https://github.com/TEK-Leads/SB-MS-Workspace

SpringBoot & MicroServices

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

Pre-Requisites

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

Core Java

Adv Java (Jdbc, Servlets, Jsp, JSTL)

Hibernate (ORM)

Spring (Core)

SpringBoot

----------

Spring vs SpringBoot

SpringBoot Advantages

IOC Container

Stereotype annotations

Auto Configuration

Starter POM

Standalone application development using SpringBoot

Introduction to Spring MVC

Spring MVC Architecture

Embedded Servers in SpringBoot

Webapp development using SpringBoot

application.properties file

application.yml file

Boot application deployment in third party server

SpringBoot profiles

SpringBoot Actuators

Exception Handling in SpringBoot

Logging in SpringBoot (SLF4j, Log4j, Logback, MDC)

TheamLeaf in SpringBoot

Apache Kafka

JAX-RS (JAVA API FOR XML RESTFUL SERVICE)

From Spring 3.0v we have support to develop rest application using spring.

Spring with Rest

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

SOAP vs REST

Rest Architecture Principles -- Roy Fielding

XML, XSD and JAX-B

JSON, JsonSchema and Jackson

HTTP Methods

HTTP Status Codes & messages

REST API Development

REST Client Development(RestTemplate, WebClient)

Swagger

Asynchronus Communication Services (Reactive Programming)

Exception Handling in REST API (ExceptionMapper)

SOAP UI & POSTMAN

Securying REST API(BasicAuth & OAuth2.0)

SpringData

----------

Introduction to SpringData

Introduction to SpringData Starter POM

DataSource Configuration

CRUDRepository

JpaRepository

Custom Queries execution using @Query annotation

Sorting

Pagination

MicroServices

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

What is Monolithic Architecture

What is Load Balancer (Cluster) - LBR

What is MicroService Architecture

Advantages of MicroServices

Service Registry(Netflix server)

Gateway Service (Zuul proxy server)

MicroServices Intercommunication (FiegnClient)

Microservice deployment in Pivotal Cloud Foundry

MicroService deployment in AWS

Ribbon (Load Balancer)

Circuit Breaker (Hystrix)

Docker (Orchestration) | Kubernetes

Microservice application monitoring(JMETER)

Tools

------

Maven

Log4J

Junit & PowerMock

Jenkins (CI & CD)

Duration: 45-60 days

Timings: 9:00 AM - 10:30 AM (Monday-Monday)

No class notes

Facebook Group : TEK Leads

email id : tekleads4u@gmail.com or ashok@nareshit.com

3 expereience (Boot & MicroService)

Service based company - Accenture, Deloitte, TCS, Infy, TechM..

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

years of experience \* 3 lakhs

3\*3=9 lakhs

Product based company- Oracle, DeShaw, Pramati, JpMorgan etcc

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

years of experience \* 4 lakhs

3 \* 4 = 12 lakhs

Note: DS, Algos, DP, System Design are very imp

(www.geeksforgeeks.com)

Spring

------

It is an opensource application development framework

Pivotal team

Latest version of spring is 5.x (Reactive Programming-webflux)

Spring is mainely using to develop below 3 types of applications

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

Web applications (C 2 B)

Distributed applications (B 2 B)

MicroService applications (B 2 B)

Spring Core

Spring AOP

Spring MVC & Spring Rest

Spring Data

SpringBoot

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

Spring is framework

SpringBoot is an approach to develop spring applications with minimal or less configuration.

Spring Boot Provided Starter Pom (Simplifying maven dependencies)

Spring Boot Provided Http Embedded Servers (Tomcat , Jetty, netty)

SpringBoot works based on Opinionated Configuration

SpringBoot supports Auto Configuration

SpringBoot supports Profiles (env specific configuration)

SpringBoot supports Actuators (Production Ready Features)

SpringBoot = Spring + Embedded Servers - xml or annotation config

Dis-Advantages of SpringBoot

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

Migrating spring project to spring boot will be very difficult

It is recommended for scratch development project

Un-necessary dependencies will be downloaded from starter pom. Manually we need to exclude them.

Creating SpringBoot Application

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

1) Spring Starter Wizard in STS IDE

2) Spring initializer website ( start.spring.io )

3) CLI

4) Maven project

pom.xml

-------

spring-boot-starter-parent (version number) -- 2.2.0

spring-boot-starter (packaging type is jar)- standalone

spring-boot-starter-test --- unit testing (junit)

Note : If we are not using any dependency then it is recommended to exclude that from build path.

Note: Project specific dependencies we should add manually in pom.xml file.

ex : jdbc-driver, lombok, kaptcha etc.....

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

Main class in springboot application

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

package com.example.demo;

@SpringBootApplication

public class SpringBootApp1Application{

public static void main(String... args){

SpringApplication.run (SpringBootApp1Application.class,args);

}

}

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

spring-boot-starter (standalone)

spring-boot-starter-web (web)

spring-boot-starter-webflux (reactive)

18-10-2019

-----------

-> Spring Starter Wizard in STS

-> Create project from start.spring.io website

-> download project zip file

-> import project into IDE

pom.xml

-------

-> spring-boot-starter-parent

-> spring-boot-starter (standalone) - packaging - jar

-> spring-boot-starter-test

main class

----------

package com.nit;

@SpringBootApplication

public class SpringBootApplication{

public static void main(String... args){

SpringApplication.run(Class, args);

}

}

com.nit.service

com.nit.controller

com.nit.dao

com.util.nit

-> Starts StopWatch

-> Which is profile is activated (to load properties)

-> Type of application (web | reactive | default)

-> Starts IOC Container

-> ClassPath Scan (to identify spring beans)

-> Bean Objects creation & dependency injection

-> Calculate time taken to start application and print it

-> Return IOC Container reference

properties file

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

application.properties or application.yml

presentation

web layer ==> (com.ibm.dbs.admin.controller)

business layer ==> (com.ibm.dbs.admin.service)

persistence layer ==> (com.ibm.dbs.admin.repository)

base package ===== > com.ibm.dbs

21-Oct-19

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

Stereotype annotations

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

@Component

@Service

@Repository

@Controller (C 2 B)-- Spring 3.0

@RestController (B 2 B) -- Spring 4.0

@Configuration

@Bean (Method level)

22-Oct-19

---------

Beans Colloboration

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

SI : Injecting dependent object into target object by calling target class exposed setter method.

public class ContactController{

private ContactService contactService;

public void setContactService(Contactservice cs){

this.contactService = cs;

}

//methods

}

<bean id="controller" class="pkg.ContactController">

<property name="contactService" ref="cs"/>

</bean>

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

CI: Injecting dependent bean into target objan ect by calling target bean constructor.

public class ContactController{

private ContactService contactService;

public ContactController(ContactService cs){

this.contactService = cs;

}

//methods

}

<bean id="controller" class="pkg.ContactController">

<constructor-arg name="contactService" ref="cs"/>

</bean>

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

@Autowired

----------

byName

byType

constructor (internally byType mode will be used)

When we use this Annotation, internally spring uses AutowiredAnnotationBeanPostProcessor.

This annotation we can use at below 3 levels

1) field

2) setter method

3) constructor (if we have only one parameterized constructor with beans as parameters @Autowired is optional)

@Controller

public class ContactController{

@Autowired

private ContactService contactService;

}

@Primary :

----------

If we have multiple beans qualified for Autowiring then to choose one bean we can use @Primary annotation.

The bean which contains @Primary will be considered for Autowiring.

@Qualifier :

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

If we want to inject dependent object based on name of the bean then we can use @Qualifier annotation.

Note: We can use @Primary & @Qualifier only in Autowiring in situation.

public String toString(){

return

this.getClass().getName()+"@"+Integer.toHexString(this.hashCode());

}

SpringApplication.run(..)

@SpringBootApplication

How IOC will start in SpringBoot

ComponentScan

Stereotype annotation

Base Package Naming Convention

Bean Colloboration (Autowiring)

24-Oct-2019

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

SpringBoot Web Application Creation

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

spring-boot-starter-web

- Spring MVC

- Restful App

- Embedded Tomcat Container

Note: When we run this SB web application it Starts IOC container by using WebApplicationContext.

It tries to start Tomcat Server with 8080 as port number.

Note: We can change Embedded Server Port Number in

application.properties

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

server.port=9090

application.yml

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

server:

port: 9090

YAML : Yet Another Markup Language

Note: Indent spaces plays very important in yml file

In SpringBoot web application context-path is empty by default.

Note: We can specifiy context-path in properties file or yml file

application.properties

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

server.servlet.context-path=/App

application.yml

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

server:

servlet:

context-path: /App

With Context-Path : http://localhost:7070/App/welcome

Without Context-Path : http://localhost:7070/welcome

Note:

-----

Context Path should start with '/' and should not end with '/'

If we have spring-boot-starter-web then by default we will get

Embedded Tomcat container.

If we don't want to use Embedded Tomcat container we should remove that from build path.

To remove dependency in maven we can use exclusions option

Making Jetty Serve as default embedded server

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

1) Exclude starter-tomcat from spring-boot-starter-web

2) Add Jetty starter dependency in pom.xml

Deploying SpringBoot Webapplication in External Tomcat server

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

1) Specify embedded server scope as 'provided' in pom.xml

2) Add <start-class> in <properties /> section of pom.xml

RestAPI

-------

1) SOAP vs REST

2) Rest Architecture Principles

3) RestController

4) Http Annotations

- @GetMapping

- @PostMapping

- @PutMapping

- @DeleteMapping

5) @RequestBody & @ResponseBody

6) @RequestParam & @PathParam

7) Consumes

8) Produces

9) Content-Type & Accept headers

10) POSTMAN

11) Basic Authentication

12) XML & JAX-B

13) JSON & Jackson

14) Swagger

15) Exception Handling

16) Rest Client Development (RestTemplate & WebClient)

SOAP Webservices

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

Provider : Application which is providing services

Consumer : Application which is accessing services

WSDL : Webservice description language (xml)

SOAP : Simple Object Access Protocol (xml)

SOAP UI : Provider functionality testing tool

Note: Webservices is the first technology which supported for Distributed application development with Intereoperability.

Roy fielding identified some loop holes available in SOAP Based Services. They are below

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

1) Always we should exchange data in soap xml format only

(Real intereoperability is not available)

2) Everybody can't access soap webservices

(Real adoptability is not available)

To avoid problems with SOAP development Roy Fielding Provided Rest Architecture principles

1) Unique Addressbility

2) Uniform Constraint Interfaces

3) Message Oriented Representation

4) Communication Stateless

5) HATEOS

JAX-RS API (java api for xml - Restful services)

- Jersey (Sun)

- Rest Easy (Jboss)

Spring also supporting for Restful services development from Spring 3.0 version onwards.

3.0 =====> @Controller + @ResponseBody

4.0 ===> @RestController

Unique Addressability

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

Every distributed component should bind to unique address.

Note: Class level & method level address mapping.

Uniform Constraint interfaces

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

To achieve easy adoptability we should bind our rest controller method to HTTP methods

GET method ----> @GetMapping

POST method ---> @PostMapping

PUT method ----> @PutMapping

DELETE method ---> @DeleteMapping

Message Oriented Representation

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

Rest api supports for multiple data formats

Ex: xml, json, yml, text, html etc....

Communication Stateless

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

In Rest api communication, every client request will be considered as new request. Nothing will be stored at server.

HATEOS

------

Hypermedia as an engine for application state

Working with JSON

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

JSON - Java Script Object Notation

JSON is platform independent & language independent

JSON is light weight when compared with XML

JSON represents data in key-value format

To work with JSON data in Java applications we can use below apis

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

1) Jackson

2) Gson

Jackson API

-----------

ObjectMapper mapper = new ObjectMapper();

//converting java obj to json data

String personJson = mapper.writeValueAsString(personObj);

//converting json data to java obj

Person p = mapper.readValue(personJson,Person.class);

Gson API

--------

Gson gson = new Gson();

//convert java obj to json

String personJson = gson.toJson(personObj);

//convert json to java object

Person p = gson.fromJson(personJson,Person.class);

RestController with Json Data

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

@RestController ---> Class will become distributed component

@GetMapping --> To bind method to HTTP GET request

GET : If server wants to send data to client then use GET request method (client will send GET request to get the data)

-> GET request will not contain body

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

@PostMapping --> To bind method to HTTP Post request

POST : If server wants to recieve the data to create new record then use POST request method (Client will send data in POST request body).

-> POST request will contain body

Consumes & Produces

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

Consumes represents in which format rest api method can recieve the data

Produces represents in which format rest api method can provide the response

Note: one method can support for multiple consume and produces formats.

GET request with Produces syntax

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

@GetMapping(

value="/getCustomer",

produces={

"application/json"

}

)

public Customer getCustomer(){

//logic

return customerObj;

}

Note: When client sending GET request to above method he should send

Accept header.

syntax : Accept=application/json

Note: Accept header represents client expecting response format from server.

POST Request method with JSON

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

@PostMapping(

value="/addCustomer",

consumes= {

"application/json"

}

)

public String addCustomer(@RequestBody Customer c){

//logic

return "Success";

}

When Client is sending POST request with data in request body, he should send one header called 'Content-Type'

Content-Type header represents in which format client is sending data to server in request body.

Syntax: Content-Type=application/json

Rest API Testing Tools

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

1) SOAP UI

2) POSTMAN

3) Swagger UI

JAX-B API

---------

Java Architecture For XML Binding. This api is used to convert java object to xml data and vice versa.

XML

XSD

One Time Operation

Runtime Operations

- Marshalling

- Un-Marshalling

XML

---

XML stands for extensible markup language

XML provided by w3c org

Initial version of xml is 1.0 and current version of xml is also 1.0

XML is platform independent and language independent

XMl is intereoperable

People will use xml to represent the data and to transfer the data

Sample

------

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

<person id="101">

<first-name>Cedrone</first-name>

<last-name>Charles</last-name>

</person>

Simple Element : The element which contains data

Ex : first-name & last-name

Compound Element : The element which contains child element(s)

Ex : person

Attribute : To store supplement information

Ex : id

Xml is used to store the data

XSD is used to represent structure of the xml

One Time Operation

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

Creating binding classes is called One Time Operation

Note: If we have XSD, then we can create binding classes for that XSD using XJC compiler.

Once we have binding classes then we can perform Runtime Operations

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

1) Marshalling

2) Un-Marshalling

Note: Without having binding classes we can't perform Runtime operations.

Marshalling

-----------

The process of converting java object to xml data

JAXBContext ctxt = JAXBContext.newInstance(Customer.class);

Marshaller marshaller = ctxt.createMarshaller();

marshaller.marshall(customerObj, System.out);

Un-Marshalling

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

The process of converting xml data to java object

JAXBContext ctxt = JAXBContext.newInstance(Customer.class);

UnMarshaller unmarshaller = ctxt.createUnMarshaller ( );

File f = new File("customer.xml");

Customer c = (Customer) unmarshaller.unmarshall(f);

Note: Marshalling & Un-Marshalling we can perform only on binding objects.

Query Paramter

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

->Query Parameters are used in url to send data to server

-> Query parameters will be available in key-value format

Ex: ?id=101

-> Query Parameter will start with question mark (?) symbol

Note: Query parameter always should be at end of the url

-> If we want to send more than one query parameter then we should use & operator

To recieve query parameter we will write below piece of code in JEE

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

String id = request.getParameter("id");

Integer cid = Integer.parseInt(id);

To represent query parameters in Spring Rest Controller we will use

@RequestParam("key") annotation

public Customer getCustomer(@RequestParam("id") Integer cid) {

//logic

return c;

}

Path Parameter

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

-> Path Parameters are used to send data to server in URL directly

-> To recieve PathParameters we can use @PathVariable annotation

@GetMapping(value="/getCustomer/{id}",produces="xml/json")

public Customer getCustomer(@PathVariable("id") Integer cid){

//logic

return c;

}

Http Status Codes

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

200-299 : OK (success)

300-399 : Redirect

400-499 : Client problem

500-599 : Internal Server Problem

Exception Handling in Rest API

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

We can handle exceptions in 2 ways

1) Local Exception Handling (Controller specific)

2) Global Exception Handling (Entire application)

@ExceptionHandler(CustomerNotFoundException.class)

public ResponseEntity<String> handleCNFException(){

ResponseEntity<String> re =

new ResponseEntity("Invalid ID",400);

}

Difference between WSDL and WADL

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

-> WSDL stands for Webservices description language

-> WSDL contains details of provider

-> WSDL is mandatory for SOAP based webservices

-> Using WSDL consumer will understand provider

-> WADL stands for Webapplication Description Language

-> WADL provides webapplication documentation

-> Rest API also can be described by using WADL

-> WADL demand is very less in industry because of Swagger

-> Swagger is used to generate documentation for Rest API

-> Swagger UI can be used to test Rest api

What is Swagger

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

Swagger is a custom documentation plugin which is used to generate documentation for web application.

In Industry people are using Swagger mainely to describe only

Rest API.

Adding to Swagger to Project

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

1) Add swagger dependencies to project pom.xml

1) swagger documentation

2) swagger ui

2) Create SwaggerConfiguration class using @Configuration annotation

and write a method to create Docket object using @Bean annotation

and enable swagger using @EnableSwagger2 annotation.

3) Use Swagger annotations to provide more details about Rest API

4) Run SpringBoot application and access documentation & UI using below URLs.

http://localhost:9090/swagger-ui.html --> for UI screen

http://localhost:9090/v2/api-docs --> for documentation in json

Below are few annotations of Swagger

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

@Api

@ApiOperation

@ApiParam

@ApiModel

@ApiModelProperty

@ApiResponses

@ApiResponse

Spring Data

-----------

Spring Data is used to develop persistent logic for our application.

This is mainely used to to perform only CURD operations.

As part of Spring Data, we have below 2 Repositories

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

1) CrudRepository (curd operations)

2) JpaRepository (curd operation + pagination + sorting)

Database Setup

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

We need to install below 2 softwares to work with DB operations

1) DB Server (Oracle, MySql, SqlServer, DB2 etc)

2) DB Client (SQL Developer, Toad)

-> First install Oracle Database 10g

-> Download SQL developer from Oracle website.

-> Create Connection in SQL developer using DB details.

Note: To Connect with Oracle Database we need Oracle Database Driver (OracleDriver). Oracle Driver will be available in ojdbc6.jar file.

As Oracle is Proprietary software we can't get Maven Dependency From mvnrepository.com.

To use ojdbc6.jar we have below 2 options

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

1) Add ojdbc6.jar file to project build path

2) Install ojdbc6.jar file to maven local repository and add local repository dependency in project pom.xml

Installing Third Party Jar to Maven Local Repository

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

1) Install Maven

2) Add MAVEN\_HOME to environment variables

3) Use maven install command to install jar to local repo.

> mvn install:install-file -Dfile=ojdbc6.jar

-DgroupId=com.oracle

-DartifactId=ojdbc6

-Dversion=2.2.2

-Dpackaging=jar

Once above command is execute, ojdbc6.jar is installed to maven local repository. We can use below dependency in pom.xml to get ojdbc6.jar file to project build path.

<dependency>

<groupId>com.oracle</groupId>

<artifactId>ojdbc6</artifactId>

<version>2.2.2</version>

</dependency>

Note: This dependency applicable only for the system where above command is executed.

Working with SpringData in Project

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

1) Add Spring Data Jpa dependency in pom.xml file

2) Add JDBC driver dependency in pom.xml

3) Configure Data Source Properties in application.properties file or application.yml file

4) Create Entity Class (mapping with database table)

5) Create Repository interface by extending from SpringData Repository

6) Inject Repository interface into service class and call methods to perform curd operations.

Methods available in CrudRepository

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

T save(T entity) --- polymorphic method (save and update)

Optional findById(Serializable id)

Iterable<T> findAll( )

Iterable<T> findAllById(Iterable<Serializable> ids)

delete(T entity)

deleteById(Serializable id)

deleteAll( )

deleteAllById(Iterable<Serializable> ids)

long count()

existById(Serializable id)

Custom Queries in Spring Data

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

We can execute Custom queries also in Spring data

Custom Queries are fail-fast

Custom queries will be compiled at the time of application startup

To write the custom queries we will use @Query annotation

Custom Queries Examples

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

@Query(value = "select customerEmail from CustomerEntity")

public List<String> findAllEmails();

@Query(value = "from CustomerEntity where customerEmail=:email")

public CustomerEntity findByCustomerEmail(String email);

Application environments

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

Every applicaiton contains below environments to test all functionalities properly.

Local : Development will happen

DEV : Developer Integration Testing

QA : System Integration Testing

UAT : User acceptance Testing

Pilot/Staging : Similar to prod to test with real data

Production : Go live

From environment to environment few properties values are going to change like DB, Smtp, logging, webservices endponints etc....

In SpringBoot application we will configure below properties in application.properties file or application.yml file

1) DB Config Properties

2) ORM properties

3) SMTP properties

4) Webservice Endpoint URLs

5) Application messages

At the time of deployment changing environment specific properites everytime is difficult and time waste process.

To avoid this problem we will maintain envrinoment specific configuration files using Profiles concept.

SpringBoot profiles

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

Profiles are used to main environment specific configuration.

we will create profiles like below

application-dev.yml --> DEV profile file

application-qa.yml --> QA profile file

application-uat.yml --> UAT profile file

application.yml ------ base file

Note : in base yml we need to activate profile like below

spring:

profiles:

active: qa

Note: As per above configuration application application will load

qa yml file.

Creating User in Oracle Database

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

1) Connect to DB as a SysDba in SQL command line

Open sql command line and execute below command

Command : connect as sysdba;

Username : dba

password : abc

2) Execute below queries

create user <uname> identified by <pwd>

grant connect, resource to <uname>

3) Commit

Spring Boot Actuators

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

To provide production ready features

1) Health check

2) Thread Dump

3) Heap Dump

4) Config Props

5) Classes

6) Http Trace

7) Stop the application

/actuator/shutdown ---> This is by default disabled

This we need to enable

This endpoint is binded to HTTP Post request

Unit Testing

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

JUnit

JUnit annotations (@Test) --- @SpringBootTest

Assertions --- to verify results

Mocking (EasyMock, PowerMock, Mockito, WireMock etc....)

Mockito

Code Coverage report using Jacocco

Note : If we create SpringBoot project with 2.x Boot version, then Junit 5.x and Mockito will come to build path.

@SpringBootTest :- This is used to represent class as Unit testing class.

@Test : To represent method as unit testing method.

Assertions :- Predefined class in Junit. Provided many static methods to verify test results.

@InjectMocks : This annotation used to represent target class object with dependents as mock objs.

@Mock : It is used to create Mock object and inject into @InjectMocks object.

Demo d = new Demo(); //9797979 --- entity

Demo d1 = new Demo(); //9968686 ---- null

Mockito.when(customerRepo.save(entity))).thenReturn(entity);

Mockito.when(customerRepo.save(Mockito.any(CustomerEntity.class))).thenReturn(entity);

@SpringBootTest

@Test

@InjectMocks

@Mock

Code Coverage

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

To Code Coverage report we can use Jacoco plugin in pom.xml file

Logging

-------

Maintaing application execution status

To track application exceptions in higher environments

It is very important for every application to generate log messages.

to generate log messages we need to configure logging for our project.

Apache provided log4J framework to generate log message for java applications.

Configure log4j using properties file

configure log4j using xml file

Configure log4j using programmatic approach

In log4J mainly we have 3 components

1) Logger : It is providing methods to generate log msg

2) Layout : It represents pattern of log msg

3) Appender : It represents destination

Log LEVELS

----------

TRACE < DEBUG < INFO < WARN < ERROR < FATAL

DEBUG < INFO < WARN < ERROR

public void m1(){

//method started ---- DEBUG

//exception occured -- ERROR

//method ended ---DEBUG

//method completed successfully -- INFO

}

SpringBoot supports for below logging techniques

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

1) Log4J

2) Log4J2

3) Logback

In Spring below are the default configurations

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

rootLogger Level is : INFO

Layout : Pattern Layout

Appender : ConsoleAppender

Logger : Providing methods

Layout : PatternLayout

Appender : ConsoleAppender, FileAppender etc...

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

logging.pattern.console= null

logging.file=file-path

logging.pattern.file=

logging.level.org.springframework= ERROR

logging.level.com.nit.service= DEBUG

functional team

dev team

testing team

db team

devops team

splunk team

SpringBoot application

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

RestController --- Distributed component

- POST req method

- GET request methods

- XML and JSON data input & output format

JAX-B and Jackson APIs

Service class -- Business logic

Spring Data Repository interface -- Persistence logic

Junit with Mockito -- Unit testing

Jacocoo -- Code coverage

Logging ---to generate log messages

Exception Handling -- Local & Global exception handling

Profiles -- environment specific config

Actuators -- Production ready features

Swagger -- To generate Rest Documentation

Embedded Server

External Server

Rest Client Development

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

The application which is going to access business services from other application is called as Client application.

In order to access Rest API in our project, we should develop Rest Client.

We can develop Rest client in below ways

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

1) java.net client

2) Apache Http Client

3) Jersey Client

4) RestEasy Client

5) RestTemplate --- (Spring) Deprecated

6) WebClient ---(Spring) (SpringBoot 2.x feature)

7) Fiegn Client (MicroService approach)

GET request : http://localhost:9091/getAllEmails

Resource : @RestController --> Distributed Component

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

JAX-RS

Spring Framework

CRNK

Client : RestClient

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

RestTemplate

WebClient (Sync & Async)

Mono

Flux

Flux Delay

Stream

URL : http://localhost:9090/getByEmail

Request Type : GET

Parameter : ?email=<email-id>

Reponse Content-Type : application/xml & application/json

Response data : Customer Data (id, name, email)

HttpHeaders headers = new HttpHeaders();

headers.add("Accept", "application/xml");

HttpEntity<Customer> entity = new HttpEntity<Customer>(body, headers);

restTemplate.postForEntity(url,entity,Customer.class);

restTemplate.postForEntity(url,customer,String.class);

RestTemplate

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

It is a predefined class available in Spring Web MVC module

It introduced in Spring 3.0 v

Using this RestTemplate we can access Rest API

Using This RestTemplate we can send Http request to access Rest API.

RestTemplate supports Synchronus way to access Rest API

In order to use RestTemplate we need spring-boot-starter-web dependency

WebClient

----------

From Spring 5.0v onwadrs, it supporting for Reactive Programming.

As part Reactive programming, Spring introduced WebClient class as a alternate for RestTemplate.

WebClient class is used to send requests Asynchronsly

Using WebClient we can make both sync and async calls also

To use WebClient in project, we should use spring-boot-starter-webflux

Mono :- 0---1 : bodyToMono(T)

Flux :- 0----N : bodyToFlux(T)

Customer c = restTemplate.getForEntity(url,Customer.class);

//logic

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

webClient.get()

.uri(GET\_CUSTOMER\_BY\_EMAIL\_URL)

.retrieve()

.bodyToMono(Customer.class)

.subscribe()

//logic

@RestController

public class CustomerEventRestController{

@GetMapping(

value="/getEvents",

produces=MediaType.TEXT\_EVENT\_STREAM

)

public Flux<CustomerEvent> getCustomerEvents(){

CustomerEvent ce = new CustomerEvent("ashok",new Date());

Flux<Interval> interval =

Flux.interval(Duration.ofSeconds(3));

Flux<CE> eventStream =

Flux.stream(Stream.generate( () -> ce ));

return Flux.zip(interval,eventStream)

.map(Tuple2::getT2);

}

}

RestTemplate (3.0v) vs WebClient (5.0v)

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

RestTempalte --- sync approach

WebClient --- sync & async approches

block( ) --- sync

sybscribe(Consumer c) --- async

WebClient.Builder webClientBuilder = WebClient.builder( );

WebClient webClient = webClientBuilder.build();

webClient.get( )

.uri("")

.retrieve( )

.bodyToMono(T cls)

.block( );

//logic

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

webClient.get()

.uri("")

.retrieve()

.bodyToMono(T cls)

.subscribe(System.out::println);

//logic

Mono vs Flux

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

Mono ---> 0----1

Flux ---> 0 --- N

Rest Endpont URL : http://localhost:1111/getEvent --- Mono

Rest Endpoint URL : http://localhost:1111/getEvents --- Flux

WebClient

sync request using block ( )

async request using subscribe ( )

Mono ---> 0 --- 1

Flux ---> 0 --- N

webClient.get( )

.uri( )

.addHeader(key,value)

.retrieve( )

.bodyToMono(T.class)

.subscribe(System.out::println);

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

webClient.post ( )

.uri("")

.syncBody(customerObj)

.addHeader("","")

.retrieve( )

.botyToMono(String.class)

.block ()

Reading Properties file & Yml file

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

1) Environment

2) @Value

3) @ConfigurationProperties

CommandLineRunner (CLR)

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

CommandLineRunner is a functional interface in SpringBoot

It is having run(String... args) method

This is used to execute logic only one time when application starts

We can create more than one command line runner bean also

To set order for CLR execution we can use @Order annotation

Note: CLR will execute once all bean objects created then CLR will execute.

-> Data clean up activity we can do using CLR

-> We can access application cmd args using CLR

-> We can check bean classes loaded using CLR

ApplicationRunner (ALR)

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

public void run(String... args)

public void run(ApplicationArguments args)

IT helpdesk ---> We should raise a request

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

Working with H2 Database

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

1) Add h2 dependency in pom.xml file

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

2) Create below files with required queries in src/main/resources folder

schema.sql --> Table creation sql

data.sql ---> Insert sample data sql

3) Configure DataSource in application.yml file

spring:

datasource:

username: sa

password: null

url: jdbc:h2:mem:testdb

driver-class-name: org.h2.Driver

4) To open In memory database use below URL

http://localhost:<PORT-NO>/h2-console

https://github.com/TEK-Leads/SB-MS-Workspace

MicroServices

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

Monolithic Architecture

Challenges of Monolithic

Load Balancer (Cluster)

Microservices Introduction

MicroServices Advantages

Microservices challenges

MicroServices Architecture

MicroServices Development (Eureka & zuul)

Introduction to cloud(PCF, AWS and Azure)

MicroServices Deployment using PCF

AutoScaling

Interservice Communication (FeignClient)

Docker

Security

Ribbon

Hystrix

Kafka

Use Case

--------

Develop a webapplication to perform Currency Conversion based Currency Exchange Cost

Ex: Dollar to Rupee | Euro to Rupee vice versa

This application should support for both C 2 B and B 2 B operations.

Components

----------

UI (presentation)-- TheamLeaf

Controller (To handle C 2 B request)

RestController (To handle B 2 B request)

Service (To execute core Business Logic)

Repository (To execute Persistent Logic)

In Memory DB (To store Currency Exchange Cost)

Application Deployment

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

If application is deployed to single server we have both advantages and dis-advantages.

Advantages

----------

-> Infra Cost is less

-> No need to recruite multiple people to mange deployment process

Dis-Advantages

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

-> If requests are more, response time is going to increase

-> There is a chance server may crash

-> If server is crashed, application will be then revenue will be impacted.

-> We can't go for prod deployment without bringing down the application.

-> Prodiving Project Details to new team member is difficult

Load Balancer

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

Instead of deploying our application to single server, we will deploy our application to mutliple servers using Load Balancer.

Load Balancer is used to distribute client requests to all the nodes which are connected to cluster.

Advantages

----------

-> Burden on server will be reduced bcz multiple servers will process

-> Response time will be decreased

-> Less Chances for server crash

-> We can scale our application for all scenarios (Big Day sales)

-> We can do production deployments without application down time.

Dis-Advantages

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

-> Cluster is a costly solution

-> Lot of money we shud spend to create Infrastructure

-> We need team of admins to setup, maintain and monitor Cluster

-> Resource cost will be high

To resolve problems of Monolithic We are going to use MicroServices

Micro Services

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

-> Micro Service is not a Framework

-> Micro Service is not a Technology

-> Micro Service is not an API

-> MicroService is an architectural Design Pattern.

-> The main aim of MicroService is to resolve the problems of Monolithic Architecture.

-> MicroService is not specific to any programming language. It is an universal Design Pattern to develop applications.

MicroServices Advantages

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

1) New Technologies

2) Parallel Development

3) Fast Releases

4) Dynamic Scaling ( Horizontal Scaling & Vertical Scaling )

5) Explaing Project to new Team members is easy (Single Business Unit)

MicroServices Dis-Advantages

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

1) Bounded Context (Designing)

2) Lot of Configuration

3) Visibility

4) Pack of cards problem

MicroServices Architecture

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

MicroServices architecture will talk about how many components will involved and how those components will be connected.

Note: All projects may not follow same architecture. MicroServices architecture will change from project to project.

Currency-Exchange-Service

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

This MicroService is responsible to provide Currency Exchange Cost.

Who ever want to do Currency Conversion they should get currency exchagne cost using this micro service.

Note: In this Currency Exchange Service, current exchange cost will be available in in-memory database.

Components

----------

CurrencyExchangeRestController.java

CurrencyExchangeService.java

CurrencyExchangeCostBean.java

CurrencyExchangeEntity.java

CurrencyExchangeRepository.java

schema.sql

data.sql

Currency-Conversion-Api

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

This api is responsible to perform Currency Conversion according to given values.

From :

To :

Quantity :

To perform Currency Conversion it needs Currency Exchange Cost.

Currency Exchange Cost will be provided by Currency-Exchange service

Note:

-----

Currency-conversion-api will communicate with currency-exchange api

Components

----------

CurrencyConversionRestController.java

CurrencyConversionService.java

CurrencyExchangeCostBean.java--- CE reponse bean

CurrencyBean.java -- CC response bean

http://localhost:2222/convert/from/USD/to/INR/quantity/300

Cloud Platform

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

AWS (Amazon Webservices) -- Free Trial (5 GB)

PCF (Pivotal Cloud Foundry) -- $87

Microsoft Azure--

PCF CLI Commands

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

cf login

cf target

cf push ---> It requires appname

cf push app-name --> Unable to recognize build pack

Ex : cf push SBRestApi -p target/jarfilename.jar

Our API URL : https://sbrestapi.cfapps.io/

https://tekleads-rest-api.cfapps.io/

https://sbrestapi.cfapps.io/welcome/Smith

cf map-route sbrestapi cfapps.io --hostname tekleads-rest-api

https://sbrestapi.cfapps.io/welcome/Ashok

https://tekleads-rest-api.cfapps.io/welcome/Ashok

cf login

cf target

cf push app-name -p abosolute-path-of-jar-file

cf apps

cf routes

cf -map-route

cf stop app-name

cf start app-name

cf delete app-name

build pack

Scaling

cf login

cf target

cf apps

cf routes

cf push APP\_NAME -p JAR\_FILE

cf stop APP\_NAME

cf start APP\_NAME

cf restart APP\_NAME

cf map-route APP\_NAME DOMAIN\_NAME -hostname tekleads-api

cf delete APP\_NAME

cf logs APP\_NAME

cf push ---> it will search for manifest.yml

https://tekleads-ce-api.cfapps.io/

/getCurrencyExchangeCost/from/USD/to/INR

http://tekleads-cc-api.cfapps.io/convert/from/USD/to/INR/quantity/10

IP : http://tekleads-cc-api.cfapps.io/

Path : convert/from/{form}/to/{form}/quantity/{quantity}

Request Type : GET

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

Response Type

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

{

"currencyFrom":"USD",

"currencyTo":"INR",

"totalCurrencyAmt":750.0

}

Currency Converion UI Application

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

This web application is used to provide Currency Conversion Functionality for end users.

Note: This UI application will do currency conversion with the help of CC and CE microservices which are running in PCF.

Components

----------

Create Web application using SpringBoot

Controller class (To display Form and To Handle Form submission)

Form Binding Class( To bind form data to java object)

Service class (To handle business logic - Restcall)

Response Bean Class (To store CC api response data)

View File (jsp - to display form)

CC\_REST\_ENDPONT =

"http://tekleads-cc-api.cfapps.io/convert/from/{from}/to/{to}/quantity/{quantity}";

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

Builder builder = Webclient.builder();

WebClient webClient = builder.build( );

CurrencyResponseBean responseBean =

webClient.get( )

.uri(CC\_REST\_ENDPOINT,1,2,3)

.retrieve( )

.bodyToMono(CurrencyResponseBean.class)

.block( );

Spring MVC Flow

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

DispatcherServlet : FrontController

HandlerMapper : mapping between request and request-handler

Controller : Request-Handler (@Controller)

ModelAndView : Model represents data, view represnets UI page

ViewResolver : To locate view files, to understand view technology

View : To render Model Data on View File

Service Registry For MicroServices

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

Service Registry is used to maintain project related microservices details.

We can use Netflix-Eureka Server as a Service Registry.

Working with Service Registry

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

1) Create SpringBoot Web application

2) Add Eureka-Server dependency in pom.xml file

3) Use @EnableEurekaServer at SpringBoot start class

4) ServiceRegistry project should not register with Eureka

eureka.client.register-with-eureka=false

5) After Running this application, eureka dashboard using below url

localhost:<port-num>

Creating MicroService which acts as Eureka Client

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

1) Create SpringBoot Web application

2) Add Eureka-Client dependency in pom.xml file

3) Use @EnableDiscoveryClient at SpringBoot main class

4) Create Required RestControllers, Services etc...

5) Change port number in application.yml file

6) Add Application Name application.yml file (It is vvvvimp)

7) Run application, then check Eureka Dashboard

Note: If Eureka Service Registry Project running in diff machine, then we should add below property in client application yml file.

eureka:

client:

service-url:

defaultZone: http://IP:PORT-NUM/eureka

Creating API Gateway

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

1) Create SpringBoot Web application

2) Add Eureka Client Dependency in pom.xml

3) Add Zuul dependency in pom.xml

4) Use @EnableDiscoveryClient at SpringBoot start class

5) Change port number and configure Application name in yml file

6) Configure zuul routings in application.yml file

7) Run this application and check Eureka dashboard.

API-1 URL : http://IP:PORT/welcome/getWelcomeMsg

API-2 URL : http://IP:PORT/greet/getGreetMsg

/welcome/\*\* -----> WELCOME-SERVICE

/greet/\*\* -----> GREET-SERVICE

01-Eureka : http://IP:8761/

02-Welcome-Service : http://192.168.100.164:2222/

03-Greet-Service : http://192.168.100.164:3333/

04-API-Gateway : http://192.168.100.164:4444/

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

http://192.168.100.164:4444/api/welcome/getWelcomeMsg/Ashok

http://192.168.100.164:4444/api/greet/getGreetMsg/Ashok

Service registry

How to register client application with Eureka Server

Port : 8761

@EnableFeignClients

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

@FeignClient(name="GREET-SERVICE")

public interface GreetApiClientProxy{

@RequestMapping("/getGreetMsg/{name}")

public String invokeGreetApi(@PathVariable("name") String name);

}

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

https://github.com/TEK-Leads/SB-MS-Workspace

Singleton

Factory

Builder

Proxy

Observer & Observable

Dao

MVC

FrontController

Strategy

Cache

MicroService (Single Responsibility)

CircuitBreaker (Hystrix)

Ribbon

Admin UI

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

Class Loaders

Docker

Apache Kafka

Payment Gateway

SonarQube

Security

Technical FLow of modules

UML Diagrams

SOLID OOPS

Hikari

PM Round Questions

Daily Activities in Ofc

11-Dec-2019

-----------

1) Microservice down

2) MicroService performance is slow (JMETER)-- NewRelic

3) Exception Occured In MicroService due to ThirdParty api is down

(Circuit Breaker)-- Hystrix

Working with CircuitBreaker in MS

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

1) Create SpringBoot web application

2) Add Hystrix and Hystrix Dashboard dependencies in pom.xml file

3) Configure @EnableCircuitBreaker annotation at start class

4) For which method we should apply fallback mechanism, at that method use @HystrixCommand annotation

@HystrixCommand(fallBackMethod="method-name")

5) Add Springboot-starter-actuator dependency in pom.xml file

6) Configure hystrix.stream endpoint in application.yml file

7) Configure @EnableHystrixDashboard annotation at start class

8) After deploying the application we can access Hystrix dashboard using below url

http://localhost:5555/hystrix

9) In hystrix dashboard pass stream url to monitor traffic

http://localhost:5555/actuator/hystrix.stream

Load balancing

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

The process of distributing load to multiple servers

Load Balancing is used to reduce burden on server

Load Balancing we can do in 2 ways

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

1) Server Side Load Balancing

Load Balancing Algorithms

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

1) Round Robbin

2) IP Hashing

3) Sticky Session etc...

2) Client Side Load Balancing

pom : ribbon-client

@EnableRibbonClients

@FeignClient(name="CURRENCY-EXCHANGE-SERVICE)

@RibbonClient(name="CURRENCY-EXCHANGE-SERVICE)

public interface CECLient{

}

CE Response

-----------

from : USD

to : INR

cost " 75.0

port : 1111

String port = environment.getProperty("local.server.port");

-Dserver.port=8888

http://localhost:8888/getData

http://localhost:9999/getData

http://localhost:8888/convert/from/USD/to/INR/quantity/200

http://localhost:8888/convert/from/USD/to/INR/quantity/100

http://localhost:8888/convert/from/USD/to/INR/quantity/50

http://localhost:8888/convert/from/USD/to/INR/quantity/20

Spring Boot Admin

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

Actuators

----------

To provide production ready features of our application we are using Actuators.

Actuator Endpoints

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

health

info

beans

mappings

threadump

heapdump

httptrace

shutdown (POST) etc.......

SpringBoot Admin

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

Springboot admin provides, one comman place to maintain and monitor all spring boot based applications.

Creating Spring Boot Admin Server Project

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

1) Add spring-boot-admin-server-starter dependency in pom.xml file

2) Specifiy @EnableAdminServer annotation at SpringBoot start class

Creating Boot Admin Client Project

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

1) Add below dependencies in pom.xml file

a) spring-boot-admin-client-starter

b) spring-boot-starter-actuator

2) Configure below details in application.properties or yml file

a) Configure port number

b) Configure application name

c) Register Client with Admin using URL

d) Expose required actuator endpoints

https://github.com/TEK-Leads/SB-MS-Workspace

(MicroServices-Workspace.rar)