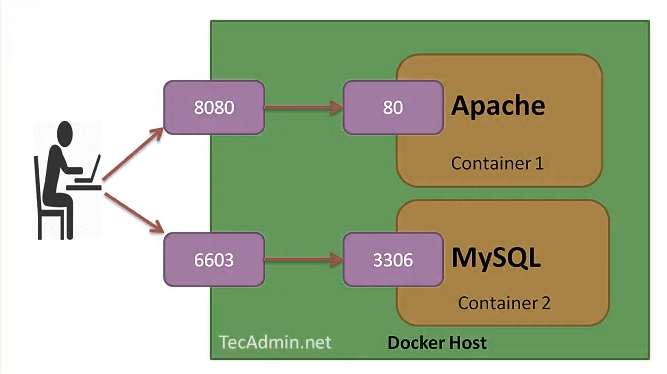
docker ps/ docker ps -a 

**Restart docker container:** docker stop container\_id/ docker start container\_id

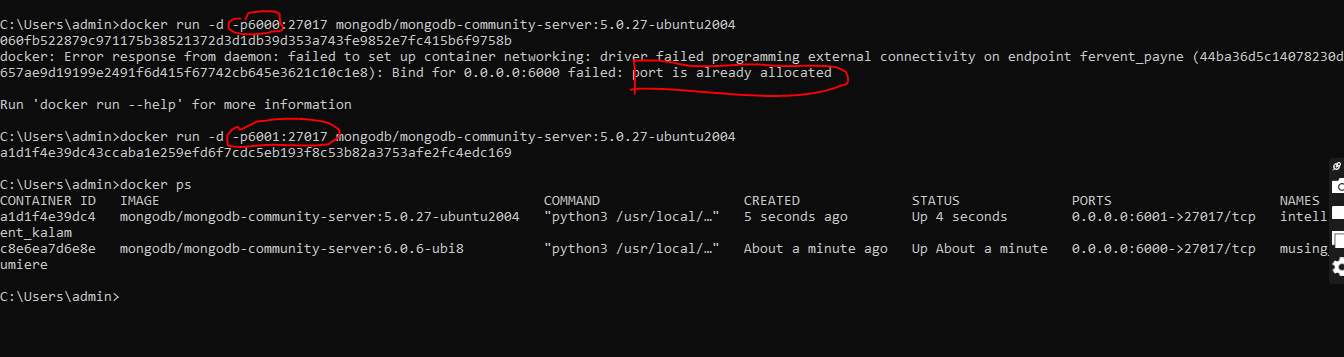


**Port binding**

Container is running is on your host so if you want to access container you need to do host port and container port binding.

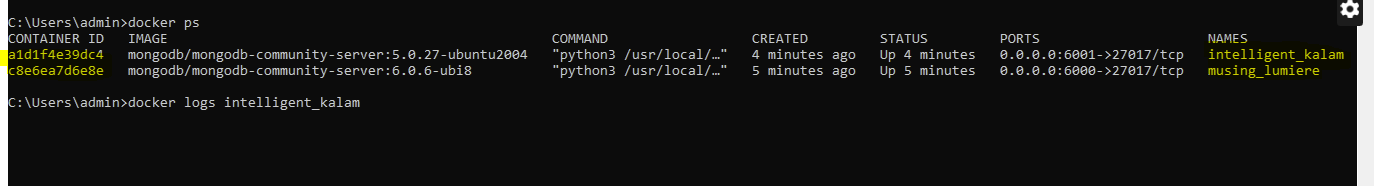






**Troubleshooting on command**

C:\Users\admin>docker logs a1d1f4e39dc4



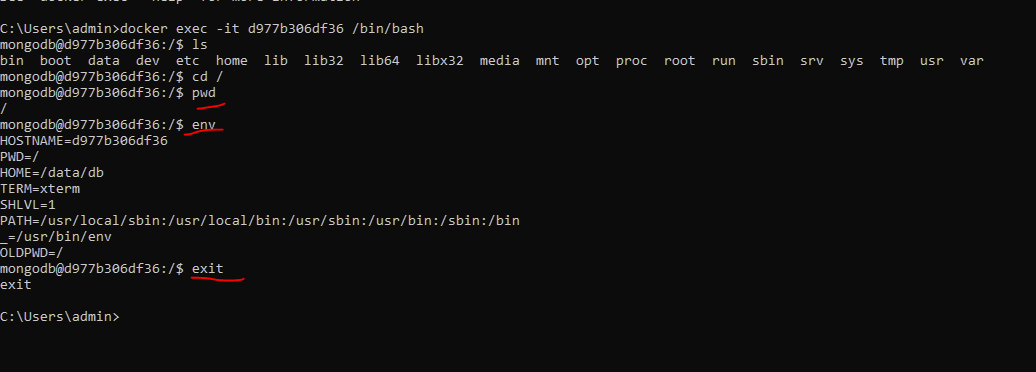
Get terminal over container for troubleshooting check log file , print env variable

C:\Users\admin>docker exec -it d977b306df36 /bin/bash



**Name your container**

docker run –d –p4000:27017 mongodb/mongodb-community-server:5.0.27 -ubuntu2004



Eureka-server Dockerfile

|  |
| --- |
| FROM openjdk:17-alpine COPY target/\*.jar eureka-server.jar ENTRYPOINT ["java", "-jar", "eureka-server.jar"] |

**Build docker image**

mvn clean install

docker build -t eureka-server:0.0.1 .

docker images

docker run -d --name eureka-server -p 8761:8761 eureka-server:0.0.1