

#####

Microservices

#####

=====

What is Monolith Architecture

=====

-> If we develop all the functionalities in single project then it is called as Monolith architecture based application

-> We will package our application as a jar/war to deploy into server

-> As monolith application contains all functionalities, it will become fat jar/war

Advantages

1) Simple to develop

2) Everything is available at once place

3) Configuration required only once

Dis-Advantages

1) Difficult to maintain

2) Dependencies among the functionalites

3) Single Point Of Failure

4) Entire Project Re-Deployment

***** To overcome the problems of Monolithic, Microservices architecture came into market*****

-> Microservices is not a programming language

-> Microservices is not a framework

-> Microservices is not an API

Microservices is an architectural design pattern

-> Microservices suggesting to develop application functionalities with loosely coupling

-> In Microservices architecture we don't develop all the functionalities in single project. We will divide project functionalities into several REST APIs.

*****Note: One REST API is called as one Microservice*****

-> Microservices architecture based project means collection of REST APIs.

-> Microservices is not related to only java. Any programming language specific project can use Microservices Architecture.

Advantages

- 1) Loosely Coupling
- 2) Easy To maintain
- 3) Faster Development
- 4) Quick Deployment
- 5) Faster Releases
- 6) Less Downtime
- 7) Technology Independence (We can develop backend apis with multiple technologies)

Dis-Advantages

- 1) Bounded Context (Deciding no.of services to be created)
- 2) Lot of configurations
- 3) Visibility
- 4) Pack of cards

Microservices Architecture

-> We don't have any fixed architecture for Microservices

-> People are customizing microservices architecture according to their Project requirement

-> Most of the projects will use below components in Microservices Architecture

- 1) Service Registry (Eureka Server)
- 2) Services (REST APIs)

3) Interservice Communication (FeginClient)

4) API Gateway

5) Admin Server

6) Zipkin

=====

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

=====

Backend 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)

Note: We will register every service with Service Registry

=====

API Gateway

=====

- > API Gateway is used to manage our backend apis of the project
- > API Gateway acts as mediator between end users and backend apis
- > API Gateway can contain filter logic to decide request processing (Authentication)
- > API Gateway will contain Routing logic (which request should go to which REST API)
- > API Gateway also will be registered with Service Registry
- > Spring Cloud Gateway we can use as API Gateway

=====

Admin Server

=====

- > Admin Server is used to manage all backend apis actuator endpoints at one place
- > Our backend apis will be registered with Admin Server

-> Admin Server will provide User interface to monitor apis actuator endpoints

=====
Zipin Server
=====

-> Zipkin Server is used for Distributed Tracing
-> Using this Zipkin, we can monitor which API is taking more time to process our request.

=====
Mini Project Implementation using Microservices Architecture
=====

STOCK-PRICE-API : It will maintain companies stock price details in db table

-> Input : Company Name
-> Output : Company Stock Price Value

Sample Data

TCS-3000
HCL-1500
AXIS-1200

STOCK-PRICE-CALC-API : It is used to calculate total stocks cost based on company name and quantity

-> Input : Company Name & Quantity
-> Output : Total Stocks Cost

Note: To calculate total stocks cost, STOCK-PRICE-CALC-API should get company stock price from STOCK-PRICE-API (INTER SERVICE COMMUNICATION)

=====
Steps to develop Service Registry Application (Eureka Server)
=====

1) Create Service Registry application with below dependency

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

2) Configure @EnableEurekaServer annotation in boot start class

3) Configure below properties in application.yml file

server:
port: 8761

eureka:
client:
register-with-eureka: false

Note: If Service-Registry project port is 8761 then clients can discover service-registry and will register automatically with service-registry. If service-registry project running on any other port number then we have to register clients with service-registry manually.

4) Once application started we can access Eureka Dashboard using below URL

URL : <http://localhost:8761/>

Steps to develop stock-price-api (Eureka-Client-1)

+++++

1) Create Spring Boot application with below dependencies

- eureka-discovery-client
- starter-web
- starter-datajpa
- h2
- devtools

2) Configure @EnableDiscoveryClient annotation at start class

3) Configure below properties in application.yml file

server-port
h2-datasource-properties
application-name

```
-----application.yml-----  
server:  
  port: 1111  
spring:  
  application:  
    name: STOCK-PRICE-API  
  datasource:  
    username: sa  
    password: sa  
    url: jdbc:h2:mem:testdb  
    driver-class-name: org.h2.Driver  
  jpa:  
    defer-datasource-initialization: true
```

4) Create Entity class & Repository interface for COMPANY STOCK PRICE DETAILS

5) Create "data.sql" file under src/main/resource folder with insert queries to load data into db table like below

```
INSERT INTO STOCK_PRICE_DTLS (STOCK_ID, COMPANY_NAME, COMPANY_PRICE) VALUES  
(101, 'TCS', 3000.00);  
INSERT INTO STOCK_PRICE_DTLS (STOCK_ID, COMPANY_NAME, COMPANY_PRICE) VALUES  
(102, 'HCL', 1500.00);  
INSERT INTO STOCK_PRICE_DTLS (STOCK_ID, COMPANY_NAME, COMPANY_PRICE) VALUES  
(103, 'HDFC', 4500.00);
```

```
INSERT INTO STOCK_PRICE_DTLS (STOCK_ID, COMPANY_NAME, COMPANY_PRICE) VALUES  
(104, 'SBI', 450.00);
```

6) Create RestController to handle request & response

Input : CompanyName (Path Parameter)

Output : Stock Price Details

7) Run the application and check in Eureka Dashboard (It should display in eureka dashboard)

Steps To Develop STOCK-PRICE-CALC-API (EUREKA-CLIENT-2)

+++++

1) Create Spring Boot application with below dependencies

- web-starter
- devtools
- eureka-discovery-client
- feign-client

2) Configure @EnableDiscoveryClient annotation at boot start class

3) Configure Server Port & application name in application.yml file

4) Create Rest Controller with required method

Input : Company Name & Quantity (Path Parameters)

Output : Total Cost

Note: In Rest Controller we should have logic to access STOCK-PRICE-API

@RestController

public class StockCalcRestController {

@GetMapping("/calc/{cname}/{qty}")

public ResponseEntity<String> calculate(@PathVariable String cname, @PathVariable Integer qty) {

String url = "http://localhost:1111/price/{cname}";

RestTemplate rt = new RestTemplate();

ResponseEntity<StockPrice> resEntity = rt.getForEntity(url, StockPrice.class, cname);

StockPrice body = resEntity.getBody();

Double companyPrice = body.getCompanyPrice();

Double totalCost = companyPrice * qty;

String msg = "Total Cost : " + totalCost;

return new ResponseEntity<>(msg, HttpStatus.OK);

}

}

-
- > In the above logic we have hard coded STOCK-PRICE-API url
 - > If STOCK-PRICE-API url changed then calc-api logic should be changed
 - > If STOCK-PRICE-API is running in Multiple instances for load balancing our calc-api should access all the instances in round-robin methodology
 - > To overcome these problems we can use Interservice Communication using 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

```
@FeignClient(name = "STOCK-PRICE-API")
public interface StockPriceClient {

    @GetMapping("/price/{cname}")
    public StockPrice invokeStockPrice(@PathVariable String cname);

}
```

Note : Write @EnableFeignClients annotation at boot start class

```
@RestController
public class StockCalcRestController {

    @Autowired
    private StockPriceClient priceClient;

    @GetMapping("/calc/{cname}/{qty}")
    public ResponseEntity<String> calculate(@PathVariable String cname, @PathVariable Integer
qty) {

        StockPrice stockPrice = priceClient.invokeStockPrice(cname);
        Double companyPrice = stockPrice.getCompanyPrice();

        Double totalCost = companyPrice * qty;

        String msg = "Total Cost : " + totalCost;

        return new ResponseEntity<>(msg, HttpStatus.OK);
    }
}
```

Note: Run price-api with multiple instances using Run Configuration Option

Note: Configure Port Number as VM Argument in Run Configuration

-Dserver.port=port-number

```
#####  
API Gateway  
#####
```

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

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

-> In Api Gateway we will write mainly 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) 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-starter
-> 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: stock-price-api
uri: lb://STOCK-PRICE-API
predicates:
- Path=/price/{companyName}
- id: stock-calc-api
uri: lb://STOCK-CALC-API
predicates:
- Path=/calc/{companyName}/{qty}
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);
}
}

-----

@Component
public class MyPostFilter implements GlobalFilter {

    final Logger logger = LoggerFactory.getLogger(MyPostFilter.class);

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

        return chain.filter(exchange).then(Mono.fromRunnable(() -> {
            logger.info("Global Post-filter executed...");
        }));
    }

}

-----

```

- > 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, monitoring can be implemented using Filters

```

=====
Spring Boot Admin Server and Admin Client
=====

```

- > Admin server will provide user interface to monitor and manage all the apis actuator endpoints
- > The REST APIs of our application should register with admin server (It is called as Admin client)

Note: Using this approach we can monitor all the apis at one place

Working with Admin-Server

- 1) Create Boot application with admin-server dependency (select it while creating the project)
- 2) Configure @EnableAdminServer annotation at start class
- 3) Run the boot application

4) Access application URL in browser (We can see Admin Server UI)

Working with Admin-Client

1) Create Spring Boot application with below dependencies

- a) starter-web
- b) starter-actuator
- c) admin-starter-client
- d) devtools

2) Configure below properties in application.yml file

- a) server-port
- b) application-name
- c) enable-actuator-endpoints
- d) configure admin serve URL to reiger

```
server:
port: 1111
spring:
application:
name: CLIENT-ONE
boot:
admin:
client:
url: http://localhost:8080/
management:
endpoints:
web:
exposure:
include: '*'
```

3) Create Rest Controller with required methods

4) Run the appplication (It will register in Admin Server)

```
#####
Cloud Config Server
#####
```

-> It is used to seperate our application and application config properties

Ex: SMTP props, DB props, App msgs etc...

-> We can externalize configuration properties by using Config Server

-> We will maintain application configuration yml files in git hub repo.

-> Config Server application will connect with Git Hub repo to load all ymls

-> Our APIs (Microservices) will connect with config server to load yml data based on application name.

Note: app name and yml file name should be same.

Note: If we use config server then we no need to re-package our application when we make some changes in properties.

```
=====
Config Server App
=====
```

1) Create Git Repository and keep ymls files required for projects

Note: We should keep file name as application name

app name : greet then file name : greet.yml

app name : welcome then file name : welcome.yml

Git Repo : https://github.com/ashokitschool/configuration_properties

2) Create Spring Starter application with below dependency

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

3) Write @EnableConfigServer annotation at boot start class

```
@SpringBootApplication
@EnableConfigServer
public class Application {

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

}
```

4) Configure below properties in application.yml file

```
spring:
cloud:
config:
server:
git:
```

uri: https://github.com/ashokitschool/configuration_properties
clone-on-start: true

5) Run Config Server application

=====
Config Server Client Development
=====

1) Create Spring Boot application with below dependencies

- a) web-starter
- b) config-client
- c) dev-tools

```
<dependency>  
<groupId>org.springframework.cloud</groupId>  
<artifactId>spring-cloud-starter-config</artifactId>  
</dependency>
```

2) Create Rest Controller with Required methods

```
@RestController  
public class WelcomeRestController {  
  
    @Value("${msg}")  
    private String msg;  
  
    @GetMapping("/")  
    public String getWelcomeMsg() {  
        return msg;  
    }  
}
```

3) Configure ConfigServer url in application.yml file like below

```
server:  
  port: 8081  
spring:  
  config:  
    import: optional:configserver:http://localhost:8080  
application:  
  name: greet
```

4) Run the application and test it.

=====
====

=====
Redis Cache

=====

- 1) What is Cache ?
- 2) Why we need to go for Cache ?
- 3) What is Redis ?
- 4) Spring Boot with Redis Integration

=> Cache is a temporary storage

=> Cache will represent data in key-value format

=> By using Cache data, we can reduce no.of db calls from our application.

Note: DB call is always costly (it will take more time to execute)

=> By using Cache data (Cache Memory) we can increase performance of the application.

Redis Cache is one of the most famous Cache available in Market

=> The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.

=> Spring Boot provided below starter to communicate with Redis Server

springboot-starter-redis

=> We will use below components to communicate with Redis Server

- 1) JedisConnectionFactory : It represents connection with Redis Server
- 2) RedisTemplate : It provides methods to perform operations with Redis Server
- 3) OpsForHash: It is providing methods to perform operations based on Hash key

put (...)

get(..)

entries (.)

delete(..)

Key-Value

KEY - EMP
101 - Raju

102 - Rani

KEY - BOOKS

103 - Java

KEY - CITIES

104 - Hyd

```
=====
Working with Mono & Flux Objects
=====
```

- > Mono & Flux objects are used to achieve reactive programming
- > Reacting Programming means Non-Blocking execution based on events
- > In Spring 5.x version Reactive Programming got introduced
- > To work with reactive programming Spring Boot provided below starter

'spring-boot-starter-webflux'

- > Mono object represents single response
- > Flux object represents stream of responses

```
-----Binding class-----
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CustomerEvent {

    private String name;
    private Date eventDate;

}
```

```
-----RestController-----
@RestController
public class CustomerRestController {

    @GetMapping("/event")
    public Mono<CustomerEvent> getCustomerEvent() {

        CustomerEvent event = new CustomerEvent("Smith", new Date());

        Mono<CustomerEvent> mono = Mono.just(event);

        return mono;
    }
```

```
@GetMapping(value = "/events", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public ResponseEntity<Flux<CustomerEvent>> getCustomerEvents() {
```

```
// Creating Customer data in the form of object
CustomerEvent event = new CustomerEvent("Smith", new Date());

// Create Stream object to send the data
Stream<CustomerEvent> customerStream = Stream.generate(() -> event);

// Create Flux object with Stream
Flux<CustomerEvent> customerFlux = Flux.fromStream(customerStream);

// Setting Response Interval
Flux<Long> interval = Flux.interval(Duration.ofSeconds(3));

// Combine Flux Interval and Customer Flux
Flux<Tuple2<Long, CustomerEvent>> zip = Flux.zip(interval, customerFlux);

// Getting Flux value from the zip
Flux<CustomerEvent> fluxMap = zip.map(Tuple2::getT2);

// Returning Flux Response
return new ResponseEntity<>(fluxMap, HttpStatus.OK);
}
}
```

=====
Redis Cache
=====

- 1) What is Cache ?
- 2) Why we need to go for Cache ?
- 3) What is Redis ?
- 4) Spring Boot with Redis Integration

=> Cache is a temporary storage

=> Cache will represent data in key-value format

=> By using Cache data, we can reduce no.of db calls from our application.

Note: DB call is always costly (it will take more time to execute)

=> By using Cache data (Cache Memory) we can increase performance of the application.

Redis Cache is one of the most famous Cache available in Market

=> The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.

=> Spring Boot provided below starter to communicate with Redis Server

springboot-starter-redis

=> We will use below components to communicate with Redis Server

- 1) JedisConnectionFactory : It represents connection with Redis Server
- 2) RedisTemplate : It provides methods to perform operations with Redis Server
- 3) OpsForHash: It is providing methods to perform operations based on Hash key

put (...)

get(..)

entries (.)

delete(..)

=====
Spring Boot with Redis Integration
=====

0) Download Redis Server

Link : <https://www.mediafire.com/file/ul4aeeirc8nrs2a/Redis-x64-3.0.504.rar/file>

-> Extract rar file and Run Redis-Server.exe and Run Redis-Client.exe

1) Create Spring Boot application with below dependencies

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

```
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
</dependency>
```

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

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

```

<artifactId>spring-boot-devtools</artifactId>
<scope>runtime</scope>
<optional>true</optional>
</dependency>
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<optional>true</optional>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>

```

2) Create binding class to represent data

```

@Data
public class Country implements Serializable {

private Integer sno;
private String name;
private String countryCode;

}

```

3) Create Redis Config Class

```

@Configuration
public class RedisConfig {

@Bean
public JedisConnectionFactory jedisConn() {
JedisConnectionFactory jedis = new JedisConnectionFactory();
// Redis server properties
return jedis;
}

@Bean
public RedisTemplate<String, Country> redisTemplate() {
RedisTemplate<String, Country> rt = new RedisTemplate<>();
rt.setConnectionFactory(jedisConn());
return rt;
}
}

```

4) Create RestController class with Required methods

```

@RestController
public class CountryRestController {

```

```
private HashOperations<String, Integer, Country> opsForHash = null;
```

```
public CountryRestController(RedisTemplate<String, Country> rt) {  
    this.opsForHash = rt.opsForHash();  
}
```

```
@PostMapping("/country")  
public String addCountry(@RequestBody Country country) {  
    opsForHash.put("COUNTRIES", country.getSno(), country);  
    return "Country Added";  
}
```

```
@GetMapping("/countries")  
public Collection<Country> getCountries() {  
    Map<Integer, Country> entries = opsForHash.entries("COUNTRIES");  
    Collection<Country> values = entries.values();  
    return values;  
}
```

```
}
```

5) Run the application and test it using Postman.

```
{  
    "sno": 1,  
    "name": "India",  
    "countryCode": "+91"  
}
```

=====