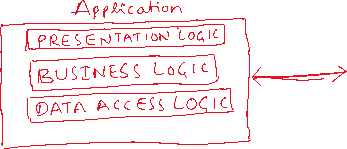
Microservices



Monolithic application:

* The term “monolithic” represents large, indivisible unit.
* A monolithic application is a single codebase application, where all the components of the application are contained within a single codebase.
* The all functionalities of the application are contained with in a single jar/war file, like Presentation logic(UI), business logic, and data access logic.



* For example, Netflix, Amazon, Facebook etc.. they are all started as monolithic applications.
* If you consider early Amazon application, it is a monolithic application, where it includes features like a single user interface, it has product catalog, it has Shopping cart component, it has Payment processing component, it has Order Management component, etc.. are part of a single application.

Chanllenges with monolithic applications:

1. If a specific component is getting heavy traffic, the entire application has to be scaled on to the multiple servers. So, it will waste the resources.

For example, In E-commerce application, when a sale is announced, heavy traffic comes to the product catalog, but to handle the traffic, entire application has to be scaled on to the multiple servers.

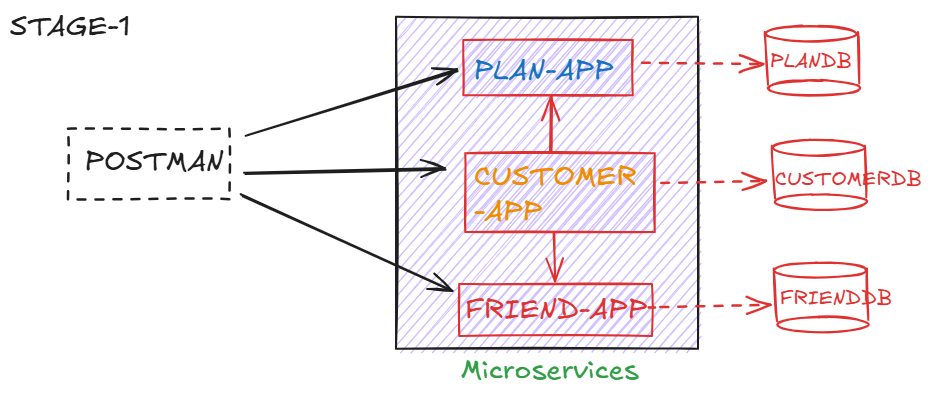
1. If a minor issue occurred in one of the component, then entire application has to be redeployed again. It will increase the downtime of the application.
2. All the teams has to use the same technology stack to implement all the components.
3. Code complexity.

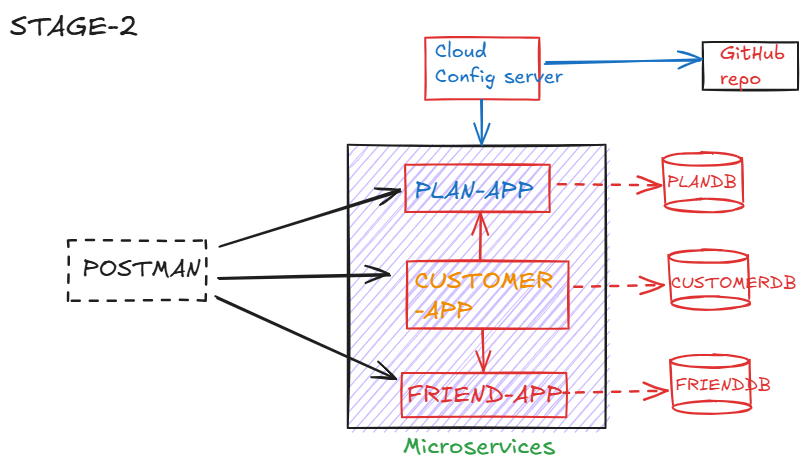
* To overcome the challenges faced in monolithic applicatins, we got a new architectural style called “MICROSERVICES”.
* The term architectural style refers to some patterns/principles and constraints that define the structure of the system.
* It defines, how components should interact and how the data should flow in a system.
* Microservices is a distributed system, where a large application can be decomposed into smaller apps(micro apps) with independent responsibility principle.
* The characterstics of Microservices are,

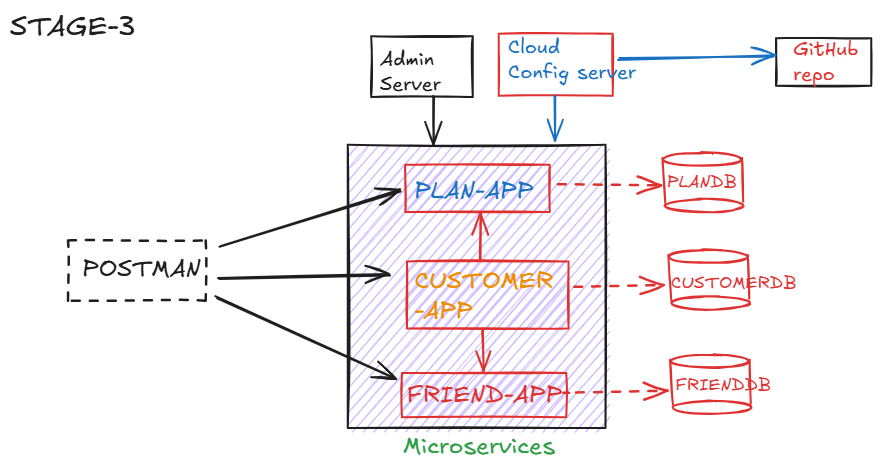
1. individual development: development of one microservice does not depend on the completion of another.
2. individual deployment: each microservice can be deployed in a separate docker image.
3. individual scaling: if one microservice is getting more traffic then that one service can be scaled onto the multiple servers.
4. individual monitoring: the performance and errors of each microservice can be monitored separately.
5. individual tech stack: each microservice can be developed in a different tech stack.

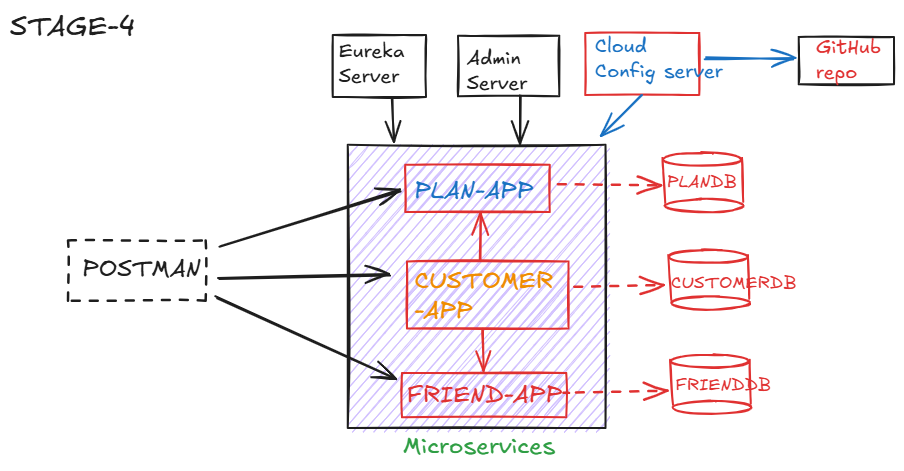
* Microservices are distributed by nature. It means, the different components executes at different places and connected over a network, and provides an end result to the user of the application.
* For example, if you want to design a social media platform with microservices, then the individual microservice apps could be,

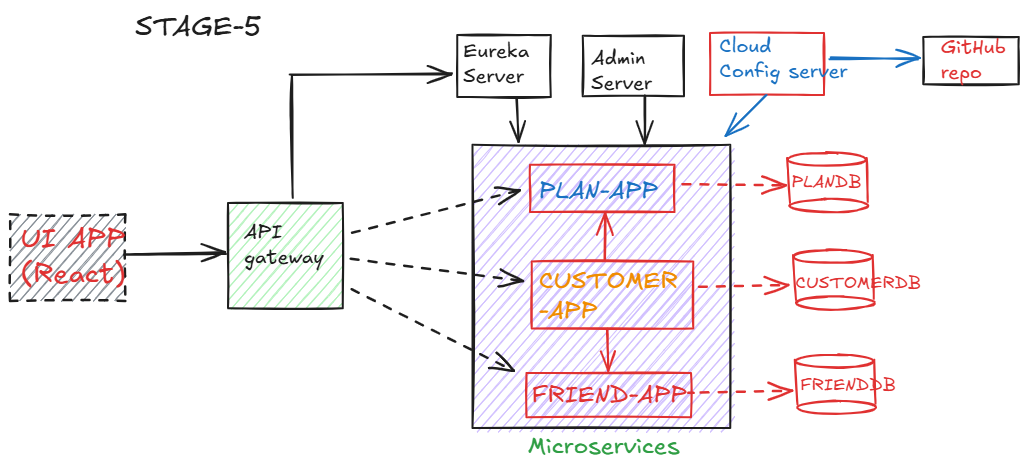
1. User Profile management
2. News Feed
3. Friend Requests
4. Notifications
5. photo/video sharing, etc..











PLAN-APP:

* This service exposes API endpoints to fetch plan details.

GET /plans : returns all the existing plans

GET /plans/{id} : return the details of a specific plan.

* We create a table PLANDETAILS, with columns,

PLANID

PLANNAME

DESCRIPTION

VALIDITY.

* insert some sample records into the table.

<https://github.com/ShekherJava/PLAN-APP.git>

FRIEND-APP:

* This service exposes API endpoints to add a friend contact and to fetch the friends contacts of a customer.

POST /friend/save : adds a friend contact

GET /friend/{phoneNumber} : returns friends contacts

* create a table in MySQL, like below

use FRIENDDB;

CREATE TABLE FRIEND (

ID INT PRIMARY KEY auto\_increment,

PHONE\_NUMBER LONG,

FRIEND\_NUMBER LONG

);

* insert some sample records like below and commit.

INSERT INTO FRIEND(PHONE\_NUMBER, FRIEND\_NUMBER) VALUES(9009009001, 9191919191);

INSERT INTO FRIEND(PHONE\_NUMBER, FRIEND\_NUMBER) VALUES(9009009001, 9292929292);

INSERT INTO FRIEND(PHONE\_NUMBER, FRIEND\_NUMBER) VALUES(8008008001, 8181818181);

COMMIT;

<https://github.com/ShekherJava/FRIEND-APP.git>

CUSTOMER-APP

* This service exposes API endpoints to register a customer, login a customer and fetch the profile of a customer.

POST /customer/register: to register a customer

POST /customer/login : to login a customer

GET /customer/profile/{phoneNumber}: to fetch the customer profile.

* create a table in MySQL like below,

use CUSTOMERDB;

CREATE TABLE CUSTOMER (

PHONE\_NUMBER LONG,

USERNAME VARCHAR(100),

PASSWORD VARCHAR(150),

EMAIL VARCHAR(50),

PLAN\_ID VARCHAR(100)

);

* insert some sample records like below and commit.

INSERT INTO CUSTOMER VALUES(9009009001, 'JOHN', 'DOE', 'john@gmail.com', 'PLAN-399');

COMMIT;

<https://github.com/ShekherJava/CUSTOMER-APP.git>

Cloud Config Server:

* We are going to place the configuration properties of our microservices in the respective application.properties or application.yml files.
* The problem with this approach is, if any configuration requires modifications then we need to change in multiple services and redeploy them.
* If a service has multiple instances, then all these instances must be redeployed.
* The solution is, maintain the common confiugration properties of the services at a centralized place like Github.
* Our microservices will get the configuration properties from the cloud config server and this config server will load the configuration properties from the github.

1. create a repository in GitHub.
2. create a file application.properties in the repository.
3. place the common properties in the application.properties file.
4. create a spring boot project Cloud\_Config\_Server with the required dependencies
5. add the git repository url in the properties file of the config server.
6. add @EnableConfigServer annotation on spring boot main class

Note: To verify whether config server has loaded the configuration properties from the github repository or not, send a request to the below url.

<http://localhost:8094/application-default.properties>

* In the microservices, do the below chanages.

1. remove the common properties from the respective application.properties files.
2. add a new starter config client to the microservices.

Note: right click on microservice 🡪 spring 🡪 add starters🡪 choose config client 🡪 next🡪 finish.

1. add the below property in the respective application.properties file of microservices.

spring.config.import=optional:configserver:http://localhost:8094

How to reload the configuration properties dynamically?

* A microservice can import the configuration properties from the config server, during application startup.
* After service is started, if any modifications are done to the configuration properties in the Github repository, then the changes are not automatically refreshed into the microservice.
* Again, you need to restart the service, to get the latest configuration properties.
* To avoid restart of the service, we need to make the below changes to the microservice.

1. add spring boot actuator dependency
2. add @RefreshScope annoation at boot main class or rest controller class.
3. enable refresh endpoint of actuator in the application.properties file of the microservice.

management.endpoints.web.exposure.include=refresh

1. send HTTP POST request to the /actuator/refresh endpoint.
2. After refresh, test your microservice, to check whether configuration properties are refreshed or not.

<https://github.com/ShekherJava/Cloud_Config_Server.git>

Admin server:

* Spring Boot Admin Server is a monitoring tool which is designed for managing the spring boot applications.
* It provides a centralized dashboard, where the developers can monitor the health, status and other metrics of the spring boot applications in a single user interface.
* It will show the application metrics in a visualized format.
* add @EnableAdminServer on boot start class, to make a spring boot application as admin server.
* while creating admin server application, in pom.xml file, the below dependency is required.
* <dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-server</artifactId>

* </dependency>
* Add amdin client and spring boot actuator dependencies to each microservice application.
* add the below two properties in application.properties file of each microservice.
* spring.boot.admin.client.url=http://localhost:8095
* management.endpoints.web.exposure.include=\*
* Access the admin server dashboard at http://localhost:8095

<https://github.com/ShekherJava/Admin_Server.git>

Eureka server (Service Registry):

* In microservices, there is a inter-service communication between one service to another.
* one service may talk to multiple services also.
* In this communication, suppose if host or port of a service is changed, then the other service can’t able communicate to it.
* So, as a solution, we got service discovery pattern in microservices.
* In this pattern, each microservice registers with a central server called service registry.
* During registration, the service name(application name), its host and port are registered with the service registry.
* If a service is registered then only it can able to find the other services registered in the registry.
* If one service get redeployed at a different host or port then other services need not worry about it.
* When the service gets redeployed, it would simply updates its information in the service registry.
* So, the other services would discover its updated details through service registry.

steps to create eureka server:

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

1. add eureka server and dev tools dependencies while creating the project.
2. add @EnableEurekaServer at boot start class.
3. add the below properties to application.properties file.

spring.application.name=Service\_Registry

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

* If you don’t specify server.port=8761, by default eureka server runs on port 8761.
* register-with-eureka=false, prevents the server from registering itself as a client.
* fetch-registry=false, stops the server from trying to fetch the registry, which is unnecessary for the server role.
* service-url.defaultZone defines the URL where Eureka clients can find the Eureka server to register themseleves.

<https://github.com/ShekherJava/Service_Registry.git>

* add eureka client dependency in microservices.
* add @EnableDiscoveryClient to the boot start class of microservices.

<https://github.com/ShekherJava/CUSTOMER-APP.git>

<https://github.com/ShekherJava/FRIEND-APP.git>

<https://github.com/ShekherJava/PLAN-APP.git>

Feign client:

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

* To facilitate the communication between the microservices, we have 3 options.

1. RestTemplate
2. WebClient
3. Feign client

How to start multiple instances of a microservice?

* For example, if you want to start multiple instances of PLAN-APP application, do the following changes.

1. in application.properties file, disable server.port
2. Right click on PLAN-APP 🡪 RunAs 🡪 Run Configurations 🡪 choose the application PLAN-APP 🡪 Arguments tab 🡪 in VM arguments, enter -Dserver.port=7091 🡪 apply 🡪 Run (instance1)
3. Right click on PLAN-APP 🡪 RunAs 🡪 Run Configurations 🡪 choose the application PLAN-APP 🡪 Arguments tab 🡪 in VM arguments, enter -Dserver.port=8091 🡪 apply 🡪 Run (instance2)
4. Right click on PLAN-APP 🡪 RunAs 🡪 Run Configurations 🡪 choose the application PLAN-APP 🡪 Arguments tab 🡪 in VM arguments, enter -Dserver.port=9091 🡪 apply 🡪 Run (instance3)

Note: Open Eureka dashboard and find out the number of instances registered with Eureka Registry.

* Open PlanRestController class, and autowire Environment object.

@Autowired

Environment env;

* goto getPlanById() method of PlanRestController class and add the below statement.

System.***out***.println("Port : " + env.getProperty("server.port"));

* After making these changes, start the multiple instances of PLAN-APP
* Now open POSTMAN, and send mulitple requests to the /customer/profile/{phoneNumber} endpoint of CUSTOMER-APP service.
* Now observe the PLAN-APP console’s to see the port message, and observe the feign client is load balancing the request to the multiple instances of PLAN-APP or not.

API Gateway/Edge Server:

* Suppose, a client application is making REST API calls to back-end microservices.
* The direct communication between clients and the microservices network has the below problems.

1. If any microservice’s host/port is changed, then according to that the client application also need to be changed.
2. Suppose, if the microservices are refactored like two services are combined as a single service then also the client application has to be changed.
3. Suppose, if we want to allow only authorized clients to access the back-end services, then we need to implement security at each service. This is duplication of code.

* The solution for the above problems is, create a Gateway server between the client and the microservices network.
* Now all the communication between the client and the back-end services will happen through the Gateway server.
* When a client makes a request to the gateway server, the gateway server will fetch the host and port of the request microservice from Eureka server and then it will forward/route the request to that service.

step-by-step development:

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

1. create a new spring boot project, choose gateway, eureka client and devtools dependencies
2. add @EnableDiscoveryClient on top of spring boot main class.
3. in application.yml file, configure the below properties.
4. spring:
5. application:
6. name: API\_Gateway
7. cloud:
8. gateway:
9. discovery:
10. locator:
11. enabled: true
12. lower-case-service-id: true
13. server:
14. port: 8096
15. eureka:
16. client:
17. service-url:
18. defaultZone: <http://localhost:8761/eureka/>

* Run the gateway server.
* start all the applications like admin server, config server, services,eureka.
* Now start the postman and pass the requests through api-gateway.

ex: <http://localhost:8096/plan-app/plans>

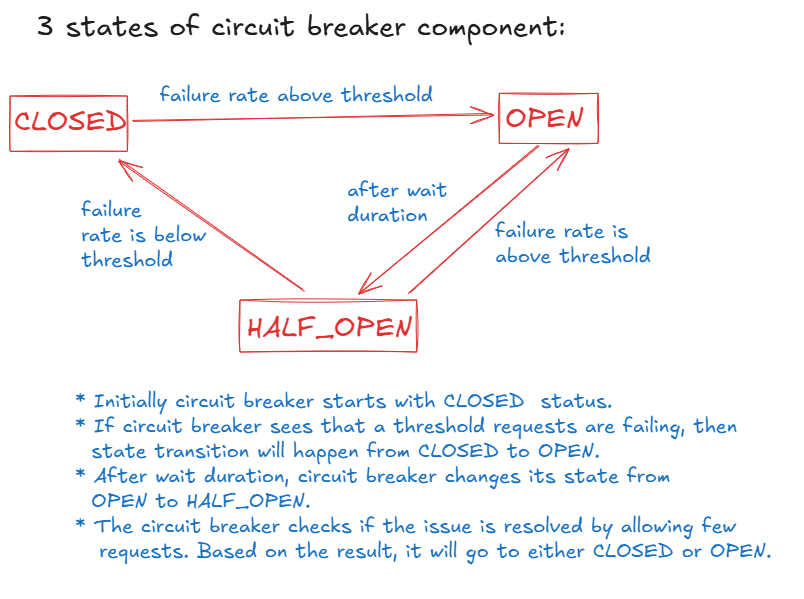
<http://localhost:8096/customer-app/customer/profile/9009009001>

API Gateway github url:

<https://github.com/ShekherJava/API_Gateway.git>

Resilience4j

* Resilience means, adding the capabilities to the application that it can recover quickly from its failures.
* To avoid cascading failures and also to handle the failures gracefully, we have to use resilience4j library.
* Before resilience4j, we had hystrix library and it was developed with the netflix library support.
* In microservices, for a request, a chain of services will communicate.
* When a there is a failure occurred in a single service, then it effects the entire chain.
* So, we need to ensure that the entire chain of microservices should not fail, when there is a failure occurred in one service.
* We need to define a fallback service, so that when actual service is failing, the fallback service should be invoked to get some default values.
* To make a service, quickly recover from its failures, we have to implement circuit breaker pattern.
* circuit breaker will monitor the remote calls and if enough calls fail, then circuit breaker will popup and it prevents the future calls to that service.



* Do the below changes to the CUSTOMER-APP

1. add the dependency in pom.xml

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

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>

</dependency>

1. add the below properties to the application.properties file.

resilience4j.circuitbreaker.configs.default.sliding-window-size=6

resilience4j.circuitbreaker.configs.default.failure-rate-threshold=50

resilience4j.circuitbreaker.configs.default.permitted-number-of-calls-in-half-open-state=2

resilience4j.circuitbreaker.configs.default.wait-duration-in-open-state=10000

spring.cloud.openfeign.circuitbreaker.enabled=true

Here, sliding-window-size denotes how many requests that the circuit breaker has to watch initially.

failure-rate-threshold denotes the percentage of requests to be failed, to move the circuit breaker from one state to another state.

wait-duration-in-open-state denotes a milliseconds of time that circuit breaker has to wait to move from open to half open state.

1. start all the applications like, admin server, service registry, config server, friend and customer services except plan service.
2. open postman, send a request to profile endpoint of customer. You will get 500 internal server error.
3. open a browser, and send a request to circuitbreakerevents endpoint of actuator.

<http://localhost:8093/actuator/circuitbreakerevents>

retry pattern:

* The retry pattern makes a service to retry for other service for a given attempts.
* After retry attempts also, if still the service is unavailable then a service will invoke the fallback method configured.
* The basic idea of retry pattern is a service may be temporarily unavailable and if tried for mulitple times with a wait duration, then it may become available. Because, a service may have temporary network disruption.
* If the service is still unavailable after retries also, then fallback configured will be called.
* The fallback method should have the same return type of the retry method and also should contain the same parameters. Additionally fallback method should contain Throwable parameter.
* In application.properties file, you need to configure the below properties.
* resilience4j.retry.configs.default.max-attempts=3
* resilience4j.retry.configs.default.wait-duration=2s

microservices security:

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

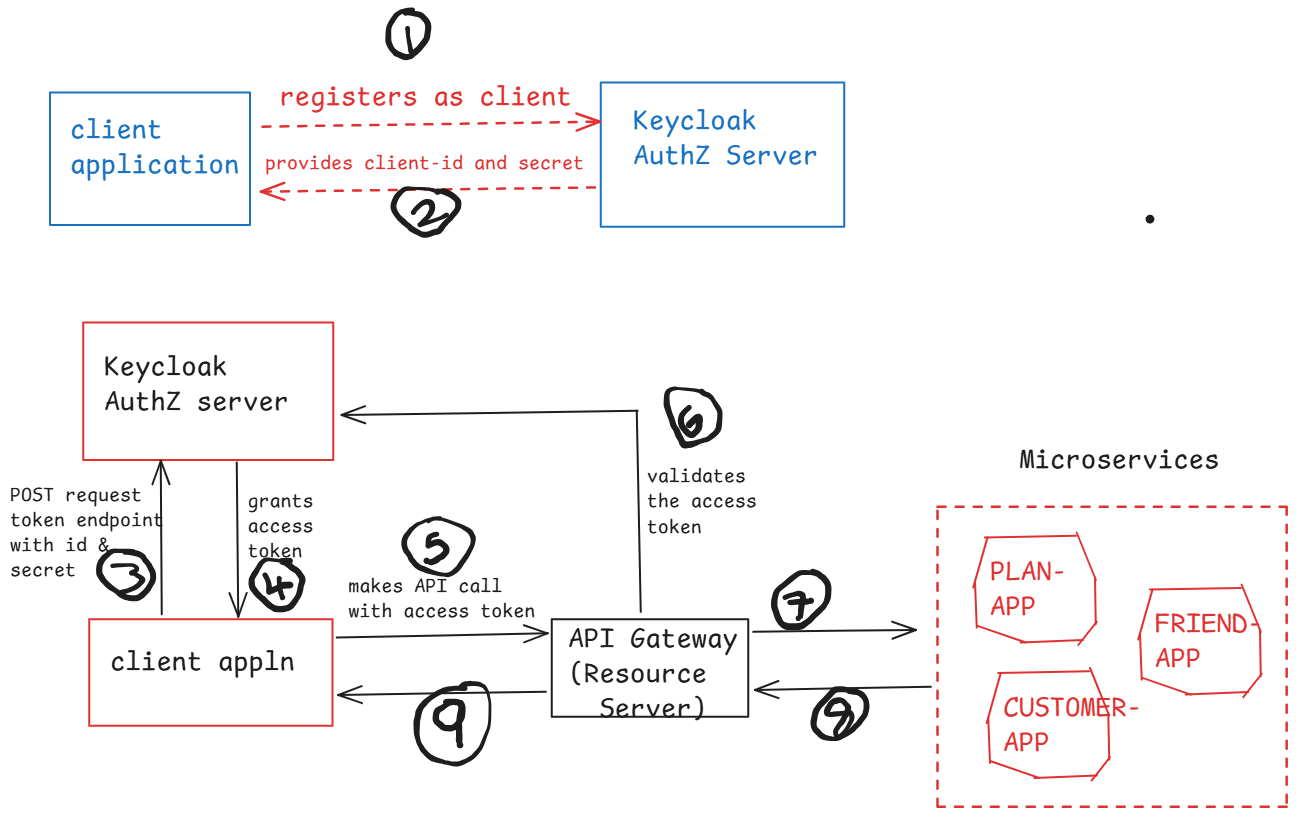
* we can implement security at each microservice.

or

* we can implement security at api gateway.
* if we implement security at each microservice, the code should be duplicated at each microservice and also in future if any changes are required in security configurations, then the changes must be made at each service.
* The changes many take more time and we also need to restart the services. If multiple instances are started for a service, then all those instances must be restarted.
* That’s why we implment the security for the microservices at the gateway.
* We use Oauth2 framework/specification to implement the security for API Gateway application.
* Oauth2 protocol, has defined different grant types for different types of clients to get access token, in order to fetch the data from the resource server with access token.
* Mostly applications use either Authorization code grant type or client credentials grant type of Oauth2 framework, to get the access token.
* client credentials grant type is used for server-to-server communication, where the client(an application/service) accesses the API of the server(an application/service), but not on behalf of a user.
* For example, there is an e-commerce application called ShopEasy and it interacts with a payment processing application called PayFast to handle the payments.
* Here, ShopEasy server needs to interact with the PayFast API to process the payments or for generating invoices, etc.
* So, here the client credentials grant flow steps are,

1. ShopEasy(client) register with the Authz Server of PayFast as a client.
2. PayFast Authz server provides Client credentials to ShopEasy like client-id and client-secret.
3. ShopEasy requests access token from Authz Server, provides its credentials in the request.
4. Now, if client credentials are valid, then PayFast Authz server issues access token.
5. ShopEasy application makes a call to the PayFast API, with this access token.
6. PayFast application validates the access token and will process the request and sends the response to the ShopEasy application.

* We are going to setup the keycloak as Authorization Server and our API Gateway application as Resource server and Postman as a client application.



1. install docker desktop on to your machine
2. start the keycloak authorization server as a docker container.
3. visit <https://www.keycloak.org/getting-started/getting-started-docker> and copy the docker run command and execute it from the command prompt.
4. after executing the docker run command, execute “docker ps” command, and check for the container started or not.
5. open the browser and enter <http://localhost:7080>

username: admin

password: admin

1. click on clients at left side ---> create client
2. Client type: OpenID Connect

Client ID : demo-client-cc

Name: demo app

Description: demo application

Next button

1. Client authentication : on

un-select : standard flow, and direct access grants

select : service accounts roles

1. click on Next button then Save button.

10)click on credentials tab, verify that client secret is generated.

Note: with the above steps, client application is registered with keycloak authz server, and client-id and client-secret are generated.

Setting up API Gateway as Resource Server:

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

1. open pom.xml file, add the below dependencies.

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-oauth2-resource-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-oauth2-jose</artifactId>

</dependency>

1. create a new package, com.example.demo.configuration and create a new class SecurityConfig, like below.

@Configuration

@EnableWebFluxSecurity

**public** **class** SecurityConfig {

@Bean

**public** SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity httpSecurity) {

httpSecurity.authorizeExchange(exchanges -> exchanges.anyExchange().authenticated())

.oauth2ResourceServer(spec -> spec.jwt(Customizer.*withDefaults*()));

httpSecurity.csrf(csrf -> csrf.disable());

**return** httpSecurity.build();

}

}

1. add the below property to the application.yml file.

spring:

security:

oauth2:

resourceserver:

jwt:

jwk-set-uri: <http://localhost:7080/realms/master/protocol/openid-connect/certs>

1. start the cloud config server, service registry, api gateway and plan-app applications.
2. open postman and send a GET request to

<http://localhost:8096/plan-app/plans>

The response will be 401 Unauthorized. It means, Gateway is not allowing the request to plan-app, as the client has not sent access token in the request headers with the name Authorization.

1. Now send a POST request to the token endpoint of keycloak auth server

url: http://localhost:7080/realms/master/protocol/openid-connect/token

click on body tab, choose x-form-url-encoded and add the below key-value pairs.

key value

grant\_type client\_credentials

client\_id demo-client-cc

client\_secret paste the secret value

scope email profile

click on Send button.

1. In response, you will receive access token.
2. copy the access token, now send request to plan-app through Gateway, by adding access token to the Authorization header.
3. <http://localhost:8096/plan-app/plans>

key value

Authorization Bearer paste access token

1. Now you will receive the response data.

<https://github.com/ShekherJava/API_Gateway.git>

Logging in Spring Boot:

* We can write System.out.println() statements in our code, to check whether the control has reached upto a specific point in the code or not.
* But with S.o.p statements, we have have some issues like two threads can’t execute S.o.p statement at a time and the messages are displayed on the console. So, we can’t use these messages for debugging.
* So, we write log statements instead of S.o.p statements in an application.
* Log statements are the records of events that happen in our application.
* log statements contains timestamp information like when the event was happened, what happened at that time, which thread was processing at that time, etc.
* log statements helps us to monitor and analyze our application, and also for troubleshooting and debugging tasks.
* Spring Boot by default uses logging framework as Logback.
* We can also use either Log4j2 or Java Util Logging as a logging framework in spring boot.
* In a spring boot application, we write the log statements using SLFJ API. This api delegates the logging calls to the underlying logging implementation.
* SLF4J (Simple Logging Façade for Java) is not a logging framework. Instead, it is an abstraction for logging frameworks like Logback, Log4j2, etc.
* SLF4J allows the developers to write the code independent of the logging framework.
* Each log statement has a log level and choosing the appropriate log level is crucial for maintaining clarity.
* The log levels are,

TRACE

DEBUG

INFO

WARN

ERROR

* TRACE is used for tracing the internal flow of a program. It is used for very low-level operations.
* For example, like tracing the method entry/exit point, or like detailed application flow like loops or recursive methods.
* DEBUG is used for logging the information that helps developers to debug the application.
* For example, like showing the current state of an object or logging the variable values like the current user who has logged in or the orderid of the order processed etc.

logger.debug(“Current user : {}”, currentUser);

logger.debug(“Order processed with id : {}”, orderId);

* INFO is used to log the general information about the application state.
* For example, like logging the application startup or shutdown events or for recording the successful operations like “Order places” or “File uploaded”, etc.
* WARN indicates potentially harmful situations in the application, but it does not cause shutdown, but might require some attention.
* For example, like some feature is deprecated or some configuration is missing, etc.
* ERROR is used to log error events that disrupt the operation. So, immediate action/attention is required.
* For example, logging the exceptions stack trace or logging the database connection failures etc.
* The default logging behavior in spring boot is,

1.logs are displayed on console

2.The default logging level is INFO

3.Uses logback as the default logging implementation.

* To write a log statement in a spring boot application like in a controlller class or in a service class, etc., then you need to create a Logger object like below.

private static final Logger logger = LoggerFactory.getLogger(classname.class);

For example:

@RestController

public class WelcomeController {

private static final Logger logger = LoggerFactory.getLogger(WelcomeController.class);

@GetMapping(value = “/welcome/{username}”)

public String sayWelcome(@PathVariable String username)

{

logger.info(“The current username : {}”, username);

return “welcome : “ + username;

}

}

* The configuration settings in application.properties file.

#default logging level for the entire application

logging.level.root=INFO

#logging level for specific packages

logging.level.com.example=DEBUG

logging.level.org.springframework.web=ERROR

* logginging.level.root applies to all the loggers in the application, unless they have configured with a specific logging level.
* To write/store the logs in a file,

logging.file.name=application.log

logging.file.path=/logs

Prometheus:

-----------

* Spring Boot Admin server and Prometheus can be used to monitor the application’s health, metrics and other environment details.
* Admin server is used for monitoring and managing the spring boot applications. But Prometheus could be used for monitoring and managing the spring boot applications and also any other framework applications.
* Admin server doesn’t have alert mechanism. But Prometheus has alert mechanism.
* When something unusual happens in the system, then based on the defined conditions, Prometheus triggers notifications. This is called alerting.
* Follow the below steps, to work with prometheus.

1. visit <https://prometheus.io/download/>
2. download [prometheus-2.53.3.windows-amd64.zip](https://github.com/prometheus/prometheus/releases/download/v2.53.3/prometheus-2.53.3.windows-amd64.zip).
3. extract the zip file.
4. prometheus is installed in prometheus-2.53.3-windows-amd64 folder.
5. open prometheus.yml file, and add the below configuration.

# my global config

global:

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

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

scrape\_configs:

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

- job\_name: "PLAN-APP"

metrics\_path: "/actuator/prometheus"

static\_configs:

- targets: ["localhost:8091"]

6. open PLAN-APP pom.xml file, add the below dependencies

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-core</artifactId>

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

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

</dependency>

7.open application.properties file, add the below configurations.

management.endpoints.web.exposure.include=\*

management.endpoint.metrics.enabled=true

management.metrics.export.prometheus.enabled=true

management.endpoint.prometheus.enabled=true

8.start the applications(config server, service register, plan app)

9.start the prometheus.exe

10. open prometheus dashboard

<http://localhost:9090>

Grafana installation:

1. visit <https://grafana.com/grafana/download>
2. choose either 10.x or 11.x version, and enterprise edition.
3. download windows installer file
4. run the installer file.
5. The grafana installed at C:\Program Files\Grafana Labs\Grafana.
6. Goto \bin folder and right click on grafana-server.exe file and run as administrator.
7. start the prometheus server.
8. open grafana admin page

<http://localhost:3000>

login username/password: admin/admin

1. Click on Menu 🡪 connections 🡪 data sources
2. add data source 🡪 select Prometheus 🡪 enter the url : <http://localhost:9090> -🡪 save and test.
3. Dashboards 🡪 Add dashboard 🡪 import dashboard
4. copy the JSON value from <https://github.com/amrutprabhu/grafana-prometheus/blob/main/Spring%20Boot%20Statistics%20Dashboard.json>

and load it the import dashboard.

1. click on Load button, choose DataSource as Prometheus, then click on Import button.

Spring Boot and Kafka integration

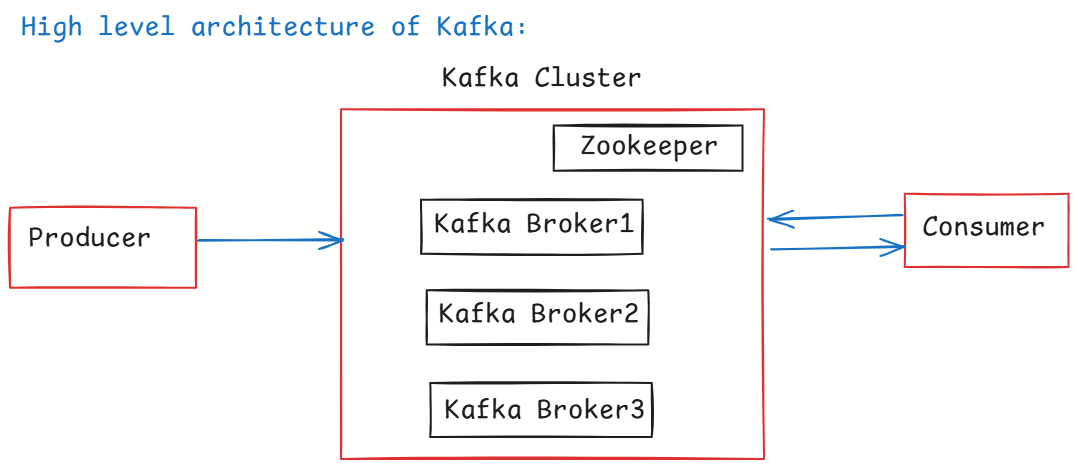
* In Realtime systems, data gets generated continously and applications need to capture this data to perform analysis and also to improve the business.
* Organizations requires some infrastructure to implement,

1. highly-reliable and scalable storage systems to store the real-time data.
2. It should support high throughput to read/write the messages.
3. It should maintain this infrastructure efficiently.

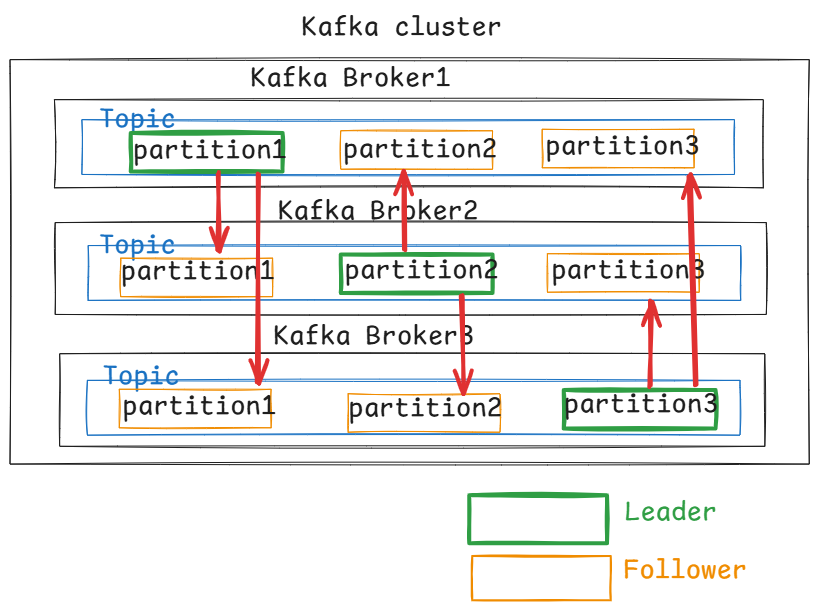
* The organizations can use open source streaming platform called Kafka, without maintaining their own infrastructure.
* Messagging systems such as Kakfa, RabbitMQ, JMS, MSMQ, etc.. follows publish-subscribe model.
* pub-sub model has 4 components.

1. Producer: Application that generates the messages continuously.
2. Message Broker: It is a server which stores all the messages that are published by the producer.
3. Topic: category of messages stored together.
4. Consumer: Application that subscribe to a specific topic and consumes the messages.

* Apache Kafka is an open-source and distributed messaging system.
* Kafka is highly scalable, reliable and it can store large volumes of data.
* Kafka allows message producers to publish millions of messages per second, and also allows message consumers to consumer millions of messages per second.
* Kafka can run on a single server or in a distributed cluster.



* Kafka Broker is a Kafka server and a Kafka cluster is a group of Kafka brokers, and these Kafka brokers many exist at one location or may exist at different geographic locations.
* Zookeeper manages the entire metadata of the Kafka cluster, like the number of brokers and their locations, the topics in each broker, and its partitions, etc..
* A Kafka Broker/Server, contains one or more topics.
* A Topic contains one or more partitions. A partition stores messages.
* While creating a Topic, replication factor is provided, to maintain the backup copies of the Topic in other Kafka brokers in the cluster.
* For example, there are 3 brokers in the cluster then we can not specify the replication factor as 4. Because, always the replication factor must be less than or equals to the number of brokers in the cluster.
* Suppose, a Topic has partitions, then zookeeper elects which broker should have the leader partition and and the follower partitions.



Installing Apache Kafka:

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

1. visit <https://kafka.apache.org/downloads>
2. download kafka\_2.13-3.7.0.tgz file
3. extract the downloaded file into a location.
4. open kafka\_2.13-3.7.0/config folder, copy zookeeper.properties and server.properties files into /bin/windows folder.

start zookeeper:

* open a command prompt(administrator mode)

D:\kafka\_2.13-3.7.0\bin\windows>zookeeper-server-start.bat zookeeper.properties

(zookeeper starts on port 2181)

start kafka server:

* open a command prompt(administrator mode)

D:\kafka\_2.13-3.7.0\bin\windows>kafka-server-start.bat server.properties

(kafka server starts on port 9092)

create a topic:

* open a command prompt(administrator mode)

D:\kafka\_2.13-3.7.0\bin\windows> kafka-topics.bat --create --topic DemoTopic --replication-factor 1 --partitions 1 --bootstrap-server localhost:9092

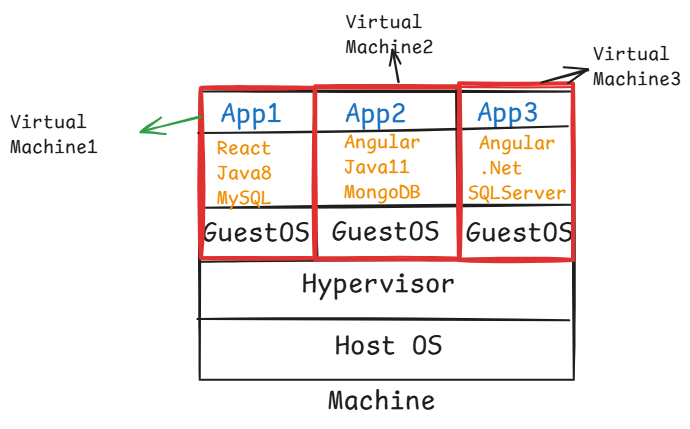
<https://github.com/ShekherJava/SB-Kafka-Producer.git>

<https://github.com/ShekherJava/SB-Kafka-Consumer.git>

=============================================================

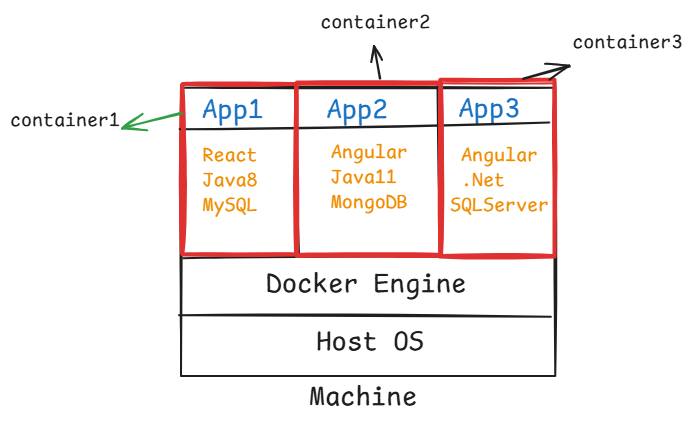
Dockerizing Spring Boot application:

Virtualization:



* Virtualization enables the creation and management of Virtual Machine. It virtualizes the hardware, and allows multiple operating systems to run the same physical machine.
* The physical machine has Host Operationg System and each VM has its own operating system called Guest Operating System.
* Each virtual machine is allocated with CPU, Memory(RAM) and Storage.
* When you boot-up the physical machine, it will take more time because into each VM, its Guest OS must be fully loaded. So it decreases the performance of the physical machine.

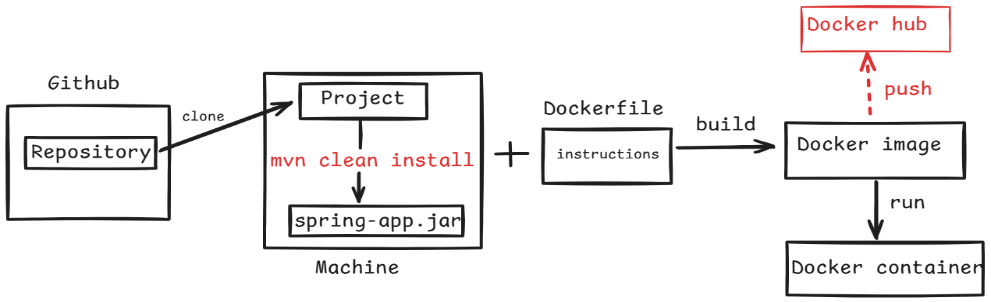
containerization:



* Docker is a containerization platform which enables the creation of lightweight environments called containers.
* containers uses shared Operating System kernal and creates isolated environments for running applications.
* containers are lightweight, because they don’t include the entire OS, they only contain the application code and application dependencies.
* containers are faster because they don’t boot an OS.

Dockerizing Spring Boot application:

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



step-1: create a new spring boot rest application.

* create a rest controller class(HelloController), with

HTTP GET endpoint /hello, to return a welcome message.

step-2: package the application as a jar file.

* Right click on the project 🡪 RunAs 🡪 maven build… 🡪 Goal: clean install 🡪 Run
* Refresh the project

step-3: create Dockerfile in the project directory.

# Use a base image as jdk 17

FROM openjdk:17-jdk-slim

# Copy the Spring Boot jar file into the container

COPY target/SB-Hello-App-0.0.1-SNAPSHOT.jar /usr/app/

# Set the working directory inside the container

WORKDIR /usr/app/

# Expose the port the app runs on

EXPOSE 8080

# Run the Spring Boot application

ENTRYPOINT [ "java","-jar","SB-Hello-App-0.0.1-SNAPSHOT.jar" ]

step-4: start the Docker engine.

* double click on Docker Desktop icon

step-5: Build the docker image

* open command prompt and move to the project location.
* execute the below command

docker build -t my-spring-boot-app .

Note: Here .(dot) refers the current directory where the Dockerfile is located.

step-6: Run the Docker container

* After building the image, you can run the spring boot application inside a Docker container.
* execute the below command

docker run -d -p 8080:8080 my-spring-boot-app

step-7: Test the application.

<http://localhost:8080/hello>

Dockerizing Spring Boot + MySQL

step-1:

create a new Spring Boot Rest application

* create a rest controller class, entity class, service class, repository interface.

step-2:

build the spring boot application as jar

* Right click on the project 🡪 RunAs 🡪 maven build… 🡪 Goal: clean install -DskipTests 🡪 Run
* Refresh the project

step-3: create Dockerfile in the project directory.

# Use a base image as jdk 17

FROM openjdk:17-jdk-slim

# Copy the Spring Boot jar file into the container

COPY target/SB-Book-Rest-App-0.0.1-SNAPSHOT.jar /usr/app/

# Set the working directory inside the container

WORKDIR /usr/app/

# Expose the port the app runs on

EXPOSE 8080

# Run the Spring Boot application

ENTRYPOINT [ "java","-jar","SB-Book-Rest-App-0.0.1-SNAPSHOT.jar" ]

step-4: start the Docker engine.

* double click on Docker Desktop icon

step-5: Build the docker image

* open command prompt and move to the project location.
* execute the below command

docker build -t my-spring-book-rest-app .

Note: Here .(dot) refers the current directory where the Dockerfile is located.

step-6:

create a docker network with the below command.

* docker network create sb-mysql-network
* To verify the network is created or not,
* docker network ls

step-7: create docker-compose.yml in the project directory.

version: "3.8"

services:

application:

image: my-spring-book-rest-app

container\_name: spring-container

ports:

- "8080:8080"

networks:

- sb-mysql-network

depends\_on:

- mysqldb

deploy:

restart\_policy:

condition: on-failure

max\_attempts: 10

mysqldb:

image: mysql:8.0

container\_name: mysqldb-container

ports:

- "3306:3306"

networks:

- sb-mysql-network

environment:

- MYSQL\_ROOT\_PASSWORD=root

- MYSQL\_DATABASE=test

networks:

docker-mysql:

name: sb-mysql-network

external: true

step-8: execute the below command from the command prompt at project directory.

* docker compose up -d

step-9: open postman to test the api endpoints.

POST <http://localhost:8080/add>

add the below sample JSON in request body.

{

“bookId”: 10101,

“bookName”: “Java”,

“author”: “Gosling”

}

GET <http://localhost:8080/fetch>

Note: If you want to stop the containers started by docker compose, execute the below command.

* docker compose down

<https://github.com/ShekherJava/SB-Book-Rest-App.git>

Spring Boot + MongoDB

* Download mongodb from <https://www.mongodb.com/try/download/community>
* mongodb-windows-x86\_64-7.0.16-signed.msi file will be downloaded.
* install mongodb by double click on the downloaded file.
* MongoDB is a **NoSQL database** that stores data in a flexible, JSON-like format called **documents**. It is widely used for its ease of use and scalability.
* **Database**: A container for collections.
* **Collection**: Similar to a table in relational databases, it is a group of documents.
* **Document**: A single record in a collection, stored in a JSON-like format (BSON internally).

<https://github.com/ShekherJava/SB-MongoDB-CRUD.git>

Testing CRUD operations using POSTMAN:

1. create employee

POST <http://localhost:8082/api/employees>

{

“id”: 101,

“name”: “John”,

“salary”: 4000

}

Note: create multiple employees.

1. Get all employees

GET <http://localhost:8082/api/employees>

1. Get employee by ID

GET [http://localhost:8082/api/employees/{id}](http://localhost:8082/api/employees/%7bid%7d)

1. Update employee

PUT [http://localhost:8082/api/employees/{id}](http://localhost:8082/api/employees/%7bid%7d)

{

“name”: “Waden”,

“salary”: 8000

}

1. Delete employee

DELETE [http://localhost:8082/api/employees/{id}](http://localhost:8082/api/employees/%7bid%7d)