Introduction to Spring boot :

Spring Boot is one of the module which was implemented on the top of the Spring framework.

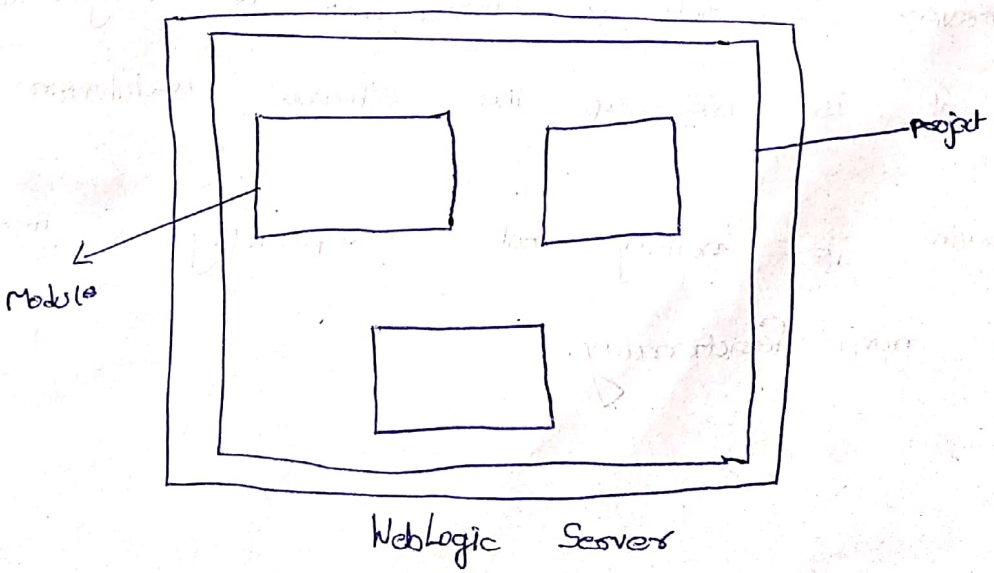
Spring Boot is used for developing a micro service based applications.

In our s/w environment, there are two types of architectures to implement a high level system.

1. Monolithic Architecture
2. Micro Service Architecture

Monolithic Architecture :

In this architecture, the entire project i.e. collection of modules are composed into a war or ear and it will deployed into target server instance like tomcat or weblogic or jboss etc...



From the diagram of Monolithic architecture

→ All modules of the project deployed on the same server instance and so managing the load is a challenging task.

→ If something went wrong with the server machine it impacts overall system to went down.

→ For accessing different services, all requests are hitting with the same server and once again increases a load.

→ On demand load - balancing is not possible in this architecture i.e base on the bussiness hours increasing and decreasing instances of the server machine. (Servers machines setup in diff areas)

To overcome these above problems, we need to break a monolithic based system into mini functionalities. And each functionality to be deployed in its own server and it must run in its own process too. This process is nothing but a Micro Services Architecture.

Micro Service is not a framework or Programming language and it is one of the software architecture model.

Micro Service is nothing but encapsulating a micro functionality as mini functionality.

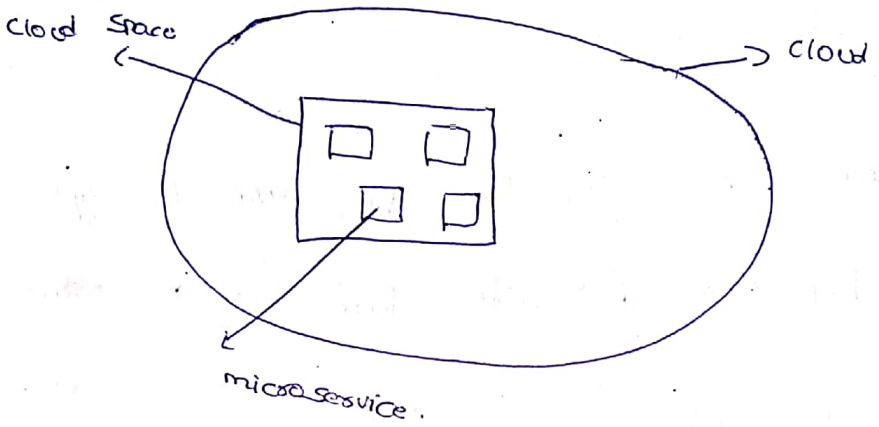
Example

Booking_Service --> booking a ticket

CancelService --> to cancel a ticket

Refund_Service --> to refund amount back

According to the microservices, we can create a unique database instance seperately.



Advantages of microservice:

It allows horizontal scaling i.e increasing or decreasing instances of the microservice ordered.

Note: Here each instance is nothing but a single server.

If something went wrong in the microservice, only that service will down and other services are still up.

Performance is good i.e reducing the load on the microservice by spanning into multiple instances.

Drawbacks

1. Microservices communication is difficult task because these services are in dynamic ip addresses in the cloud.

Each microservices is having a unique database and so managing the data across other services is also difficult task.

To overcome the above problems we need to implement the cloud concepts with microservices by using "Spring Cloud" module.

Currently Pivotal team (opensource company) introduced two types of Spring boot versions.

1. Spring boot 1.x (supported upto Java 7 only)
2. Spring boot 2.x (Supported from Java 8 onwards)

Spring boot 1.x introduced in 2014.

Spring boot 2.x introduced in 2018.

→ Spring boot 1.x supports only synchronous based microservices.

→ Spring boot 2.x additionally supports for asynchronous based microservices

What is the diff b/w Spring boot and Spring cloud?

Spring boot is for developing a microservice application as cloud enabled application (i.e enriched with the cloud concepts).

There are four types of components

1. Starters
2. Auto Configuration
3. CLI
4. Actuator.

Starters

It is one time step for the java programmers to setup all the required dependencies for type of the module to integrate in the project.

Infact starter is also one of the dependency which contains bundle of transitive dependencies with compatible versions which are managed in the pivotal repository.

All predefined starters must be followed with a given convention

For all the modules of the

some of the built in Spring boot starters are as follows.

1. spring - boot - starter - web → for mvc & restful web applicatio
2. spring - boot - starter - jdbc → for spring jdbc module integratio
3. spring - boot - starter - security → for integrating spring Security module.

4. Spring-boot-starter-test → for performing unit testing and integrating testing with a help of Junit and Mockito.

5. Spring-boot-starter-data-jpa → for integrating spring data jpa module.

Spring boot is a opinionated framework. i.e we can customize the spring boot by adding some addition code to support for our project requirement.

We can create customized starters also and we can reuse across other microservice applications.

Auto Configuration :-

It is a process of performing boiler plate configuration to our added starters.

i.e for each spring boot starters there is a auto Configuration classes which are responsible to perform the configuration or integration of the respective module into our application.

For Example

1. spring-boot-starter-jdbc :-

For this starter, internally the autoconfiguration class is responsible to setup database configuration according to the "identified jdbc" driver in the classpath. (If driver is not available then condition is false).

Note:-

If jdbc driver is not available on the class path auto configuration process for this starter is skipped.

2. Spring-boot-Starter-web :-

For this starters, internally auto Configuration classes Performs a below operation.....

1. It configures "DispatcherServlet" → Front controller.
2. It starts a embedded i.e Tomcat, Netti, Jetti
   ↓ default.

3. Spring Boot CLI :-

CLI (command line interface) used by the groovy programmers to generate a groovy scripts for the spring boot application and later these scripts translated into byte code at runtime.

Here groovy programmers no need to work with starters or auto configuration because these libraries are added at runtime depends on the code developed in the groovy scripts.

4. Actuator :-

This component is responsible to provide post production support to our spring boot application like health, metrics and memory management.

In the earlier days i.e for monolithic application we have to use a thirdparty s/w to monitor the health or metrics related to the applications.

There are two ways to create a spring boot application.

1. Using https://start.spring.io

By using this website we can provide our project meta data information for creating a skelton project of the spring boot.

2. Using IDE we can create a skelton spring boot project.

Note:-

STS (Spring Tool Suite) & IntelliJ ide's are by defaultly supported to spring boot. i.e they are enabled with "spring-boot" plugin.

For other ide's like netbeans, eclipse, idevelopes etc we must install/configure spring boot plugin.

Internally this spring boot plugin connects with "start.spring.io" service for creating a skelton Spring boot application.

This "start.spring.io" website is maintained by Pivotal team.

To create a custom conditional annotation for reusing a condition class across multiple beans we can follow a below steps

1. create a annotation by using "@interface" keyword.
2. Add the properties for the annotation and it must return some type of value (i.e void is not allowed).

Note:-

we can also make property as optional by declaring default value using "default" keyword.

3. Add "@Target" and "@Retention" as a meta annotations to indicate where to apply our annotation and how long the annotation to be available.
4. Finally add "@Conditional" as a meta annotation to make our annotation as custom conditional type.

ex:-

```
@Target ( ElementType. METHOD)
@Retention ( RetentionPolicy . RUNTIME)
@Conditional ( value = MyDatabaseCondition . class)
Public @interface DatabaseCondition
{
    String dbName();
}
```

Based on the above concept, spring boot internally created some of the custom conditional annotations which are

internally used by multiple auto configuration classes belongs to the multiple starters.

Some of the custom conditional annotations like

1. @ConditionalOnBean
2. @ConditionalOnMissingBean
3. @ConditionalOnClass
4. @ConditionalOnMissingClass

Spring Data:-

This module is introduced as a part of Spring framework. This module provides a ready made solution for the persistence logic.

This module can be integrated with relational databases and no sql databases as documented databases like cassandra or mongo.

Features of Spring data module:-

1. It generates a runtime proxy classes for the given repository interfaces which encapsulates persistence logic implemented on jpa (Java persistence api).

2. Each generated proxy class is registered as spring bean component i.e we can use in our dependency injection process.

3. It provides some set of standard repository interfaces which provides common methods for performing persistence operations.

4. It makes our application as ORM independent type i.e we can plug any kind of ORM Providers at runtime like hibernate or top link or eclipse link etc...

5. It provides a concept of finder methods for deriving select query on demand i.e based on the given naming convention.

Note:-

Finder methods must be begin with either findBy or getBy or fetchBy and it must contain an entity class properties with database clauses.

Spring Data Repository:

Spring Data contains a standard repository interfaces and based on this we can create a custom repository interface by providing our own methods also.

1. Repository
2. CrudRepository
3. PagingAndSortingRepository
4. JPA Repository.

According to the serial order, one interface extends another interface.

Repository is a base interface or marker interface which tells to the spring data module to prepare a proxy class for our interface indirectly.

→ CrudRepository provides a common methods to perform crud operations on the tables.

→ PagingAndSortingRepository additionally provides methods for performing pagination (lazy loading) and sorting operations on the tables.

→ JPARepository is a ultimate sub type which contains all the above interfaces methods and also it provides some additional methods for native operations on the database tables.

Q. what is the advantage of using Spring data module with spring boot application ??

Spring data comes with a starter and which contains auto configuration classes which are responsible to perform a below configuration automatically...

1. Database provider configuration (defaultly uses hikaari datasource) with a help of given jdbc driver.

2- Prepares a native orm provider integration with dataSource provided i.e defaultly hibernate "SessionFactory"

3. Prepares a jpa context i.e constructs EntityManager and EntityManagerFactory objects and which integrates with native orm provider.

Q. How to prepare test cases in spring boot application.

→ Defaultly for every spring boot application, Spring-boot-starter-test dependency added in pom.xml file and it adds "Junit" testing framework and it runs along with our spring boot application during the build process.

Junit is one of the popular testing framework for preparing unit test cases and integration test cases.

Unit test cases are nothing but testing our logic with positive input data and negative input data.

A unit test case means it contains a coupl of test methods i.e a method must be annotated with "@Test" annotation.

Using "@SpringBootTest" we can make our class as not only a test case but also it makes as a spring component to participate in dependency injection process.

Using "@RunWith" we can make our test case to execute along with a given runner class i.e in this case "SpringRunner" class.

SpringRunner class calls our main method from the Spring boot application and then it initializes spring container and spring components registration.

Q. What are the annotations you know in Junit ??

A. @Test, @Before, @After, @BeforeClass, @AfterClass, @RunWith, @TestSuite, @Ignore.

@Query :-

It is used for defining either JPQL (Java Persistence query language) based query or native sql - query for an method declared inside a repository.

@Modifying

It is for allowing modifying operations like insert, update or delete or alter or truncate.

Note :-

From the junit test cases → Spring data module immediately performs a rollback on the current transaction for the methods which are declared externally from the repository.

Using @Transactional and @Rollback(false) we can make test cases related persistence operations to be committed on the database.

Spring data module supports to the two ways of injecting data dynamically into the sql query.

1. Positional parameters as index based
2. Named Parameters

Named Parameters increases good readability i.e understanding about the type of data to be injected to the sql query whenever if number of inputs are more.

ex:-

```
@Query (value = "update teacher set specialization = ?2 where id = ?1",
        nativeQuery = false).
```

Here "?1" means first argument value of our method and "?2" means second argument value of our method.

Note:-

If index positions are not mentioned for the placeholders follows sequential order of arguments given for the method.

```
@Query ( value = "update teacher set specialization = :spec
where id = :teacherid", nativeQuery = false)
```

Here ":spec" is nothing but a named parameter and in the placed of named parameter we can substitute our actual value with a help of "@param" annotation.

PagingAndSortingRepository interface additionally contains two types of methods...

1. findAll (Sort ob)
2. findAll (Pageable ob)

Additionally we can create a custom finder methods by passing additional argument either as Sort object type or Pageable object type.

Using PageRequest.of() we can create and return Pageable type of object with which encapsulates a given page number and number of records to fetch in the page.

Using Sort.by() to create and return Sort object which contains sort direction and column name as property name of the entity class...

Note:
Default sorting type direction is ascending.

Q. How to implement Pagination and along with sorting with sorting using spring data??

A. By preparing our own finder method or creating a method with @Query() by declaring jpa specific query or native query by adding extra argument i.e. pageable type.

ex:- PageRequest.of(0, 1, Sort.Direction.DESC, "name");

Here first two arguments related to pagination specific and last two arguments related to sorting specific.

Spring data module is supported to relationships management between multiple entity classes...

1. one-to-one
2. one-to-many
3. many-to-one
4. many-to-many

The above relations are implemented using a mapping b/w a foreign key column of owning entity class and Primary key column of child entity class.

Using @JoinColumn annotation we can map as class foreign key column with target class Primary key column.

Note:-

Spring data is supported to unidirection and bidirectional relationships between entity classes.

Cascading operation plays a important role for performing persistence operations on the child entity and collecting the data from the child entity and then performing operations on owning entity i.e. parent entity.

Flyway:-

It is one of the database migration tool used by the developers to track the current state of the database. i.e which script is applied currently on the database with a help of the versions defined for each script.

Flyway opensource tool introduced by "BOXFUSE" in the year of 2007 and the most companies are started using in 2018 because of the demand of micro services.

For each microservice as we know that database is unique and for every new feature always some changes to the database is required and it is always maintain in the form of scripts.

The advantage of this approach is....

1. we can track the current state of the database. i.e currently on which script the database is applied.

2. If any bugs is encountered, we can roll back to the past script easily.

Q.

→ 1. Flyway always checks for flyway_history table in the target database instance for the past updated script version.

Case-1:-

If this table is not presented, flyway considers as a first database migration and then it performs a below steps.

1. It creates a new flyway_history table
2. It applies a given database script on the database i.e executes all the given sql queries from the script.
3. Finally it adds the entry of the current applied script details into flyway_history table.

case-2:-

If table is already presented, then it gets the latest version of the script applied earlier and it compares whether any current script version is greater than the fetched version or not...

1. If yes, the given script is applied on the database and adds script entry details into "flyway_history" table.
2. If no, the given script is considered as old script and migration is ignored.

Note:

To get more information about flyway, we can go to official website ... https://flywaydb.org.

Steps to integrate flyway into Spring boot application :-

1. copy the flyway plugin for maven from https://flywaydb.org and paste into our pom.xml file of our Spring boot application.

2. create a database scripts by adding a Prefix as "V<version-number>_<script name>.sql"

Note:

The above db scripts must be save inside a below path

src/main/resources/db/migration.

3. Using a below command to ask the flyway for performing database migration.

mvn flyway:migrate.

Embeded Databases

Spring Boot defaultly bundled with a below databases

1. H_2
2. HSQL
3. Derby

Embeded databases are the light weight databases and used for performing a test feature implementation i.e Developer no need to do a external database setup and they can focus on actual feature implementation.

H_2 database is more popular than others embedded database because it is implemented on java and also it provides h2-console (a webpage) to browse all the tables which are created.

Whenever any of the above embeded database driver and detected in the classpath automatically it setup the database instance with a name as "testdb" and it configure the table creation base on two different ways ...

1. If hibernate.ddl mode is not set to "none" then hibernate will take care about creating a table for each given entity class.

2. If we set to "none" then it looks for "schema.sql" and "data.sql" scripts in the application classpath.

Note:-

To access "h2-console" (i.e. webpage) we must add spring-boot-starter-web in pom.xml file and it must access with a below url...

http://localhost:</portno>/h2-console.

Spring Cloud:-

Cloud is the ready made infrastructure setup for hosting our applications.

Benifits of cloud:-

1. No need to invest on buying a new server infrastructure.
2. No need to invest on network team for tracking the health status of the server.
3. Whenever if services are grown in future we can expand the memory space in the cloud.

At the End of the day every microservice application must be published into the cloud.

Microservice application must face a below changes once deployed in the cloud.

1. Communicating one microservice with another.
2. If service went down no backup
3. Security
4. Load Balancing.

To meet the above challenges, Spring Cloud framework provides a cloud specific components and using this we can develop our microservice application as cloud enabled application.

In our market, there are two types of cloud Platform

1. AWS (Amazon webservice)
2. Cloud Foundry (Cloudfoundry)

Spring cloud, provides a cloud components specific to the cloud platform i.e. whenever if we are migrating from one cloud platform to another we have to modify our existing microservice applications.

CloudFoundry is a opensource community provides some set of specifications (set of rules) to setup a cloud and base on this there are many thirdparty cloud providers like PCF (pivotal cloud foundry), Heroku, Google cloud, Microsoft Azure etc...

Spring - cloud - config :

Cloud - config is one of the cloud component introduced by the spring cloud and it is used for fetching external configuration from the selected profile into microservices.

Here "selected profile" means its related to the type of stages which are applied in the part of development life cycle like development, quality analysis, User acceptance test and production.

The continous integration tool like Jenkins or teamcity or bamboo etc is responsible to perform a build automation for performing continous delivery of the microservices.

The benefit of external configuration is properties which are related to ~~envir~~ environment specific like database details or security details etc which are changed from one stage to another stage and so no need to rebuild our microservices again and again and it will fetch automatically based on activated profile matched configuration.

Q. What are the ways to perform external configuration in microservices ???

1. Using "spring - cloud" config service module
2. Using Zookeeper
3. Using consul
4. Using Vault

Config - service module contains two types.

1. Config - server module
2. config - client module

config - server module is responsible to contact with a given version control system like GIT or SVN and dynamically based on the requested active profile and it uses a default branch as "master" for fetching matched profile in the json format.

Config - client module is responsible to contact with config - server module to fetch a given active profile with a given branch from the target version control repository.

Q. What is the difference between YML & properties file??

YML file maintains list of properties in the tree structure. And it provides good readability. It also uses "SnakeYAML" as a runtime interpreter to validate each entry in the YML file

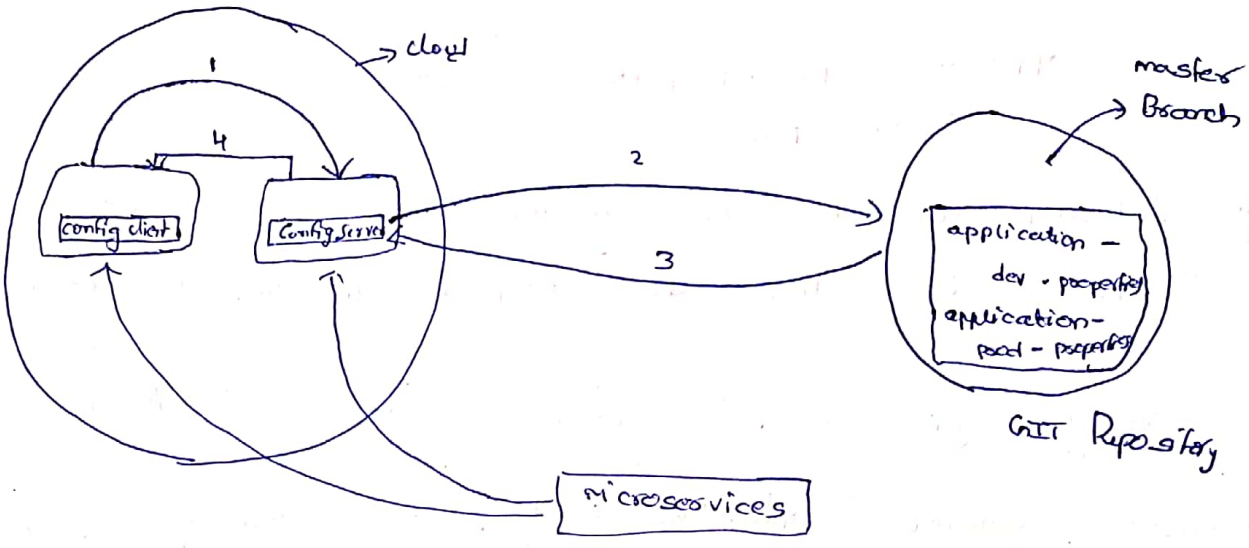
Properties file uses a combination of (key, value) pair which was seperated with "=" sign. This format creates confusion to the programmers who are dealing with a big property names multiple times with/by just changing a last word of the property.

Q. What is the difference between application.yml and bootstrap.yml files??

Application.yml files are readed by autoconfiguration classes i.e. as soon as main method is invoked from the given spring boot application.

bootstrap.yml readed by some specific modules before invoking our main method of the Spring boot application (i.e to perform some preprocessing related activities).

High level Architecture of config-server & config-client modules



For testing purpose we can configure a git local repository for integrating with config-server microservice.

Steps to configure a git local repository:-

1. Download git command line tools from the below link.

   https://git-scm.com/downloads

2. Create a empty directory in the drive. (Any choice of drive).

   Ex:- e:\>mkdir mygitrepo

   e:\>cd mygitrepo

   e:\mygitrepo>

3. Create a empty git repository

   e:\mygitrepo>git init

Note:-

Here we created a git local repository which is responsible to track all the files changes with a help of versions and timestamps which are recorded in the special folder i.e. ".git" (a hidden folder).

4. Add a below files in the new folder i.e "mygitrepo".

// application - dev - properties → file-name

myenv = Development

// application - prod . properties

myenv = prod

5. By default my new files or modified files are treated by the git as "untracked files" that are displayed in the red color.

Note: using "git status" command we can track the file status whether untracked / tracked / yet to commit / clean directory.

6. To change the newely added files to tracked files we use a below command ....

e:\mygitrepo> git add *

Finally save the changes to the files to the git using a below command.

e:\mygitrepo> git commit -m "Add: Added two new files"

Service Discovery :-

It is a process of performing the discovery of one microservice with another dynamically.

Whenever microservices are published into the cloud we are facing a below challenges.

1. How one microservice to connect with another microservice.
2. How to communicate with multiple instances of the microservice i.e loadbalancing.

The above challenges are handled by the concept of the cloud i.e "service Discovery"

Steps to implement a "Service Discovery" :-

1. Add "Service Discovery server starter" to our Spring boot application.

Note :-

currently we have Eureka Server, Zookeeper, Consul.

2. For Eureka Server we need to add a below annotation for our main class of the spring boot application i.e. @EnableEurekaServer.

3. Add "eureka server" specific properties in application.yml file of application.properties file.

Whenever "Eureka Server" starter added in our application, the auto configuration classes of this starter performs a below operations internally....

1. It verifies whether "@EnableEurekaServer" is defined on configuration class or not.

2. Once it is detected, it fetches the Eureka server specific properties from the given file and base on this it starts eureka server services internally with a help of a tomcat server.

Note:-

In fact, eureka server is also a microservice application and we have to disable eureka client for this application because this microservice application intention is as a "Service Discovery".

In order to communicate one microservice with another, microservices must register with discovery server to participate in the service discovery process and usually this registration is performed in the background with a help of either "EurekaClient" or "DiscoveryClient" modules.

EurekaClient performs "ServiceDiscovery" only on "Eureka Server" and it is a product of "Netflix"

DiscoveryClient performs "ServiceDiscovery" on any type of discovery server and it is a product of "spring cloud".

Q. How to perform service discovery operations in our microservice?

A. with a help of either "EurekaClient" or "DiscoveryClient" we can write a discovery logic followed with our loadbalancing algorithm.

To avoid the above manual process, we can use a ready made component i.e "Ribbon" provided by the netflix and now this component is accepted by other Discovery servers also.

Ribbon supports for discovery process followed with either static (or) dynamic loadbalancing.

Static loadbalancing means declaring the multiple instances i.e ipaddress and portno of each instance of micro service in a application.yml or properties file and in this case "DiscoveryServer" is not required.

Dynamic loadbalancing means contacting with the discovery Server to fetch available instance of the target microservice and base on this loadbalancing algorithm pickups the instance of the microservice and makes a rest call.

Steps to implement a static load balancing :-

1. Prepare a configuration for target micro-service to consume in application.yml file.

2. Make sure "Eureka Client" to be disabled in the same configuration

3. Declare @RibbonClients annotation on the main class which contains array of @RibbonClient annotation.

Ex:- A sample configuration for PatientService

```
Patient-Service:
  ribbon:
   eureka:
    client:
     enabled: false

list-of-servers: localhost:9191, localhost:9192
```

Ribbon component by defaultly uses "EurekaClient" to perform dynamic load balancing and we are going to disable eureka client for that particular service to implement a static load balancing.

Q. Explain how the static loadbalancing works internally??

A. In the micro-service deployment, the autoConfiguration classes of the ribbon is responsible to perform a below operations....

1. It reads the given ribbon configuration properties from the given yml (or) properties file and based on this it decides to implement either static (or) dynamic load balancing.

2. @RibbonClient for this annotation internally proposes a new RibbonClient instance to perform loadbalancing for that target microservice.

3. RibbonClient instance replaces a given rest endpoint url with actual instance url before making a rest call.

Steps to implement dynamic load balancing.

→ Add "Eureka - Client" module or "Discovery - Client" module as a part of our spring boot application.

→ Define "Eureka Server" or other discovery server properties in the yml file or properties file.

→ Add @LoadBalanced annotation for the "RestTemplate" object creation bean method.

→ In the resttemplate uri make sure to provide the target microservice nick name which has been registered in the DiscoveryServer.

Ex:

```
List<String> response = restTemplate.getForObject
    (new URI("http://PATIENT-SERVICE/api/v1/fetch
        /patients", List.class);
```

RibbonClient is responsible to findout the instance uri after loadbalancing and it replaces the uri internally with the "PATIENT-SERVICE" and then it makes a rest call.

API - GATEWAY

It is one of the design pattern used in the microservice to meet a below requirements...

→ To hide endPoints of the microservices to the outside world i.e external applications.

→ To implement centralized configuration like security, log management, transaction control etc..

→ To implement loadbalancing either static or dynamic.

currently, there are two cloud components which are implemented an API - GATEWAY design pattern.

1. Netflix Zuul
2. Spring - Cloud - Gateway

Netflix zuul is used in spring boot 1.x versions and more over in 2.x versions we are using Spring - Cloud - gateway because additionally it is supported for asynchronous transactions.

Netflix zuul is supported only for YML configuration i.e all internal microservices endPoints must be declared and it increases more complexity.

Spring - Cloud - GateWay is supported both for YML and programmatic configuration also.

Steps to configure ZUUL:

1. Add Netflix - zuul - Starter as a part of our Spring boot Application.

2. Add "@EnableZuulProxy" annotation on the top of the main class of the spring boot Application.

Note:

This annotation enables autoConfiguration classes of the ZUUL Starter to activate zuul server internally. (This is part of zuul1 zuul2 is presently not supported by springbootapp)

3. Define zuul configuration or zuul routes in application.yml file.

Internal flow of ZUUL1:-

1. whenever external application submitted a request to the zuul, internally it performs a below operations ...

   i. It verifies whether a given request path is matched with any of the path declared in the zuul route configuration or not.

Case 1:-

If a route path is matched with the given request path then it performs a below operations ...

1. It fetches a given service-id for the given route and asks a eureka client or discovery client to perform discovery process for the given service-id i.e fetching all the available instances information and that is updated into the cache.

Note:- This step is processed only for dynamic loadbalancing.

2. It prepares a RibbonClient instance for the target service to perform loadbalancing.

3. RibbonClient returns the loadbalanced url i.e. the selected instance url back to the zuul.

4. Finally, zuul hands over this loadbalanced url to the activated rest client library i.e. defaultly apache HttpClient for making a rest call to the target instance url.

Case 2:-

In case if route path is not matched then simply it returns a error back to the external application.

Initially Zuul is the official product of Netflix and later it is converted as OpenSource.

In Market, there are two versions of Zuul.

1. Zuul 1:-

It is compatible only for spring boot 1.x version and supported only for synchronous communication.

2. Zuul 2:-

It is compatible for spring boot 2.x version and spring 5 version features i.e spring web flux reactive support i.e asynchronous communication.

Spring - Cloud - Gateway :-

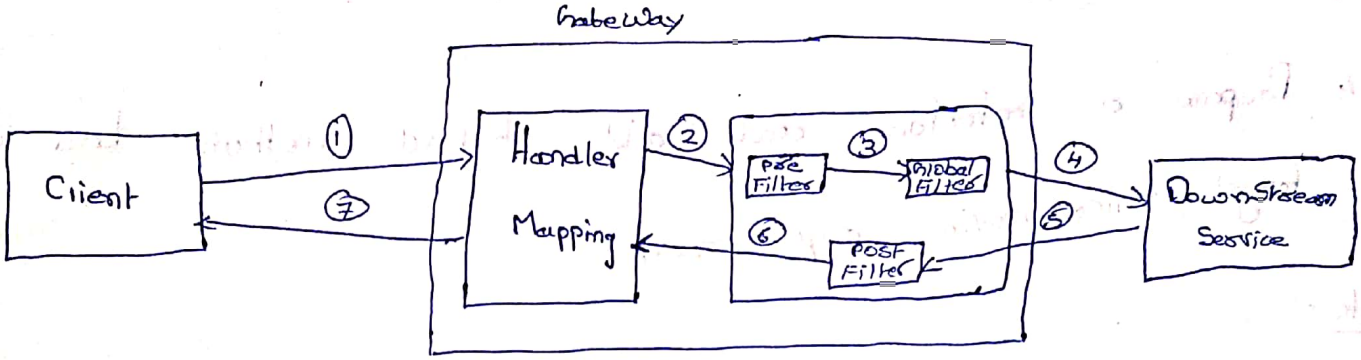
It is one of the cloud component implemented on "Reactive Gateway" architecture model.

It is provided by the Pivotal Team in 2018.

Reactive gateway means it is a concept of asynchronous or non-blocking request processing.

This gateway module is implemented with Spring Boot 2.x and Java 8 features and Spring 5 features and Spring Web Flux.

High level Architecture of Spring - Cloud - Gateway :-



Feign Client :-

Feign client is a product of Netflix and later converted as "OpenSource".

Feign client is used for below cases ...

1. To eliminate call the target microservice without writing a consuming logic explicitly i.e using rest client library.

2. To perform the discovery and loadbalancing operations without writing a consuming logic explicitly.

Steps to integrate FeignClient with Microservice :

1. Add "Feign - Client - Starter" to our application.

2. Add EurekaClient or DiscoveryClient Starter and Ribbon Starter to our application.

3. Add "@EnableFeignClients" for the main class of the application.

4. Prepare a interface and add abstract methods base on the target microservice requirement.

Note:-

Here requirement means the type of request operation, input parameters, return type value, Produces and Consumes.

Add "@FeignClient" for the interface and provide the Service-id of the target microservice to perform discovery operation internally.

Circuit Breaker:-

It is one of the design pattern used for blocking the original service method or controller method upto some period of time and diverting the requests to fallback method.

This pattern main goal is to avoid service internal communication failure i.e even if any of the microservice went down.

In microsesvices envisonment, we cannot give goarantee that at which case the sesvice was down and so its better to handle these case with a help of circuit breaker.

This pattern is implemented by different cloud components like Hystrix, Pivotal CloadFoundary - Circuit Breaker etc.

Hystrix is a opensource component i.e common to any kind of cloud which was related to cloud foundry.

Note:
Hystrix is a product of "Netflix" and later converted as "OpenSource".

Steps to configure Hystrix into micro-service application :-

1. Add "Hystrix - Starter" in our spring boot application.
2. Add "@Enable Hystrix" for our main class of the spring boot application.

Note:
This annotation is scanned by autoconfiguration class belongs to Hystrix - starter library and which is responsible to activate Hystrix component.

3. For actual method in our controller, add "@Hystrix-Command" to provide a fallback method.

Note:-

@HystrixCommand registers our annotated method meta data information with "Hystrix" component in the initial startup.

4. Provide a fallback method i.e exactly like a actual method signature i.e method return type and method parameters

Circuitbreaker comes with three different states...

1. Half - open
2. Open
3. Closed

Half - open means its a default state and in this state it tracks each and every actual method is executed properly or not i.e. if exception is encountered in the actual method automatically it invokes the given fallback method.

Open state means usually request diverted to the fallback method of the given actual method until the refresh time expires.

Closed state is generally applied whenever if the entire service is completely down i.e. instance of the server.

Integrating Cache with Circuit Breaker :-

Cache is a temporary memory maintained until the server instance is alive.

Cache is of two types ...

1. Browser side cache
2. Server side cache

Browser side cache is generally used for reducing same requests from multiple users i.e. response data returned from the server originally stored in the browser side initially and later it reuses the cache again for restoring the data.

Server side cache is used for storing the data in the server side for reducing database calls or using to return the data from the cache even if service went down.

There are different thirdparty cache providers which are available to implement a cache and some of them like spring cache, redis, ehcache etc ...

Steps to configure Spring cache :-

1. Add "spring-cache" starter in Spring boot application

2- Activate caching module by adding "@EnableCaching" on the main class of as spring boot application.

3. Configure bean method which returns CacheManager object which is responsible to create, update, insert and fetch details from the cache.

4. Use @CachePut annotation on the required method for storing a given argument and return value from the method into the cache.

5. Finally we can fetch the data from cache by using "getCache()" from cachemanager by providing a cache name as a argument.

Note :-

getCache() returns cache object for the given cache and from this we need to invoke get() by passing key name as a argument to fetch actual data.

Consul :-

1. It is introduced by Hashicorp. It performs ServiceDiscovery, service configuration.

2. Here service configuration means maintaining the data in key, value pair.

3. Using @Value annotation we can get binded value for the given key.

Steps to configure consul integration with spring boot Application

1. Add either consul - config starter or consul - discovery starter or both into our pom.xml file.

2. Add "@EnableDiscoveryClient" as a optional type for the main class of the spring boot application.

3. Define the consul instance id for the microservice and used by the discovery client internally to register with consul server.

Note:-

consul distribution contains a single batch file which is responsible to start or stop consul in client (or) server mode.

consul agent

--> consul agent -dev (command)

Note:-

http://localhost:8500 is the default port for accessing the home page of the consul server.

Actuator:- It is one of the Spring boot core Component.

It is used for providing a post production support for the microservice applications.

Post production support is nothing but monitoring the health and metrics management.

These metrics nothing but load balancing information, application internal issues, number of requests trace, bean configuration etc....

Health means service status i.e up or down and also memory details like jvm heap size and permgen space.

To use actuators add "spring-boot-starter-actuator" as a dependency to our pom.xml file of our spring boot application.

The auto configuration classes belongs to this starter is responsible to activate common end points of the actuators.

Sensitive end points must be activated through yml file (or) properties file.

End Point	Description
1. /Actuator	1. It provides a platform to refer other end points of the actuator.
2. /Audit Events	2. It provides information about all audits and events on given end point.
3. /AutoConfig	3. It provides AutoConfigeration reports (i.e Json data) for all the auto configurations applied in the application.
4. /Beans	4. It provides information about all beans configued in the application.
5. /ConfigProps	5. It shows the detailed information about a configuration properties. (Jvm arguments, environment variables).
5. /Env	6. It shows different environment properties for the configurable environment in the Spring.
6. /Flyway	4. It shows the database migration metadata information.

8. / Health

8. It displays health information like security, Authentication of number of connections which are made, Service Status.

9. / Metrics

9. It shows metrics information about our application.

Note:-

1. / Shutdown is a sensitive end point which is responsible to stop a current instance of the Server.

2. using a below property we can enable or disable any end point related to the actuator.

management.endpoint.<id>.shutdown.enabled = true.

3. using a below property we can enable or disable

management.endpoints.enabled-by-default = false.

4. To enable full report about the health of the service, add a below property in either properties file or yml file.

management.endpoint.health.show-details = always

⁂ Spring-boot actuator allows to create a custom health actuators.

⁂ Custom health actuator is used to track the information about the target microservice which was tightly coupled with a current microservice.

⁂ Predefined health actuator returns only the status and diskspace of our current microservice.

Steps to create a custom health actuator :-

1. create a java class must implement HealthIndicator interface.

note:- HealthIndicator belongs to actuator library

2. Annotate this class with "@Component" and so that Spring container makes this component to the actuator.

3. Override "health()" and provide the health information either up or down by using the methods belongs to the Builder class which is a nested class of "Health".

Custom EndPoints :-

Usually this type of endpoints are used for exposing additional information about the microservice to the outside world. (i.e external applications).

Steps to create a custom endpoint :-

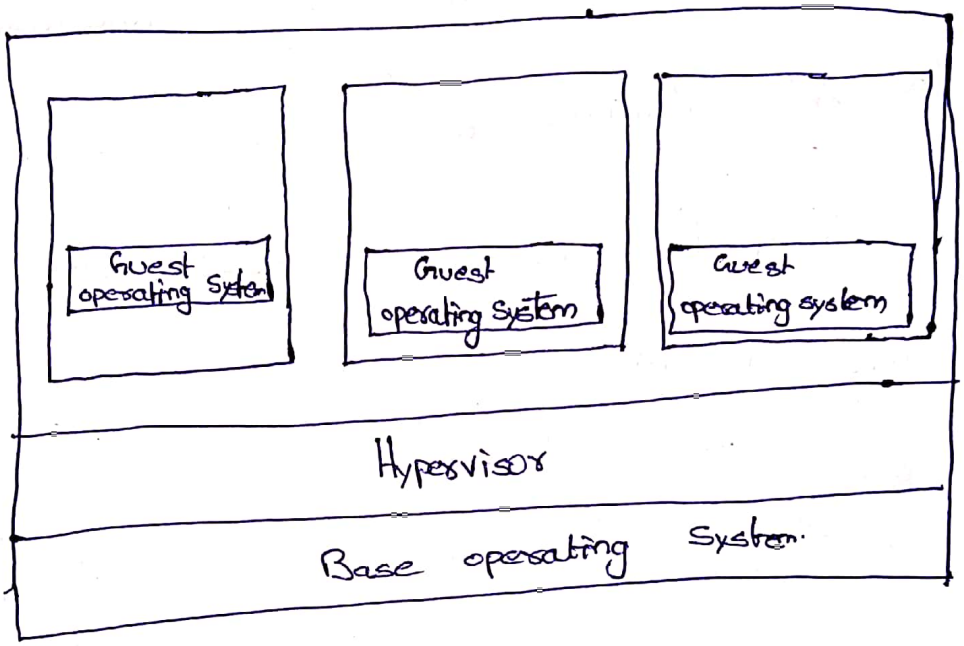
1. create a java class must implement Endpoint interface or annotate this class with @EndPoint.
2. Apply properties for the endpoint like endpoint id, isSensitive, isEnabled

Dockers :

There are two types of deployment models which are available for the software applications...

1. Shared VM
2. Containerization

Shared VM :-



In the above architecture using VMWare software (or) Oracle virtual box etc. we can configure shared vm's.

Each VM is nothing but a process which contains a guest operating system, application setup. (java s/w, maven, tomcat server etc...).

Managing multiple VM's concurrently is a heavy weight operation because it consumes huge amount of memory and hardware infrastructure.

4. Due to the Guest operating system configuration demands each VM memory in the GB only.

5. The only one benifit of this architecture is security i.e. VM's cannot interact with base operate system (os) hardware infrastructure directly.

Note:-

The above architecture is best for monolithic based software applications.

This architecture is not compatible with microservices architecture model because of the below cases...

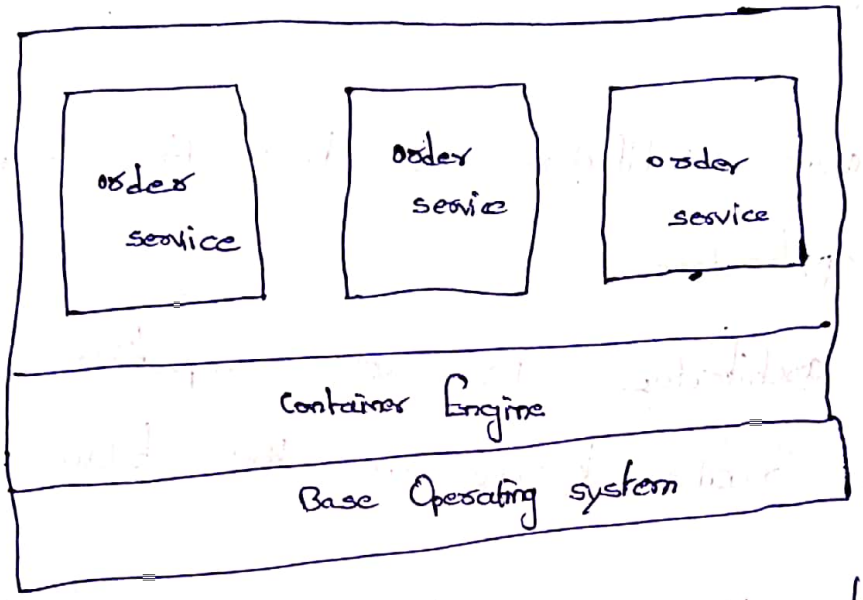
1. Elasticity (depends on the business requirement need to increases instances of the microservices on demand and viceversa)

2. Infrastructure sharing (i.e a single operating system and common hardware infrastructure).

To meet these above requirements we need to implement a containerization model.

Containerization :-

It is a process of creating a ondemand instance for installing the application setup to run services of the application.



In the above architecture there are three types of components....

1. Containes Engine
2. Image
3. Container

Containes Engine is responsible to create or delete containes on demand.

Container Engine manages multiple containers parallely by sharing underlying hardware infrastructure and base operating system services.

Image is nothing but it is a package or self executable file which contains set of instructions for preparing a application setup in the new container from the container engine.

Container is nothing but its a instance or a memory which contains required setup for the application to run.

Container is a light weight component as compared to shared VM because it consumes less amount of memory i.e. in MB. (because guest operating system is not required to configure in the container).

This container approach model is implemented by one of the software i.e. Docker.

Docker is introduced by opensource community i.e Docker community and in the earlier days is introduced for linux based operating system compatibility and later released for MAC O/S and WINDOWS.

For windows, there are two types of docker softwares

1. Docker Toolbox
2. Docker For Windows

→ Docker Toolbox is for windows 7, 8 and 10.

Docker Toolbox contains kitematic, oracle virtual box and linux image with docker setup.

→ Docker For windows is for windows 10 professional edition only.

Note:-

Docker for windows contains two types of dockes containers i.e. linux based and windows based.

Generally windows based dockes containes is used mostly for .Net applications.

Steps to configure docker on windows :-

1. create a dockes account from the below link

Goto ⟶ dockes . com ⟶ click on sign in ⟶ create account.

2. Download docker toolbox . exe file from below link

https : // github . com / docker / toolbox / releases

3. For windows 10 professional operating system, download Docker For Windows setup file from the below link.

https : // www . docker . com / products / docker - desktop.

Reactive programming with Microservices

There are two types of programming models ....

1. Imperative model
2. Reactive model

Imperative means it is process of executing the instructions sequentially i.e one after another.

If any long running task is performing in the background like database communication, external software systems communication etc.... it blocks our current request. i.e if it is not allowed to perform other operations parallely and this is known as Synchronous Driven Model.

Reactive model means it is process of executing instructions parallely i.e it doesn't block the current request and it will notify to the subscriber once response is ready to collect.

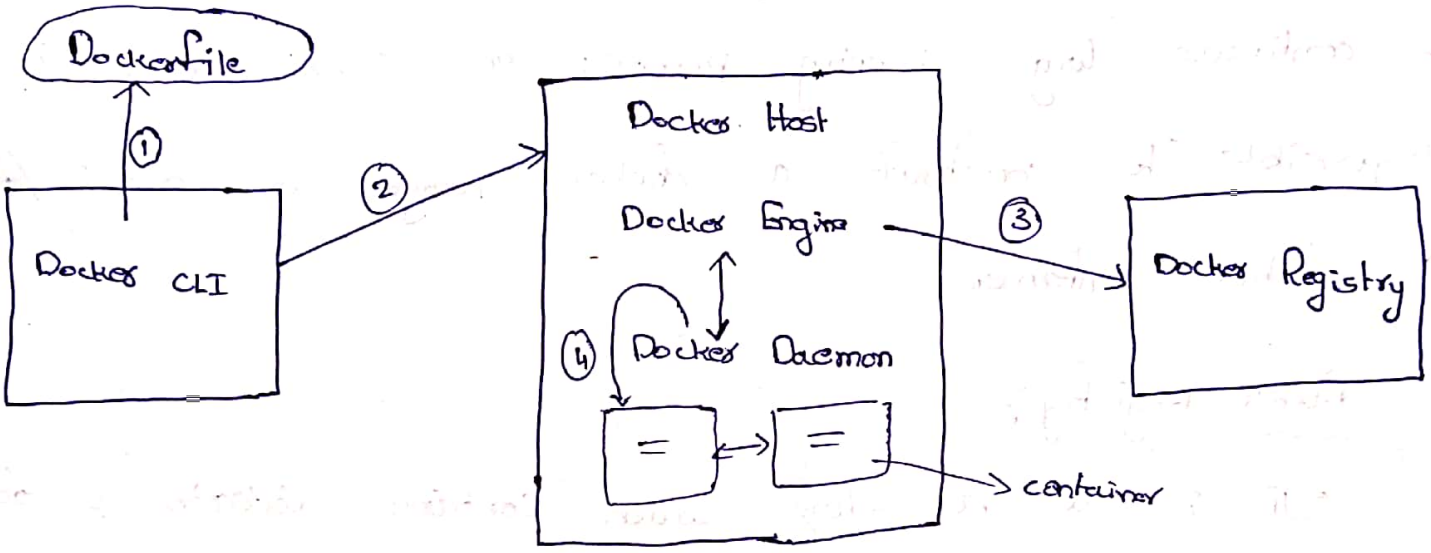
Reactive model supported is provided by java 8 version onwards by introducing "RXJava" module.

RxJava is implemented on "reactive stream specification"

Note :-

Here stream is nothing but a continous flow of data.

Docker High Level Architecture :-



From the above architecture there are 3 types of components in docker.

1. Docker CLI
2. Docker Engine
3. Docker Registry

Docker CLI

It is a command line interface module, uses internally rest api to communicate with docker engine for building a new image or creating/ moving/ deleting a Container.

Docker Engine:-

It internally communicates with a docker daemon i.e a continous long running process or server which is responsible to construct a docker image or creating/deleting a docker container.

Docker Registry:-

It is a repository which contains collection of docker images.

There are three types of docker registries...

1. Local Registry
2. Public Registry
3. Private Registry

Local Registry by default configured by the docker software during the installation process and newely created images or downloaded images are stored inside local registry only.

Public registry is common to all the people and there are two types of public registries.

1. dockerhub
2. docker cloud

private registry is specific to organization internal projects and for this need to get a license.

Devops team is responsible to configure the docker setup either as a client - server model or to be configured in the cloud.

Kubernets or Docker swarm are the concepts which are used for implementing additional features to make docker containers as robust.

1. Auto scaling
2. Load balancing
3. Self healing
4. Secret configuration

Dockerfile:-

It is a unique file to be created with a name as "Dockerfile" without any extension for each microservice application.

Dockerfile contains set of instructions which are readed by the docker daemon for setting up a new image & creating a new container.

Set of instructions can be either in the linux based & windows based i.e depends on the target docker container.

Docker commands :-

1. docker ps :-

   It shows all the list of docker containers

2. docker rm <container-id> :-

   It removes existing docker container binded with a given id.

3. docker rmi <image-id> :-

   It removes existing docker image binded with a given id.

4. docker build -t <tag-name> <dockerfile-path>

   It constructs a docker image with a help of a given dockerfile and binds with a given tag name as a image name.

5. docker run :-

It creates a container from the given docker image and starts application services inside the container.

6. docker log <container_id> :-

It shows the server log runs inside a given container.

7. docker pull <image_name> :-

It downloads the image from the public repository into the local repository of the docker.

8. docker push <image_name> :-

It publishes the given docker image into either public repository or poivate repository of the docker with a help of given docker configuration.

Reactive Stream Specification :-

It is a opensource specification given from multiple companies like pivotal etc.

These specification means some set of interfaces must provide implementation to support for reactive streams.

Reactive Stream Specification provides 4 types of interfaces...

1. Publisher
2. Subscriber
3. Subscription
4. Processor

Processor interface is a subtype of Publisher and Subscriber

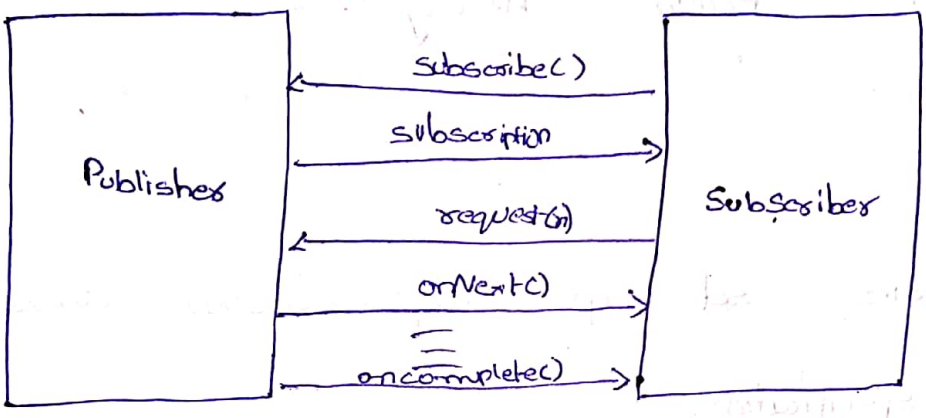
There are three types of reactive libraries which provides implementation for the above interfaces.

1. RXJava (Added in Java 8)
2. ProjectReactor or Reactor (Introduced by Pivotal)
3. Flow class (Added in Java 9)

Reactive stream specification is implemented on Publisher and Subscriber model.

Here Publisher is also known as data Provider and subscriber is also known as data consumer.

High level architecture of Publisher and subscriber model



Subscribers initially calls "subscribe()" to request for subscription from publisher and once publisher accepts subscription, it recieves subscription object from publisher.

Subscriber
~~Publisher~~ receives request amount of data from publisher with a help of request() invocation.

publisher invokes onNext() multiple times depends on the requested amount of data and it collects the data chunks from the publisher i.e nothing but a stream.

During onNext() invocation if any error is encountered, Publisher stops the stream abnormally and invokes onError() from the subscription object.

once requested amount of data sent from the Publisher and finally invokes onCompleted() from the Subscription object.

Modules in Project Reactor Library :-

1. reactor-core :-

It contains set of implementation classes for reactive stream specification.

2. reactor-test :-

It contains set of classes used for testing the reactive based microservices.

3. reactor-netty :-

It contains set of classes supported to Http or Https for handling non blocking request model and it is also known as a server.

Docker Bridge Network :

It is a internal network used for allowing multiple dockers containers to communicate with each other.

Steps to create a bridge network :-

1. docker network create <network-name>
2. docker network ls :- To see all the list of networks created
3. docker container run --network <name> --name <container-name>
   -p post1 : post2 -d <image-name> :- The above command starts the container for the given image and binds with a given network.

Flux and Mono

Flux and Mono are the container specific classes which implements Publishes interface.

Either Flux or Mono, dataStream is started only once the subscribes completes the subscription and then followed with the request() for volume of data chunks.

Reactive streams are of two types....

1. cold stream
2. Hot Stream

cold stream is nothing but a fixed set of data or Static data.

Hot stream is nothing but a continuous flow of data or infinite data.

Some of the common methods in both Flux & Mono classes are as follows

1. Just():-

It creates either Flux or Mono type object to transfer the stream of data to the subscriber.

2. filter() :-

It performs the condition checking from the reactive stream of each data chunk and whenever the given condition is true, content is passed to the next series of method chain.

3. map() :-

It maps each data chunk with a given value.

4. flatmap() :-

It maps collection of collections

5. concatMap() :-

It joins a given stream into existing stream.

6. concatWith() :-

It merges with any given type of content.

7. thenReturn() :-

It returns a given value in the Mono type.

Custom Spring Boot Starter :-

Spring Boot is a opinionated framework. i.e instead of using predefined starters, we can use Custom Starters also, as a part of Micro Services development.

Using custom starters we can reuse the functionality across multiple microservices and also we can build wrapper framework on the top of the spring boot for adding additional integration with the services to the part of Organization.

There are two ways to create a Spring boot Starter.

1. we can create a starters as a individual project.
2. we can create a starter as a module in our project.

Steps to create a custom spring boot Starter :-

1. create a configuration class which contains bean methods which are annotated either with custom conditional annotation or pre defined conditional annotation provided by the spring boot.

3. Declare fully qualified name of our configuration class into spring.factories file and it must be created inside "META-INF" folder to be created inside "src/main/resources".

Q. How AutoConfiguration Process works internally in Spring boot :-

1. Initially in the main method call i.e once Spring container is instantiated, "@EnableAutoConfiguration" annotation request "SpringBootConfigLoader" to search for all the given autoConfiguration classes presented in the classpath i.e looks for spring.factories file for each jar file.

2. Finally all the selected autoConfiguration classes are loaded one after another into ApplicationContext.

Micro Services Transaction Management

A Transaction is nothing but a operation (CRUD) which was performed on the data base.

In the monolithic architecture the entire software application communicates with a Single Database and hence we can manage the transactions with the help of Transaction Management API or any third party library like Hibernate.

The two phase model is commonly implemented in the large scale application (enterprise like IRCTC, make my trip).

Saga Architecture

It is introduced in 1987, from Princeton University to manage complex distributed transactions in the sequence pattern (i.e. group of steps).

(1) Here each step is nothing but a Local Transaction.

(2) As we know every transaction must follow ACIDC Atomicity, consistency, isolation & durability) properties to provide data consistency across concurrent transactions.

(3) ACID properties are applicable whenever if transactions are managed or bounded to the specific database. This model is well accepted in monolithic architecture model.

(4) This ACID properties we can implement with a help of specific library provided by the thirdparties like JTA (JAVA transaction api) or spring declarative transaction management.

In this microservices ~~arrive~~ environment, each microservice will have its own database and so transaction ~~cannot~~ cannot be implemented with ACID properties because it is a distributed transaction.

we can implement this transaction by using "saga Architecture" model.

In saga architecture, there are two types of models.

1. choreography based
2. Orchestration based

Saga architecture is base on message driven model.

Message driven model is nothing but allowing multiple software applications to communicate with each other in synchronous or asynchronous way.

Using message driven model we can implement loose coupling between multiple software applications.

Generally, microservices are developed either base on RPC (Remote procedure call) or Message driven.

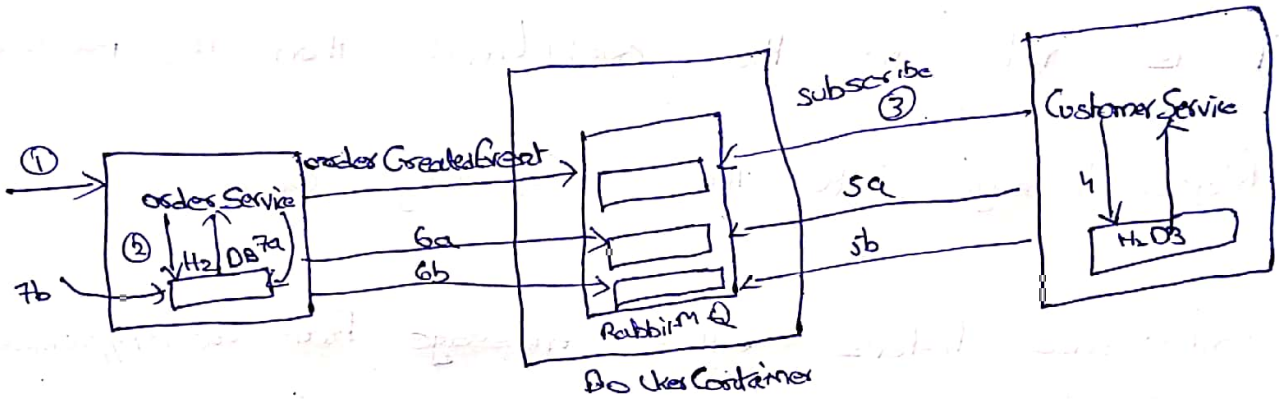
RPC based is good for communicating with external services or allowing external application to communicate with our services.

Message driven is good for inter process microservice communication for enabling transaction management via saga Architecture.

Choreography based model:-

In this model, publishing the message or subscribing the message to the message broker need to implement in each microservice.

This model is good for a small microservices based project (5 - to 10 microservices). If is better to jump into Archestrator model).



Architecture Flow:-

1. whenever user submits request to the order Service for placing a new order, Order Service performs the transaction process followed.

2. It insert a new record into the h2 db with initial state as "Processing".

3. If submits a message into the "OrdersQueue" as orders created.

4. CustomerService listens the message from the "OrdersQueue" and contacts with its "he database" to verify whether given ordersamount is within the creditLimit or not.

5. If it is in the creditLimit, then it debits the given order amount from the "creditLimit" i.e from the dB and then produces "creditApprovedMessage" into the "creditApprovedQueue"

Note:-

If it is not in the creditLimit, then it produces "CreditRejectedMessage" into the "CreditRejectedQueue".

6. OrderService listens either message from "CreditApprovedQueue" or "CreditRejectedQueue" base on this if decides to update the state of the orders as "Approved" or "Rejected".

Orchestration Model of Saga :-

In this model, we have to centralize saga-coordination logic by preparing a "Orchestrator" class.

Saga-coordination logic is nothing but executing the transactions from multiple microservices into the steps by communicating with message broker using Message driven model.

Depends on the requirement we have to prepare multiple orchestrator classes for managing different saga steps.

Ex:-

1. CreateOrderSaga:- This class is responsible to manage all the steps or transactions related to create order.

2. CancelOrderSaga:- This class is responsible to manage all the steps or transactions related to cancel order.

Link:- Microoservice.io.