Difference Between Spring @Qualifier and @Primary Annotations

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

@Qualifier Annotation:

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

@Qualifier annotation is used to resolve ambiguity when multiple beans of the same type are present in a Spring application context.

Ex:

public interface MessageService {

public void sendMsg(String message);

}

public class EmailService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

public class TwitterService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

public class SMSService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

public interface MessageProcessor {

public void processMsg(String message);

}

public class MessageProcessorImpl implements MessageProcessor {

private MessageService messageService;

// setter based DI

@Autowired

@Qualifier("twitterService")

public void setMessageService(MessageService messageService) {

this.messageService = messageService;

}

// constructor based DI

@Autowired

public MessageProcessorImpl(@Qualifier("twitterService") MessageService messageService) {

this.messageService = messageService;

}

public void processMsg(String message) {

messageService.sendMsg(message);

}

}

@Primary Annotation: which should be injucted no qualifier required for defult bean

The @Primary annotation is used to indicate a default bean when multiple beans of the same type are present. If multiple beans are eligible for autowiring and none of

them are explicitly specified using @Qualifier, the bean marked with @Primary will be selected by default.

@Component

@Primary

public class EmailSender implements MailSender {

// ...

}

@Component

public class SmsSender implements MailSender {

// ...

}

@Service

public class NotificationService {

@Autowired

private MailSender mailSender;

// ...

}

the @Qualifier annotation is used to resolve ambiguity when multiple beans of the same type exist,

whereas the @Primary annotation designates a default bean to be injected when no qualifier is specified

what are http statuscodes in spring boot:

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

200 OK The request was successfully completed

201 Created A new resource was successfully created.

400 Bad Request The request was invalid.

401 Unauthorized The request did not include an authentication token or the authentication token was expired.

403 Forbidden The client did not have permission to access the requested resource.

404 Not Found The requested resource was not found.

405 Method Not Allowed

500 Internal Server Error

503 Service Unavailable

How to handle exception in java and spring boot?

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

Java:

public class CustomerAlreadyExistsException extends RuntimeException {

private String message;

public CustomerAlreadyExistsException() {}

public CustomerAlreadyExistsException(String msg) {

super(msg);

this.message = msg;

}

}

public String addCustomer(Customer customer)

{

Customer existingCustomer

= customerRespository.findById(customer.getId())

.orElse(null);

if (existingCustomer == null) {

customerRespository.save(customer);

return "Customer added successfully";

}

else

throw new CustomerAlreadyExistsException(

"Customer already exists!!");

}

Spring:

@Data

@AllArgsConstructor

@NoArgsConstructor

public class ErrorResponse {

private int statusCode;

private String message;

public ErrorResponse(String message)

{

super();

this.message = message;

}

}

@ExceptionHandler: method level

@ExceptionHandler(value = CustomerAlreadyExistsException.class)

@ResponseStatus(HttpStatus.CONFLICT)

public ErrorResponse handleCustomerAlreadyExistsException(CustomerAlreadyExistsException ex) {

return new ErrorResponse(HttpStatus.CONFLICT.value(), ex.getMessage());

}

@ControllerAdvice: Class level Globalli

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(value = NoSuchCustomerExistsException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public @ResponseBody ErrorResponse handleException(NoSuchCustomerExistsException ex) {

return new ErrorResponse(HttpStatus.NOT\_FOUND.value(), ex.getMessage());

}

}

What is Loosely Coupling and Tighly Coupling :

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

public class Engine {

public int start ( ) {

// logic

return 1;

}

}

public class Car {

public void drive() {

Engine eng = new Engine ( );

int start = eng.start();

if (start >= 1) {

System.out.println("Journey Started");

}

// start the journey

}

}

if Engine class constructor modified then our Car class will be effected

If we use any approach from above then Car class will become tightley coupled with Engine class. That is not recommended.

Loosely Coupling:

=> Loosely coupling means without creating Object and without Inheriting properties we should be able to access one class method in another class.

=> If we make any changes in Engine class then Car class shouldn't be effected then we can say our classes are loosely coupled.

what is IOC Container:

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

-> IoC stands for Inversion Of Control.

-> IoC is responsible for Dependency Injection in Spring Applications.

-> Dependency Injection means creating and injecting dependent bean objects into target bean classes.

Note: IoC container will manage life cycle of Spring Beans.

Note : We need to provide " Java classes + Bean Configuration " as input for IOC then IOC will perform DI and provides Spring Beans which are ready to use.

what are the scopes in spring ?

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

=> Scope represents how many objects should be created for a Spring Bean

=> To represent bean scope we will use "scope" attribute

<bean id="id" class="pkg.classname" scope = "singleton | prototye | request | session " />

-> Singleton scope means only one object will be created for the class in IOC Container. This is default scope of spring bean.

-> Prototype scope means every time new object will be created.

Note: request & session scopes are related to Spring Web MVC Module.

Why Spring Bean is by default Singleton ?

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

=> To save memory of JVM spring team made singleton as default scope for the spring beans.

what is difference between byName and byType and Contructor?

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

Autowiring will not work bydefault, We have to enable autowiring on target bean like below.

<bean id="id" class="pkg.Classname" auto-wire="byName | byType | constructor | no " />

byName: IoC will identify dependent bean object based on bean id or bean name.

========

Ex:

public class Car {

private IEngine eng;

public void setEng(IEngine eng) {

System.out.println("setEng ( ) method called....");

this.eng = eng;

}

public void drive() {

int status = eng.start();

if (status >= 1) {

System.out.println("Journey Started..");

} else {

System.out.println("Engine Trouble");

}

}

}

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="eng1" class="in.ashokit.beans.PetrolEngine" />

<bean id="eng" class="in.ashokit.beans.DieselEngine" />

<bean id="car" class="in.ashokit.beans.Car" autowire="byName"/>

</beans>

Note: In the above example Car class variable name is matched with 'DieselENgine' bean id hence DieselEngine obj will be injected into Car.

byType: IoC will identity dependent bean object based on data type of the variable in Target class.

private IEngine eng; ====> data type of eng is IEngine which is an interface

=> If one interface having 2 implementations then there is a chance of getting Ambiguity problem. To overcome that we need to use 'autowire-candiate' attribute.

autowire-candidate="false" ====> Not Eligible for Autowiring

autowire-candidate="true" ===> Eligible for Autowiring

Note: As an alternate for "autowire-candidate=true" we can use "primary=true" to consider bean for Autowiring.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="eng2" class="in.ashokit.beans.PetrolEngine" autowire-candidate="false"/>

<bean id="eng1" class="in.ashokit.beans.DieselEngine" autowire-candidate="true"/>

<bean id="car" class="in.ashokit.beans.Car" autowire="byType" />

</beans>

(or)

constructor: => It is used to perform Autowiring by calling target class constructor

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="eng2" class="in.ashokit.beans.PetrolEngine" primary="true" />

<bean id="eng1" class="in.ashokit.beans.DieselEngine" />

<bean id="car" class="in.ashokit.beans.Car" autowire="constructor" />

</beans>

Note: Autowiring is applicable for Reference Type variable (not applicable for primitive types)

what is spring bean life cycle?

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

=> Life cycle means starting to ending or birth to death

=> Spring Bean object creation and object destruction will be taken care by IOC container.

=> Spring Bean Life Cycle will be managed by Ioc Container.

=> We can perform some operations using Bean Life Cycle Methods

init ( ) --------> initialization logic

destroy ( ) -----> destruction logic

=> Spring Bean Life Cycle methods we can execute in 3 ways

1) XML Approach (Declarative)

2) Progrmmatic approach

3) Annotations

<bean id="motor" class="in.ashokit.beans.Motor"

init-method="start"

destroy-method="stop"/>

init-method ==> It represents the method which should be called after bean obj created

destroy-method ==> It represents the method which should be called when bean obj removing from IoC

Programmatic:

=> We need to implement predefined interfaces provided by Spring Framework

1) InitializingBean ---> afterPropertiesSet ( )

2) DisposableBean ---> destroy ( )

public class Motor implements InitializingBean, DisposableBean {

}

Annotation:

@PostConstruct -----> It represents init method

@PreDestroy ----> It represents destroy method

what are the spring boot annotations

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

@SpringBootApplication: @Configuration+@EnableAutoConfiguration+@ComponentScan

@Component:

@Service

@Repository

@Controller:a controller in a Spring MVC application.

@RestController: combines @Controller and @ResponseBody It is used to create RESTful web services.

@RequestMapping: Used to map web requests to specific handler methods or classes.

Ex:@RequestMapping("/api")

@Autowired

@Value: Used to inject property values into Spring beans.

@Value("${my.property}")

private String myProperty;

@RequstParam and @PathParam difference between:

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

@RequestParam: query parameter or form parameter

Purpose: Used to extract query parameters from the URL.

Location in URL: Parameters are part of the query string in the URL, following a ? and separated by &.

Usage: Typically used with HTTP GET requests to retrieve parameters.

@GetMapping("/greet")

public String greet(@RequestParam String name) {

return "Hello, " + name;

}

@PathVariable

Purpose: Used to extract values from the URI path itself.

Location in URL: Parameters are part of the URL path, defined by curly braces {} in the path pattern.

Usage: Typically used with HTTP GET requests to extract variables embedded directly in the URL path.

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

public String greet(@PathVariable String name) {

return "Hello, " + name;

}

@RequestParam: Extracts parameters from the query string (e.g., ?key=value).

@PathVariable: Extracts parameters directly from the URL path (e.g., /path/{value}).

what is difference between monolithik and microservice?

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

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

Monolith Architecture

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

=> Develop all functionalities in single app

=> Application will be packaged as one fat jar / fat war

=> App will be deployed in single server

==========

Drawbacks

==========

1) Single Point of failure

2) Re-Deploy entire app

3) Maintenence of the app

4) Burden on server

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

Microservices

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

=> Microservices is a not a technology

=> Microservices is not a framework

=> Microservices is not an API

=> Microservices is an architectural design pattern

=> Microservices design pattern is universal

=> The main aim of microservices is to develop app with loosely coupling

=> Microservices based application means collection of rest apis.

=> Microservices means independetly deployable and executable services.

=========

Benefits

=========

1) Loosely Coupled

2) Easy Maintenence`

3) Load will be distributed

4) Technology Independency

5) High Availability

===========

Challenges

===========

1) Bounded Context (deciding no.of rest apis to develop)

2) Duplicate Configuration

3) Visibility

Pass by Value and Pass by Reference:

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

package com.app.core;

public class PassByReferencePassByValue {

public static void main(String[] args) {

//pass by value for primitives

int a=25;

passByValue(a);

System.out.println("after method called"+a);

//pass by reference for non primitives

int[] arr= {123,134};

passByReference(arr);

System.out.println("after method reference called"+arr[0]);

}

public static void passByValue(int num) {

System.out.println("befor"+num);

num=35;

System.out.println("befor"+num);

}

public static void passByReference(int[] num) {

System.out.println("befor"+num);

num[0]=35;

System.out.println("befor"+num);

}

}

Immutability class:

//if the class is final optional in this case

//inside all the data member should be private

//only using get method ,should not use the set methods

package com.app.core;

public final class immutabilityclass {

private String data;

public immutabilityclass(String data) {

this.data=data;

}

public String getData() {

return data;

}

public static void main(String[] args) {

immutabilityclass im=new immutabilityclass("Ankaiah");

System.out.println(im.getData());

}

}

Sigleton Class:

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

Private constructor to restrict instantiation of the class from other classes.

Private static variable of the same class that is the only instance of the class.

Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

package com.app.core;

public class SigletonClass {

private static final SigletonClass instance = new SigletonClass();

// private constructor to avoid client applications using the constructor

private SigletonClass(){}

public static SigletonClass getInstance() {

return instance;

}

}

naturalOrder:

List < String > sortedList=list.stream().sorted(Comparator.naturalOrder()).coolect(Collectors.toList());

List < String > sortedList=list.stream().sorted((e1,e2)->e1.compareTo(e2)).collect(Collectors.toList());

List < String > sortedList=list.stream().sorted().collect(Collectors.toList());

Disending order:

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

List < String > sortedList=list.stream().sorted((e1,e2)->e2.compareTo(e1)).collect(Collectors.toList());

List < String > sortedList=list.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());

Custom Object Sorting:

List < Employee > sortedList=list.stream().sorted(Comparator.comparingInt(Employee::getSalary).collect(Collectors.toList());

reversed:

List < Employee > employeesSortedList2 = employees.stream()

.sorted(Comparator.comparingLong(Employee::getSalary).reversed()).collect(Collectors.toList());

Logging Architecture:

1) Logger

2) Layout

3) Appender

1. Logger:

=> Logger is a class which is used to generate log msgs

logger.trace()

logger.debug()

logger.info()

logger.warn()

logger.error()

1. Layout:

=> Layout represents log msg format

3) Appender:

Appender represents destination to store log msgs

Ex : ConsoleAppender (will print msgs on console)

Ex : FileAppender (will store msgs in file)

Logging Levels:

logging.level.root = debug

logging.file.name=ashokit.log

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

=> In realtime , multiple users will access our application on daily basis.

=> If we maintain single log file then lot of data will be stored in that file

=> After few days/months we can't open that file because of huge size.

=> To overcome this problem we will use Rolling concept in logging.

Rolling File Appender :

1) Time Based Rolling

Ex: Create New Log file for every 24 hours (day wise)

2) Size Based Rolling

Ex: Store only 1 GB data in one log file

=> We can configure rolling in 2 ways

1) Properties / yml file

2) xml configuration (recommended)

ex : logback.xml

Log Monitoring Tools:

1) ELK / EFK - Open source s/w

1. Splunk - Commercial s/w

Circuit Breaker Design Pattern:

> Circuit Breaker is a design pattern in Microservices

-> Circuit Breaker is used to implement fault-tolerance systems

-> Fault-tolerance systems are also called as resillence systems

-> Fault-tolerance system means when main logic is failed to execute then we should execute fallback logic to process client request

Usecase

===========

=> Get data from redis, if redis logic is failing then we should get data from database

Note: If redis logic is failing for 3 requests continuosly then execute db logic for 30 mins. After 30 mins re-try for redis logic execution if it is working then execute redis logic only. If 3 re-try executions failed with redis then execute db logic for next 30 mins.

<dependency>

<groupId>io.pivotal.spring.cloud</groupId>

<artifactId>spring-cloud-services-starter-circuit-breaker</artifactId>

</dependency>

-> Write @EnableHystrix annotation at boot start class

public class DataRestController {

@GetMapping("/data")

@HystrixCommand(

fallbackMethod = "getDataFromDB",

commandProperties = {

@HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="3"),

@HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds", value="10000"),

@HystrixProperty(name="circuitBreaker.enabled", value="true")

}

)

public String getDataFromRedis() {

System.out.println("\*\*getDataFromRedis() method called\*\*");

if (new Random().nextInt(10) <= 10) {

throw new RuntimeException("Redis Server Is Down");

}

// logic to access data from redis

return "data accessed from redis (main logic) ....";

}

public String getDataFromDB() {

System.out.println("\*\*getDataFromDB() method called\*\*");

// logic to access data from db

return "data accessed from database (fall back logic) ....";

}

}

Config Server App:

=> To externalize properties from the application we can use Cloud Config Server

Config Server App:

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

@EnableConfigServer----------Spring Boot Starter Class

Configure below properties in application.yml file

spring:

cloud:

config:

server:

git:

uri: https://github.com/ashokitschool/configuration\_properties

clone-on-start: true

management:

security:

enabled: false

Config Server Client Development:

a) web-starter

b) config-client

c) dev-tools

if we want ready configuration using config server property url and read the data from external library

server:

port: 9090

spring:

config:

import: optional:configserver:http://localhost:8080

application:

name: greet

Sleuth & Zipkin: Distiributed tracing

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

Sleuth & Zipkin

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

-> Microservices application means several REST APIs will be available

-> As part of application execution one Rest API can communicate with another REST API

-> When we send request from UI, it will process by Multiple REST APIs with Interservice communication

\*\*\* How we can understand which rest api is taking more time to process our request ? \*\*\*

-> If we add Sleuth dependency in REST API then it will add span-id and trace-id for log messages

-> For every request once trace-id will be generated by Sleuth

-> If one request is processing multiple REST API then Sleuth will use same span-id for REST APIs to generate log message

-> Trace-id is specific to one REST API

-> By using span-id and trace-id we can understand which REST api has taken more time process request

-> To monitor span-id and trace-id details we will use ZipKin server

-> Zipkin server is providing user interface (UI) to monitor all the details

Note: The REST APIs which are having sleuth dependency should register with Zipkin server

Note: By using Sleuth and Zipkin we achieve Distributed Log Tracing

Steps to work with Sleuth and Zipkin:

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

2) Download zipkin-server jar file (https://zipkin.io/pages/quickstart)

3) Run zipkin-server using "java -jar <zipkin-jar-filename"

Note: Zipkin server runs on 9411 port

4) Run spring boot applications and send a request to rest controller method

5) Verify boot application logs display in console (span-id and trace-id will be attached to logs)

6) Go to Zipkin server dashboard and monitor event details

( URL : http://localhost:9411 )

API Gateway:

-> API Gateway will act as mediator between client requests & backend apis

-> API Gateway will provide single entrypoint to access our backend apis

-> In Api Gateway we will write mainley below 2 types of logics

1) Filters

2) Routing

-> Filters are used to execute some logic before request processing and after request processing

-> Routing is used to tell which request should go to which REST API

-> In Spring Cloud, we have 2 options to create API Gateway

1) Zuul Proxy (old approach)

2) Spring Cloud Gateway (latest approach)

Note: Zuul Proxy is not supported by latest versions of spring boot

++++++++++++++++++++++++++++++++++++++++

Working with Spring Cloud API Gateway

++++++++++++++++++++++++++++++++++++++++

1) Create Spring boot application with below dependencies

-> web-stater

-> eureka-client

-> cloud-gateway

-> devtools

2) Configure @EnableDiscoveryClient annotation at boot start class

3) Configure API Gateway Routings in application.yml file like below

-----------------------------application.yml file---------------------------------

spring:

cloud:

gateway:

discovery.locator:

enabled: true

lowerCaseServiceId: true

routes:

- id: welcome-api

uri: lb://WELCOME-API

predicates:

- Path=/welcome

- id: greet-api

uri: lb://GREET-API

predicates:

- Path=/greet

application:

name: CLOUD-API-GATEWAY

server:

port: 3333

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

In API gateway we will have 3 types of logics

1) Route

2) Predicate

3) Filters

-> Routing is used to defined which request should be processed by which REST API in backend. Routes will be configured using Predicate

-> Predicate : This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters.

-> Filters are used to manipulate incoming request and outgoing response of our application

Note: Using Filters we can implement security also for our application.

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

@Component

public class MyPreFilter implements GlobalFilter {

private Logger logger = LoggerFactory.getLogger(MyPreFilter.class);

@Override

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

logger.info("MyPreFilter :: filter () method executed...");

// Accessing HTTP Request information

ServerHttpRequest request = exchange.getRequest();

HttpHeaders headers = request.getHeaders();

Set<String> keySet = headers.keySet();

keySet.forEach(key -> {

List<String> values = headers.get(key);

System.out.println(key +" :: "+values);

});

return chain.filter(exchange);

}

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++

-> We can validate client given token in the request using Filter for security purpose

-> We can write request and response tracking logic in Filter

-> Filters are used to manipulate request & response of our application

-> Any cross-cutting logics like security, logging, moniroing can be implemented using Filters

Service Registry

++++++++++++++++

-> Service Registry acts as DB of services available in the project

-> It provides the details of all the services which are registered with Service Registry

-> We can identify how many services available in the project

-> We can identify how many instances available for each service

-> We can use "Eureka Server" as service registry

-> Eureka Server provided by "Spring Cloud Netflix" library

Services

+++++++++

-> Services means REST APIs / Microservices

-> Services contains backend business logic

-> In the project, some services will interact with DB

-> In the project, some services will interact with third party REST API ( external communication )

-> In the project, some services will interact with another services with in the project

( inter-service communication )

-> For inter-service communication we will use feign-client

-> To distribute the load, we can run one service with Multiple Instances (Load Balancing)

Steps to develop Service Registry Application (Eureka Server):

a) EurekaServer (spring-cloud-starter-netflix-eureka-server)

b) web-starter

c) devtools

server:

port: 8761

eureka:

client:

register-with-eureka: false

Admin Server Project:

a) web-starter

b) devtools

c) admin-server (codecentric)

@EnableAdminServer annotation---starter class level

Admin client required for access in visible the client application

Example APllication for Microservices:

1) Create Spring Boot application with below dependencies

- eureka-discovery-client

- starter-web

- devtools

- actuator

- sleuth

- zipkin

- admin-client

2) Configure @EnableDiscoveryClient annotation at start class

3) Create RestController with required method

4) Configure below properties in application.yml file

-----------------------------------application.yml-----------------------------------------

server:

port: 9090

spring:

application:

name: GREET-API

boot:

admin:

client:

url: http://localhost:8080/

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka

management:

endpoints:

web:

exposure:

include: '\*'

Steps To Develop WELCOME-API

+++++++++++++++++++++++++++++

1) Create Spring Boot application with below dependencies

- web-starter

- devtools

- eureka-discovery-client

- fegin-client

- admin-client

- zipkin-client

- sleuth

- actuator

2) Configure @EnableDiscoveryClient & @EnableFeignClients annotations at boot start class

3) Create FeignClient to access GREET-API

@FeignClient(name = "GREET-API")

public interface GreetApiClient {

@GetMapping("/greet")

public String invokeGreetApi();

}

4) Create RestController with required method

Note: In Rest Controller we should have logic to access another REST API (GREET-API)

-> For Interservice Communication we will use FeignClient

-> Using FeginClient we can make rest call to another service using name of the service (no need of url)

-> FeginClient will get service URL from service-registry based on service-name

@RestController

public class WelcomeRestController {

private Logger logger = LoggerFactory.getLogger(WelcomeRestController.class);

@Autowired

private GreetApiClient greetClient;

@GetMapping("/welcome")

public String welcomeMsg() {

logger.info("welcomeMsg() execution - start");

String welcomeMsg = "Welcome to Ashok IT..!!";

String greetMsg = greetClient.invokeGreetApi();

logger.info("welcomeMsg() execution - end ");

return greetMsg + ", " + welcomeMsg;

}

}

5) Configure below properties in application.yml file

server:

port: 9091

spring:

application:

name: WELCOME-API

boot:

admin:

client:

url: http://localhost:1111/

management:

endpoints:

web:

exposure:

include: '\*'

6) Run WELCOME-API project (it should register in Eureka and Admin server)

7) Send Request to welcome-api (it should final response)

8) Verify Zipkin Server Dashboard for log tracing

++++++++++++++++++++++++++++++++++++++++++++++++++++++

-> We are running Service Registry project with Eureka Server on 8761 port number

-> Eureka Discovery Client applications are auto-registering with Eureka Server when port is 8761

-> If we change Eureka Server port number then we have to register Eureka Client application with Eureka Server using below property in application.yml file

eureka:

client:

serviceUrl:

defaultZone: http://localhost:9090/eureka

Note: We should configure this property in eureka client application yml file

++++++++++++++++++++++++++++++++++++++++++++++++++++++

GREET API URL : DESKTOP-BDG00U7:GREET-API:9090/

WELCOME API URL : DESKTOP-BDG00U7:WELCOME-API:9091/

Spring Boot with Redis Cache Integration

-> Every application will interact with database to store and retrieve data

-> In DB we will maintain data in tables (Rows & Columns)

-> DB tables are divided into 2 types

1) Transactional Tables

2) Non-Transactional Tables

-> If our application is perfoming INSERT / UPDATE / DELETE operations in table then it is called as Transactional Table.

-> If our application is performing only SELECT operation in the table then it is called as Non-Transactional table.

Q) What is Local cache & What is distributed cache ?

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

-> If we implement cache with in one application then it called as local cache.

-> If we want to use cache data in multiple applications then we should go for distributed cache (Ex: Redis cache)

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

Q) What is Redis?

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

-> Redis is an open source cache which is used to store the data in the form key-value pair.

-> Multiple applications can connect to Redis cache to access the data.

spring:

data:

redis :

url : redis-13948.c264.ap-south-1-1.ec2.redns.redis-cloud.com

port: 13948

username: default

password: SRVREda6KkEDf7dSfAsRTKo8dXukszTb

1) spring-boot-starter-web

2) spring-boot-starter-redis

3) project lombok

4) devtools

5) swagger & swagger-ui

2) Configure JedisConnectionFactory bean with Redis Server details & inject JedisConnectionFactory bean into RedisTemplate bean

1. Create Binding class class
2. Create Service Class with required methods. Inject RedisTemplate into Repository bean and get HashOperations object to perform redis operations.

@RedisHash("statesHash")

@Component

**public** **class** RedisConfig {

@Value("${spring.data.redis.url}")

**private** String url;

@Value("${spring.data.redis.port}")

**private** Integer port;

@Value("${spring.data.redis.username}")

**private** String username;

@Value("${spring.data.redis.password}")

**private** String pwd;

@Bean

**public** JedisConnectionFactory jedisConnectionFactory() {

RedisStandaloneConfiguration serverConfig =

**new** RedisStandaloneConfiguration(url, port);

serverConfig.setUsername(username);

serverConfig.setPassword(pwd);

JedisClientConfiguration clientConfig = JedisClientConfiguration.*builder*().build();

**return** **new** JedisConnectionFactory(serverConfig, clientConfig);

}

}

Spring Boot MVC:

Spring Web MVC Architecture:

1) Dispatcher Servlet : It acts as a front controller

2) HandlerMapper : It will identify which request should be processed by which controller and which method

3) Controller : It will handle request and decides response to send using ModelAndView object.

4) ModelAndView : Model represents data in key-value format. View Represents logical file name to display.

5) View Resolver : It is used to identify physical location of view files

1. View : It is used to render model data on view file.

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

Query Paramters / Request Params

=> To send data to server in URL

=> Key Value format

=> will present At end of the URL

=> Starts with ? symbol

=> Will be seperated by & symbol

URL : [www.ashokitech.com/course?name=sbms&trainer=ashok](http://www.ashokitech.com/course?name=sbms&trainer=ashok)

=> To Read query params we will use @RequestParam annotation

=================================Query Param Example==============================

@Controller

public class BookController {

// http://localhost:8080/msg?name=ashok

@GetMapping("/msg")

public ModelAndView getMsg(@RequestParam String name) {

String msgTxt = name + ", Good Evening";

ModelAndView mav = new ModelAndView();

mav.addObject("msg", msgTxt);

mav.setViewName("index");

return mav;

}

// http://localhost:8080/book?name=spring&author=johnson

@GetMapping("/book")

public ModelAndView getBookData(@RequestParam String name, @RequestParam String author) {

System.out.println("Name :: " + name);

System.out.println("Author ::" + author);

ModelAndView mav = new ModelAndView();

mav.addObject("msg", name + " By " + author + " is out of stock...");

mav.setViewName("index");

return mav;

}

}

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

Path Paramters / URI Params

=> To send data to server in URL

=> It will represent data directley (no keys)

=> Can present anywhere in uRL

=> Starts with / symbol

URL : www.ashokitech.com/course/sbms/

=> To Read query params we will use @PathVariable annotation

@Controller

public class CarController {

// http://localhost:8080/car/101/hyd

@GetMapping("/car/{carId}/hyd")

public ModelAndView getCarColor(@PathVariable Integer carId) {

ModelAndView mav = new ModelAndView();

String color = null;

if (carId >= 100) {

color = "Red";

} else {

color = "Black";

}

mav.addObject("msg", "Car Color is :" + color);

mav.setViewName("index");

return mav;

}

// http://localhost:8080/stock/benz/location/hyd

@GetMapping("/stock/{brand}/location/{loc}")

public ModelAndView getCarStock(@PathVariable String brand, @PathVariable String loc) {

ModelAndView mav = new ModelAndView();

mav.addObject("msg", "In " + loc + " " + brand + " cars Out Of Stock");

mav.setViewName("index");

return mav;

}

}

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

We can develop controller methods in 2 ways

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

1) By Taking ModelAndView as return type

2) By taking String as return type

====================================2 approches=================================================

@Controller

public class MyController {

@GetMapping("/welcome")

public ModelAndView getWelcomeMsg(@RequestParam String name) {

String msgTxt = name + ", Welcome to Ashok IT..";

ModelAndView mav = new ModelAndView();

mav.addObject("msg", msgTxt);

mav.setViewName("index");

return mav;

}

@GetMapping("/greet")

public String getGreetMsg(@RequestParam String name, Model model) {

model.addAttribute("msg", name+", Good Evening...!!");

return "index";

}

}

Note: Method return type is string which represents logical view name. Model is used to send data from controller to UI in key-value format.

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

How to configure Jetty as Embedded Server ?

1) Exclude starter-tomcat from starter-web dependency

2) Add jetty-starter in pom.xml file

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

</dependency>

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

How to send direct response from Spring Controller without using View Pages ?

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

=> By using @ResponseBody annotation in Spring Controller class method we can send direct response to client.

@Controller

public class MessageController {

@GetMapping("/welcome")

@ResponseBody

public String getWelcomeMsg() {

return "Welcome to Ashok IT..!!";

}

@GetMapping("/greet")

public String getGreetMsg(Model model) {

model.addAttribute("msg", "Good Evening");

return "index";

}

}

What is Interceptor in Spring Web MVC ?

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

-> We can use Interceptor to perform pre-processing and post-processing of every request

Pre-Processing : Before Request Procesing by Controller method

Post-Processing : After request processed by controller method

-> Using Interceptor we can trap each and every request

Use case for Interceptor

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

1) Calculate Each Request processing time

2) Log Each Request and Response details

3) Request Authentication etc...

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

Exception Handling in Spring Web MVC

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

-> Exception means un-expected and un-wanted situation

-> Exception distrubs normal flow of our application execution

-> When exception occurs then our program will terminate abnormally

-> As a developer we should handle exception to achieve graceful termination of our application.

-> To handle exceptions, Java provided below keywords

1) try

2) catch

3) throw

4) throws

5) finally

=> To handle Exceptions in Spring Web MVC application then we can create a method and we can use below annotation

@ExceptionHandler

=> When exception occurs then we will redirect user to error page like below.

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(value = Exception.class)

public ModelAndView handleAE(Exception ex) {

ModelAndView mav = new ModelAndView();

mav.setViewName("page");

return mav;

}

}

Note : here error represents our error page which display some message to client to try after sometime.

Java Stream APi programs:

count of occurence in the given String:

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

String iter=

Map<String,long> map=Arrays.stream(input.split(regax:"")).collect(Collectors.groupingBy(Function.identity(),Collectors.counting())));

Duplicate Elements:

List<String> list=Arrays.stream(input.split(regax:"")).collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))).entryset().stram().

filter(x->x.getValue()>1).map(Map.entry::getKey),collect(Collectore.toList()))

Unique identity:

List<String> list=Arrays.stream(input.split(regax:"")).collect(Collectors.groupingBy(Function.identity(),Collectors.counting()))).entryset().stram().

filter(x->x.getValue()==1).map(Map.entry::getKey),collect(Collectore.toList()))

First Non Repeatable Element in String;

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

List<String> list=Arrays.stream(input.split(regax:"")).collect(Collectors.groupingBy(Function.identity(),Collectors.counting()))).entryset().stram().

filter(x->x.getValue()==1).findFist().get().getKey();

Second Hight elements in array:

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

Integer it=Arrays.stram(number).boxed().sorted(Comparator.reverseOarder()).skip(1).findFist().get();

Largest String in geiven String array:

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

String long=Arrays.stram(array).reduce((e1,e2)->e1.lengthe()>e2.length?e1:e2).get()

Spring Boot Actuator:

Management endpoint monitory check of the application:

Health: up or down

Beans: list of beans

Conditions: why they did or did not autoconfiguration

Loggers: logger info

Mapping : list of Request @RequestMapping

How to validate the custom validation and Spring Validation:

Spring Boot uses Hibernate Validator by default, which supports various built-in annotations like @NotNull, @Size, @Min, @Max, etc. Here’s how you use them:

public class UserDto {

@NotBlank(message = "Username is mandatory")

private String username;

@Size(min = 8, max = 20, message = "Password must be between 8 and 20 characters")

private String password;

// getters and setters

}

c. Enable Validation in Controllers:

In your Spring Boot controller, you can validate incoming request data using the @Valid annotation:

@RestController

public class UserController {

@PostMapping("/users")

public String createUser(@Valid @RequestBody UserDto userDto) {

// Process the userDto

return "User created successfully";

}

}

2. Custom Validation:

Create a Custom Annotation:

@Constraint(validatedBy = CustomValidator.class)

@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION\_TYPE, ElementType.PARAMETER })

@Retention(RetentionPolicy.RUNTIME)

public @interface ValidCustom {

String message() default "Invalid custom value";

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

}

Implement the Custom Validator:

public class CustomValidator implements ConstraintValidator<ValidCustom, String> {

@Override

public void initialize(ValidCustom constraintAnnotation) {

}

@Override

public boolean isValid(String value, ConstraintValidatorContext context) {

// Implement your custom validation logic here

return value != null && value.matches("^[A-Za-z0-9]\*$"); // Example: alphanumeric check

}

}

c. Apply the Custom Annotation

public class UserDto {

@ValidCustom(message = "Username must be alphanumeric")

private String username;

// Other fields, getters, and setters

}

Map:

**What is the difference between HashMap and Hashtable?**

* **Answer:**
  + HashMap is not synchronized and allows one null key and multiple null values, while Hashtable is synchronized and does not allow null keys or values.
  + HashMap is generally faster due to the lack of synchronization.

How can you sort a Map in Java?

Map<String, Integer> map = new HashMap<>();

map.put("a", 1);

map.put("c", 3);

map.put("b", 2);

Map<String, Integer> sortedMap = map.entrySet()

.stream()

.sorted(Map.Entry.comparingByKey())

.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,

(e1, e2) -> e1, LinkedHashMap::new));