Spring Boot - Microservices

Microservice - starts from page 35
No need to read from page 9 to 22

When java was introduced developers used to develop the application using the java J2EE (servlets , jsp etc).
Then different frameworks were created like struts , liferay , spring etc.
Spring was most famous because it provides a lot of functionalities and different modules to work with. They can easily work with different modules and we can create applications easily.
For web apps I can use spring mvc module , for bash processing I can use the spring bash processing module and many more which can easily integrate and use with the spring framework.
In the earlier days spring was configured using xml configuration so at that time developers had to manually add a lot of configuration to work with spring .

To minimise a lot of configuration developers introduced spring boot framework.

spring boot is the abstraction layer in the top of the spring framework.
All the autoconfiguration has been added to the spring framework using spring boot .
So a lot of burden from developers shoulders was put off. And all those functionality and configuration and everything was default provided to the developer.
So that developers can actually focus on business logic.

`Dependency Injection` →

Whenever we talk about spring we talk about dependency injection.
Let's have a student class with few fields and with a subject class , we have another class named subject with few fields.
Now the student class depends on the subject class so whenever I need a student object I require the subject object as well .So the student totally depends on the subject class .
Everything is done by developers, which is not ideal because there are a lot of dependencies available and there might be a chance that users have created unnecessary objects as well and garbage collection is not proper and there might be a lot of scenarios.

So spring suggests the `inversion of control(IOC)` which gives the control to spring framework itself rather than taking the control to your own.so IOC is simply giving control back to the framework.
To implement IOC we use dependency injection pattern that allows us to use the ioc where all the dependencies are being handled by spring framework itself.
Whenever we defined all the classes in the spring framework we defined all the classes as beans.
And whenever the application starts it creates a factory of all the beans.
It has a bean factory and all the beans stored in the container.
Now if any of the beans is interlinked then spring framework will take responsibility to inject wherever it is necessary based on the different scopes defined in the spring framework.
Rather than creating objects by developers , spring takes care of all of that dependency injection.

Spring initializr - tool to create spring boot project.

Maven and Gradle - project management tools.

Gradle → it is based on developing domain-specific language projects.
→It uses a Groovy-based Domain-specific language(DSL) for creating project structure.

Maven → it is based on developing pure Java language-based software.
→ It uses Extensible Markup Language(XML) for creating project structure.

For each and every project we need to defined 3 things

1) Group
2) Artifact
3) Version information

With these 3 details projects would be unique in the entire repository.
Ideally a group name given as the domain name of the company in the reverse.
Artifact means what name would be of your project.

`Packaging types`

1) **Jar** - comes with an inbuilt server and runs on the user's local machine , we directly run the project on the local machine.
2) **War** - manually deploy application on the external application server

- **.jar files:** The .jar files **contain libraries, resources and accessories files** like property files.

- **.war files:** The war file **contains the web application** that can be deployed on any servlet/jsp container. The .war file **contains jsp, html, javascript** and other files necessary for the development of web applications.

POM(project object model).XML file is the maven configuration file for the project.
For web application - Spring Web dependency
IDE - intellij idea.

Default tomcat server port : 8080

Application.yml file for config

`Annotation` → Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide **supplemental** information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

All the config in spring boot we will do mostly using annotation , we define the annotation and based on that spring boot will react to it and functionality added to those classes.

`@Controller` - class should behave as a controller and serve a different request .
`@ResponseBody`- give a response as a responsebody and pass the data as a body as well. It won't create a mvc pattern where we pass jsp and index files.
Class behaves in such a way that it should return data in responsebody it should not go to search for any views configured.
`@RestController - controller+ResponseBody`
`@RequestMapping("/)` - all requests with slash(/) will come here.it maps HTTP requests to handler methods of MVC and REST controllers.
You can have multiple request mappings for a method. For that add one @RequestMapping annotation with a list of values.

```
@RequestMapping(value = {
        "",
        "/page",
        "page*",
        "view/*,**/msg"
})
```

`Spring Boot starter project` - spring boot bundles all the required dependencies into one bundle and it's called starter. And if you want to add a whole bundle in a different project then you just need to add only one starter bundle; it will add all dependencies within the bundle.

For eg . `spring-boot-starter-web` bundle contains many dependencies like -

`Spring-boot-starter-json`
`spring-boot-starter-tomcat`
`spring-web`
`spring-webmvc`
`Spring-boot-starter`

Working of spring boot - whenever you are creating a spring boot project , it has the spring.factory added to the meta-inf folder and all the configuration and required jar files are mentioned there . whenever you add any property or configuration and it matches with the spring factory it tries to add these config in it and different config takes place internally.

```
External library →
  Maven:org.springframework.boot.spring-boot-autoconfigure:2.7.3
→ META_INF  →spring.factories
```

`@SpringBootApplication`
`@Configuration`

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

Embed server →whenever we create an application and that application will be packaged into a war file and this war file deployed to the server .
Servers - tomcat ,weblogic ,jboss etc.

In spring boot - application and server all together contained in the single jar file and run the jar file , no need to deploy to the external server.
Three types of embed server - tomcat(by default choice) , jetty , undertow .

To use other than tomcat , we first need to remove tomcat and then add jetty etc.
We know that spring-boot-starter -web dependency contains default tomcat server so first we need to exclude using exclusion tag.

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

Now add jetty server , for that add dependency of jetty

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

To add automatic - in intellij idea we have maven helper install the same and use it .

`Spring boot Actuator →` allow us to monitor spring boot applications.

1) **Add dependency** -

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

2) go to browser and hit the url - localhost:port/actuator
   By default only health endpoints are exposed because of security reasons.

`spring boot devtools→` Live reloading functionality

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
  <optional>true</optional>
</dependency>
```

`web services` - are social media apps web services ?

Web services are any services which I can access over the network.
Any app should access web services.

If any APP-1 can access APP-2 over the web is known as web services.

If the user is accessing any app is not known as web services.

Analytics of social media application is used to analyse the social media that is known as web service but if users use the social media that is not called as web services.
Applications use HTTP protocol to access web services.

URL vs URI – The URI can represent both the URL and the URN of a resource, simultaneously, while URL can only specify the address of the resource on the internet.

Web services types -

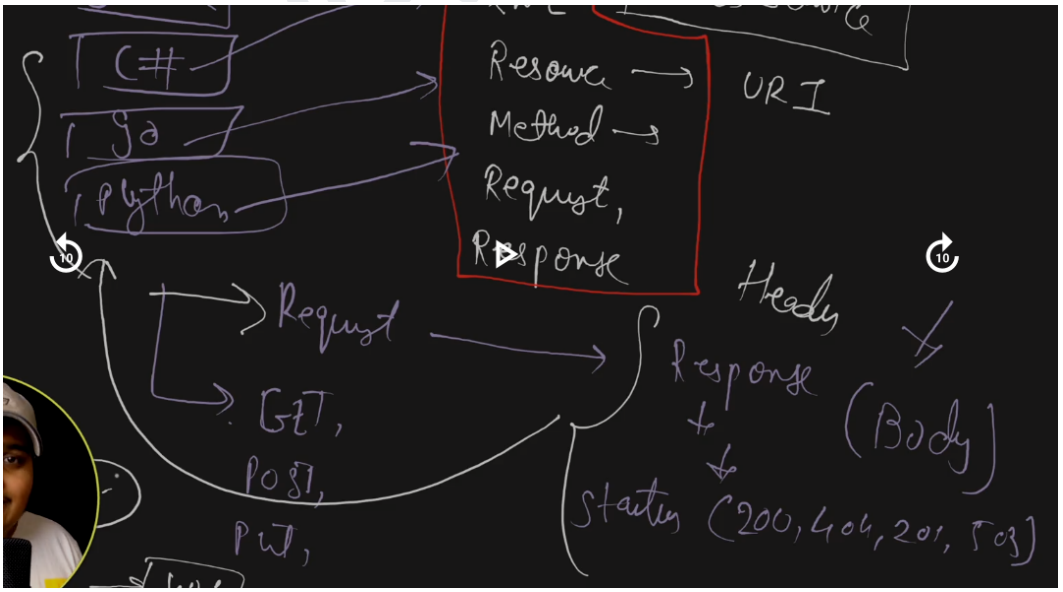
1) **REST**
2) **SOAP**

`How Web Services works` - application 1 will send the request to application 2 to get some data and application 2 will process the request and send the response back.
If app 2 is exposing some services then app 1 can use those services by calling the uri.
All these are handled over HTTP protocol.
For any web services , there are some things that needs to be defined

1) Request data - String , xml , json
2) Data type - (XML ,JSON)
3) Response data
4) Resource (URI) - **localhost:8080**/user (**url**/uri)
5) HTTP Method (for REST) - GET , POST , PUT , DELETE etc.

REST != HTTP

`What is REST` **(Representational state transfer) →**



****

In simple words, in the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs).

The resources are acted upon by using a set of simple, well-defined operations. Also, the resources have to be decoupled from their representation so that clients can access the content in various formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.
The clients and servers exchange representations of resources by using a standardised interface and protocol. Typically HTTP is the most used protocol, but REST does not mandate it.

Metadata about the resource is made available and used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

And most importantly, every interaction with the server must be stateless.

All these principles help RESTful applications to be simple, lightweight, and fast.

REST principles are not affected by SPA design.

I will suggest using all lowercase seperated with hyphens.

I suggest never using camel case notation. You should use all lowercase separated with hyphens. It helps in SEO.

Media Types have no relation to the resource methods GET/PUT/POST/DELETE/

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

For this reason, the client application must entirely keep the session state.

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behaviour.

For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of

features delivered to the client in the form of code, and the client only needs to execute the code.

If an API is following 6 guiding Principles of REST then it's a RESTful API
1)

The six guiding principles or constraints of the RESTful architecture are:

1.1. Uniform Interface

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

Multiple architectural constraints help in obtaining a uniform interface and guiding the behaviour of components.

The following four constraints can achieve a uniform REST interface:

- Identification of resources – The interface must uniquely identify each resource involved in the interaction between the client and the server.
- Manipulation of resources through representations – The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
- Self-descriptive messages – Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.
- Hypermedia as the engine of application state – The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

1.2. Client-Server

The client-server design pattern enforces the separation of concerns, which helps the client and the server components evolve independently.

By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

1.3. Stateless

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

For this reason, the client application must entirely keep the session state.

1.4. Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

1.5. Layered System

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.

For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

1.6. Code on Demand (*Optional*)

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

2. What is a Resource?

The key abstraction of information in REST is a resource. Any information that we can name can be a resource. For example, a REST resource can be a document or image, a temporal service, a collection of other resources, or a non-virtual object (e.g., a person).

The state of the resource, at any particular time, is known as the resource representation.

The resource representations are consist of:

- the data
- the metadata describing the data
- and the hypermedia links that can help the clients in transition to the next desired state.

A REST API consists of an assembly of interlinked resources. This set of resources is known as the REST API's *resource model*.

2.1. Resource Identifiers

REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

2.2. Hypermedia

The data format of a representation is known as a media type. The media type identifies a specification that defines how a representation is to be processed.

A RESTful API looks like *hypertext*. Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure).

Hypertext (or hypermedia) means the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions.

Remember that hypertext does not need to be HTML (or XML or JSON) on a browser. Machines can follow links when they understand the data format and relationship types.

— Roy Fielding

2.3. Self-Descriptive

Further, resource representations shall be self-descriptive: the client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource.

So in practice, we will create lots of custom media types – usually one media type associated with one resource.

Every media type defines a default processing model. For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

> Media Types have no relation to the resource methods GET/PUT/POST/DELETE/… other than the fact that some media type elements will define a process model that goes like “anchor elements with an `href` attribute create a hypertext link that, when selected, invokes a retrieval request (GET) on the URI corresponding to the `CDATA`-encoded `href` attribute.”

3. Resource Methods

Another important thing associated with REST is resource methods. These resource methods are used to perform the desired transition between two states of any resource.

A large number of people wrongly relate resource methods to HTTP methods (i.e., GET/PUT/POST/DELETE). Roy Fielding has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that it should be a uniform interface.

For example, if we decide that the application APIs will use HTTP POST for updating a resource – rather than most people recommend HTTP PUT – it’s all right. Still, the application interface will be RESTful.

Ideally, everything needed to transition the resource state shall be part of the resource representation – including all the supported methods and what form they will leave the representation.

We should enter a REST API with no prior knowledge beyond the initial URI (a bookmark) and a set of standardized media types appropriate for the intended audience (i.e., expected to be understood by any client that might use the API).

From that point on, all application state transitions must be driven by the client selection of server-provided choices present in the received representations or implied by the user's manipulation of those representations.

The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on the fly (e.g., *code-on-demand*). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

4. REST and HTTP are Not the Same

Many people prefer to compare HTTP with REST. REST and HTTP are not the same.

REST != HTTP

Though REST also intends to make the web (internet) more streamlined and standard, Roy Fielding advocates using REST principles more strictly. And that's from where people try to start comparing REST with the web.

Roy Fielding, in his dissertation, has nowhere mentioned any implementation direction – including any protocol preference or even HTTP.

Till the time, we are honoring the six guiding principles of REST, which we can call our interface – RESTful.

5. Summary

In simple words, in the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs).

The resources are acted upon by using a set of simple, well-defined operations. Also, the resources have to be decoupled from their representation so that clients can access the content in various formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.

The clients and servers exchange representations of resources by using a standardized interface and protocol. Typically HTTP is the most used protocol, but REST does not mandate it.

Metadata about the resource is made available and used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

And most importantly, every interaction with the server must be stateless.

All these principles help RESTful applications to be simple, lightweight, and fast.

The six guiding principles or constraints of the RESTful architecture are:

1.1. Uniform Interface
By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

Multiple architectural constraints help in obtaining a uniform interface and guiding the behavior of components.

The following four constraints can achieve a uniform REST interface:

Identification of resources – The interface must uniquely identify each resource involved in the interaction between the client and the server.
Manipulation of resources through representations – The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
Self-descriptive messages – Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.
Hypermedia as the engine of application state – The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

1.2. Client-Server

The client-server design pattern enforces the separation of concerns, which helps the client and the server components evolve independently.

By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

1.3. Stateless

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

For this reason, the client application must entirely keep the session state.

1.4. Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

1.5. Layered System
The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.

For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

1.6. Code on Demand (Optional)
REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

2. What is a Resource?
The key abstraction of information in REST is a resource. Any information that we can name can be a resource. For example, a REST resource can be a document or image, a temporal service, a collection of other resources, or a non-virtual object (e.g., a person).

The state of the resource, at any particular time, is known as the resource representation.

The resource representations are consist of:

the data
the metadata describing the data
and the hypermedia links that can help the clients in transition to the next desired state.
A REST API consists of an assembly of interlinked resources. This set of resources is known as the REST API’s resource model.

2.1. Resource Identifiers
REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

2.2. Hypermedia
The data format of a representation is known as a media type. The media type identifies a specification that defines how a representation is to be processed.

A RESTful API looks like hypertext. Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure).

Hypertext (or hypermedia) means the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions.

Remember that hypertext does not need to be HTML (or XML or JSON) on a browser. Machines can follow links when they understand the data format and relationship types.

— Roy Fielding
2.3. Self-Descriptive
Further, resource representations shall be self-descriptive: the client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource.

So in practice, we will create lots of custom media types – usually one media type associated with one resource.

Every media type defines a default processing model. For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

Media Types have no relation to the resource methods GET/PUT/POST/DELETE/… other than the fact that some media type elements will define a process model that goes like “anchor elements with an href attribute create a hypertext link that, when selected, invokes a

retrieval request (GET) on the URI corresponding to the CDATA-encoded href attribute.”

3. Resource Methods
Another important thing associated with REST is resource methods. These resource methods are used to perform the desired transition between two states of any resource.

A large number of people wrongly relate resource methods to HTTP methods (i.e., GET/PUT/POST/DELETE). Roy Fielding has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that it should be a uniform interface.

For example, if we decide that the application APIs will use HTTP POST for updating a resource – rather than most people recommend HTTP PUT – it’s all right. Still, the application interface will be RESTful.

Ideally, everything needed to transition the resource state shall be part of the resource representation – including all the supported methods and what form they will leave the representation.

We should enter a REST API with no prior knowledge beyond the initial URI (a bookmark) and a set of standardized media types appropriate for the intended audience (i.e., expected to be understood by any client that might use the API).

From that point on, all application state transitions must be driven by the client selection of server-provided choices present in the received representations or implied by the user’s manipulation of those representations.

The transitions may be determined (or limited by) the client’s knowledge of media types and resource communication mechanisms, both of which may be improved on the fly (e.g., code-on-demand). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

4. REST and HTTP are Not the Same
Many people prefer to compare HTTP with REST. REST and HTTP are not the same.

REST != HTTP

Though REST also intends to make the web (internet) more streamlined and standard, Roy Fielding advocates using REST principles more strictly. And that's from where people try to start comparing REST with the web.

Roy Fielding, in his dissertation, has nowhere mentioned any implementation direction – including any protocol preference or even HTTP. Till the time, we are honoring the six guiding principles of REST, which we can call our interface – RESTful.

5. Summary

In simple words, in the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs).

The resources are acted upon by using a set of simple, well-defined operations. Also, the resources have to be decoupled from their representation so that clients can access the content in various formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.

The clients and servers exchange representations of resources by using a standardized interface and protocol. Typically HTTP is the most used protocol, but REST does not mandate it.

Metadata about the resource is made available and used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

And most importantly, every interaction with the server must be stateless.

All these principles help RESTful applications to be simple, lightweight, and fast.

****

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

1. Informational responses (100–199)
2. Successful responses (200–299)
3. Redirection messages (300–399)
4. Client error responses (400–499)

200 - OK

201 - created

400 - bad request

401 - unauthorised

402 - payment required

403 - forbidden

404 - resource not found

405 - method not allowed

500 - internal server error

501 - not implemented

502 - bad gateway

503 - service unavailable

504 - gateway timeout

Note : by Default method (Get Method)
@RequestMapping(“/) - considered as get mapping

@RequestMapping(value = “/” , Method = RequestMethod.Get.POST) - considered as post mapping

@RequestMapping(value = “/” , Method = RequestMethod.Get.POST) is equivalent to @PostMapping(value = “/”)

Path Variable - declared with slash(/)
For eg. `@GetMapping("/{id}/{id2}")`
- It use when there is a particular mandatory field

Whenever there is a mandatory data we need to take then only use path variable

Path - localhost:8080/ , localhost:8080/1 , localhost:8080/chail/singh

```
// allowed
@GetMapping("/{id}/{id2}")
public String pathVarible (@PathVariable String id , @PathVariable
String id2){
    return "pathvariable is : " + id + "and " + id2;
}

//allowed
@GetMapping("/{id}/{id2}")
public String multiplepathVarible (@PathVariable("id2") String
name , @PathVariable("id2") String surname){
    return "pathvariable is : " + name + "and " + surname;
}


//not allowed
@GetMapping("/{id}/{id2}")
public String pathVarible (@PathVariable String name ,
@PathVariable String surname){
    return "pathvariable is : " + name + "and " + surname;
}
```

RequestParam - declared with question mark
For eg .

- Whenever there is a data we need to take but that data or field is not mandatory then only use request param
- Multiple request parameters separated by &

Path - localhost:8080/requestParam?name=chail

```
@GetMapping("/requestParam")
public String requestParam(@RequestParam String name){
    return "name is : " + name;
}
```

Path -localhost:8080/requestParam?name=chail&emailId=chail@gmail.com

```
@GetMapping("/requestParam")
public String requestParam(@RequestParam String name,@RequestParam
String emailID){
    return "name is : " + name + "and email is " + emailID;
}
```

Path -localhost:8080/requestParam?name=chail&email=chail@gmail.com

```
@GetMapping("/requestParam")
public String requestParam(@RequestParam String
name,@RequestParam(name= "email") String emailID){
    return "name is : " + name + "and email is " + emailID;
}
```

Path -localhost:8080/requestParam?name=chail (output - name is chail and email is null)

```
@GetMapping("/requestParam")
public String requestParam(@RequestParam String
name,@RequestParam(name= "email" , required = false) String
emailID){
    return "name is : " + name + "and email is " + emailID;
}
```

Path -localhost:8080/requestParam?name=chail (output - name is chail and email is abc@gmail.com)

```
@GetMapping("/requestParam")
```

```
public String requestParam(@RequestParam String
name,@RequestParam(name= “email” ,defaultValue=”abc@gmail.com”,
required = false) String emailID){
    return "name is : " + name + "and email is " + emailID;
}
```

For service , follow interface pattern because one service may have a lot of implementation for a particular logic

Content Negotiation - currently we are getting data in json format itself but what if we need to get the data in different form so for that reason we need to give all rest api's capabilities to generate data in different formats.

By Default , rest api provides us capability to generate data in JSON and XML format.

First way -

```
@GetMapping(value = "/requestParam",produces =
MediaType.APPLICATION_JSON_VALUE)
public String requestParam(@RequestParam String
name,@RequestParam String emailID){
    return "name is : " + name + "and email is " + emailID;
}
```

Or

```
@GetMapping(value = "/requestParam",produces =
MediaType.APPLICATION_XML_VALUE)
public String requestParam(@RequestParam String
name,@RequestParam String emailID){
    return "name is : " + name + "and email is " + emailID;
}
```

Other ways -

In the first way we need to write in every endpoint which is not feasible for that reason we have different options to handle this type of thing.

1) Using parameters - you can pass parameters in uri and can identify which types of data needs to be generated.

→if uri parameter content xml then generate xml like others.

Step 1 - **make a class webconfig**

```
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import
org.springframework.web.servlet.config.annotation.ContentNegotiati
onConfigurer;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer
;

@Configuration
public class WebConfig implements WebMvcConfigurer {
   @Override
   public void
configureContentNegotiation(ContentNegotiationConfigurer
configurer) {
       configurer.favorParameter(true)
                .parameterName("mediaType")
                .defaultContentType(MediaType.APPLICATION_JSON)
                .mediaType("xml",MediaType.APPLICATION_XML)
                .mediaType("json",MediaType.APPLICATION_JSON);
   }
}
```

Step 2 - **add dependency**

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Step 3 - **postman**

case 1) http://localhost:9090/employees - data default produce in json
case 2) http://localhost:9090/employees?mediaType=xml - data produces in xml
case 3) http://localhost:9090/employees?mediaType=json - data produce in json

2) **Using Http Header** - you can define in the http header as well like what is the content type and based on that api should be able to pass those data and based on that you should be able to generate those requirements.

Data Filtering - JsonIgnore annotation -

```
@JsonIgnore
private String department;
```

→ this property would be ignored and in response we will not get the property which is annotated with jsonignore.

For ignore multiple property define json ignore property on class level

```
@JsonIgnoreProperties({"lastName","email"})
public class Employee {

    private Long id;
    private String firstName;
    private String lastName;
    private String email;
}
```

API-VERSIONING

For eg.
Twitter gives the api to all different developers and user to consume those api's to do different operations like (tweet etc)
Let's we have one tweet api with property - id ,text,image
They have had this api for a long time now. They want to change it and they want to do it , rather than giving an image as a field they will give a separate api for image .
Now they will give api with fields like id,text,imageId and one api separate for image post.

Now let's say I am using an old api and twitter decided to change the entire implementation. Now all of sudden my application stops working because they change api and the old api is not in active mode.
To avoid this type of scenario we do api versioning.
Now using versioning we have both api active with the version v1 and v2.
Now I can tell users that you have a few days to use this implementation then you have to shift to the new version by updating the application.
After some time the client will decommission the older api till then the user has little time to update the application.

To handle versioning there are different ways

1) Most popular is using resource uri

For first version uri - /v1/tweet
For latest version uri - /v2/tweet
We can see the uri's are different so there is no contradiction between them.

2) Second way using query parameter

?version = v.1.1

3) Third way using http header

Java persistence api(JPA)

Whenever we want to work with a database we have to change our data which is in the object form to store in our database that is in the tabular or table format.
To do such a thing we use **ORM(object relational mapping) frameworks**.
Now our object will be mapped to our table using the orm framework.
Hibernate and ibatis are orm frameworks.
Now they can go ahead and implement their own implementation.
To make sure that everyone uses the same standard and same format JPA comes into picture.
JPA is nothing but a specification.
That means whatever you are implementing or whatever your orm framework would be it should we according to this particular api and according to this particular implementation that's what jpa defines
Now based on this jpa all the different frameworks like hibernate , ibatis etc go ahead and use this jpa specification and try to use their own implementation.
Whenever you use your code there will be a jpa layer besides this layer there will be any framework available.
The Jpa layer will be responsible to call the jdbc api and jdbc drivers to connect with the database and store the data converted into tables .
For spring boot whenever we use jpa ,default hibernate framework is beijing used.if you want to remove then you can exclude as well and use others.
Until recently this jpa was implemented for relational database itself but for spring boot you can implement nosql database as well.
If you are using mongodb(nosql) you can use mongorepository which internally uses jparepository.

Entity vs Model

Entity is nothing but POJO class annotated with @Entity .
For the database layer we create entities to save the data , for view layer or controller layer or to pass the data back we use a model .
Generally models are also called VO(value object) or DTO.

Database configuration - application file

1) H2 Database - in memory database

```
spring:
 h2:
   console:
     enabled: true
 datasource:
   driver-class: org.h2.Driver
   username: root
   url: jdbc:h2:file:~/data/employee_db
   password: root

 jpa:
   hibernate:
   database-platform: org.hibernate.dialect.H2Dialect
   ddl-auto: update
```

2) Mysql - persistence database

```
#database config
spring:
 datasource:
   driver-class: com.cj.jdbc.Driver
   username: root
```

```
    url:
jdbc:mysql://localhost:3306/databasename?serverTimezone=UTC
    password: root

  #jpa config
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
        format_sql: 'true'
    hibernate:
      ddl-auto: update
    show-sql: 'true'
```

3) Mongodb config - NOSQL database

```
#database-configuration
spring:
  data:
    mongodb:
      authentication-database: admin
      username: chail
      password: user
      database: employee
      port: 27017
      host: localhost
```

Handle two implementation with single interface service –

1) Service - interface

```
import org.springframework.stereotype.Service;


public interface EmployeeService {
}
```

2) EmployeeV1Impl

```
@Service
public class EmployeeV1Impl implements EmployeeService {
}
```

3) EmployeeV2Impl

```
@Service
public class EmployeeV2Impl implements EmployeeService {
}
```

Here we have two implementations with a single interface to handle such scenarios you can use qualifier annotation to qualify which particular class you have to use.

Case 1)

```
@RestController
public class EmployeeV1Controller {

    @Qualifier("employeeV1Impl")
    @Autowired
    private EmployeeService employeeService;
}
```

In this case this controller use the business logic which is written in employeeV1Impl

Case 2)

```
@RestController
public class EmployeeV2ontroller {
```

```
    @Qualifier("employeeV2Impl")
    @Autowired
    private EmployeeService employeeService;
}
```

And in this case this controller use the business logic which is written in employeeV2Impl

BeanUtils - copyproperty

To copy value of all parameters from one entity to another class we can use copyproperty

BeanUtils.copyProperties(source,destination)

```
EmplyeeEntity entity = new EmployeeEntity();
BeanUtils.copyProperties(employee,entity)
```

Set random id

```
employee.setEmployeeId(UUID.randomUUID().toString())
```

employee.setEmployeeId(UUID.randomUUID().toString())

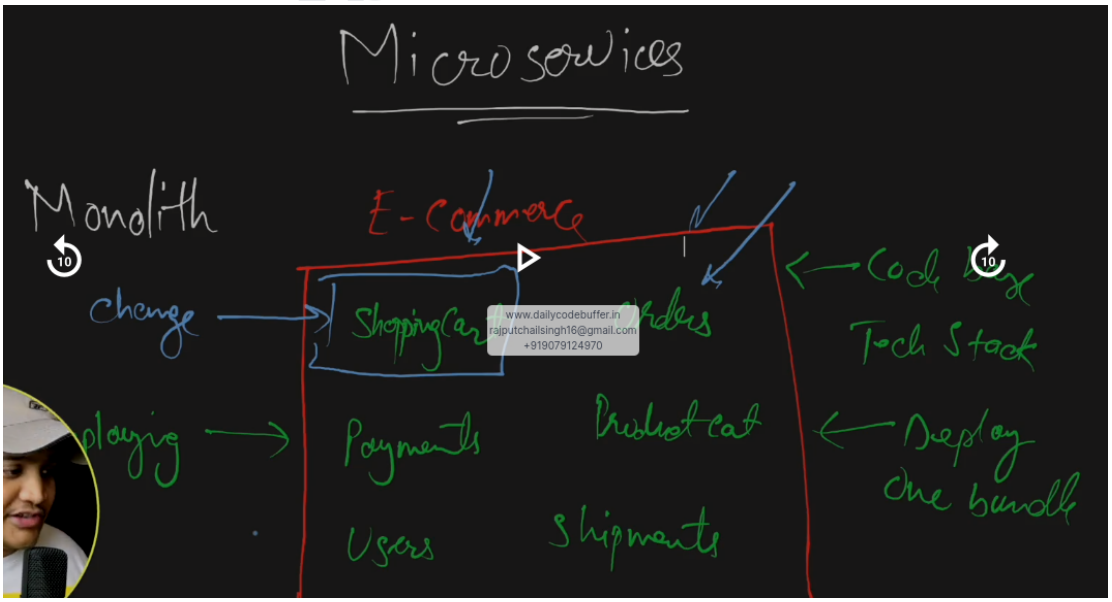
What is Microservices

Earlier - **monolithic architecture**
All different modules within the one architecture and one code base itself.
All modules are built using one tech stack. They are tightly coupled.
For eg.if you build using java then every module has to be in java

Everything deployed using one bundle.

If you want to deploy one module , the entire application will deploy and if you want to do some change in one module , we need to down the entire application and deploy the entire application .
It will also take more time to deploy because for one module we need to take the entire application.
It also adds a complexity layer for the developer purposes because we need to inform each developer to know them to deploy or redeploy.
There are many scaling problems as well , in festival season there are many orders so if you want to scale the order module then you need to scale the entire application as well.

Everything in one repository and for this repository there is CI/CD pipeline where everything deployed all together though it's time consuming but it is simple.

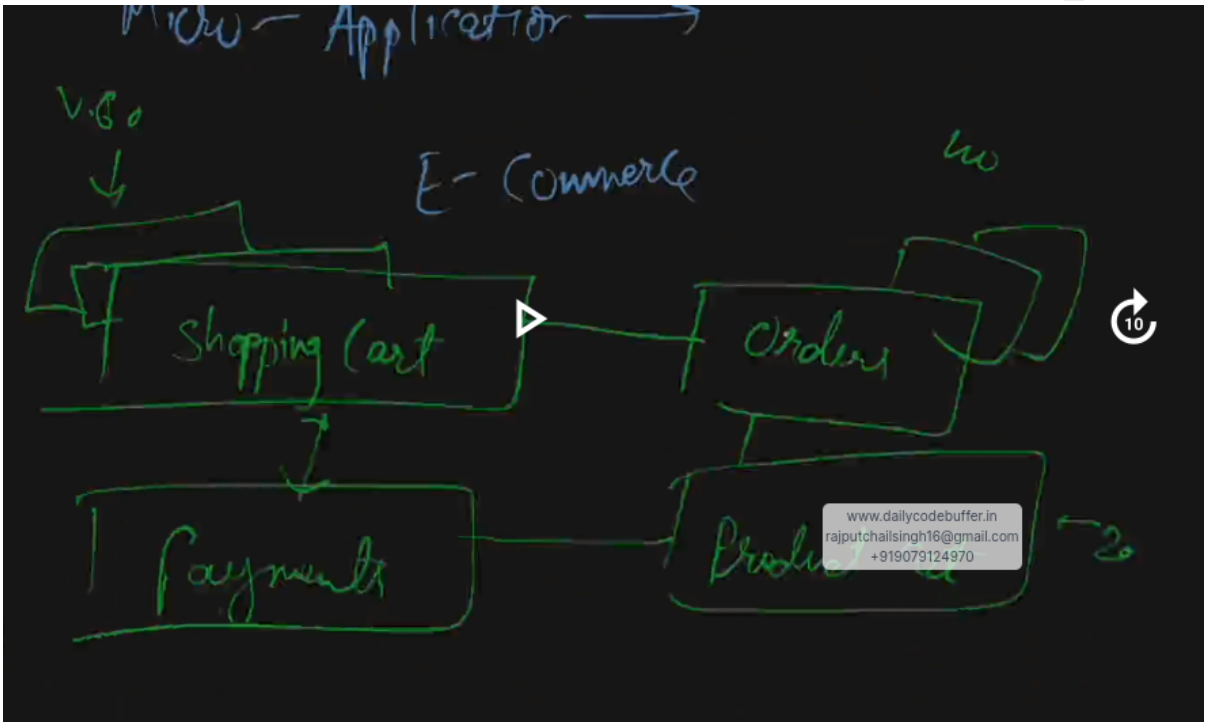


To overcome such scenario , MICROSERVICES comes into picture

—----------

Microservices are nothing but micro-application divided into small parts .

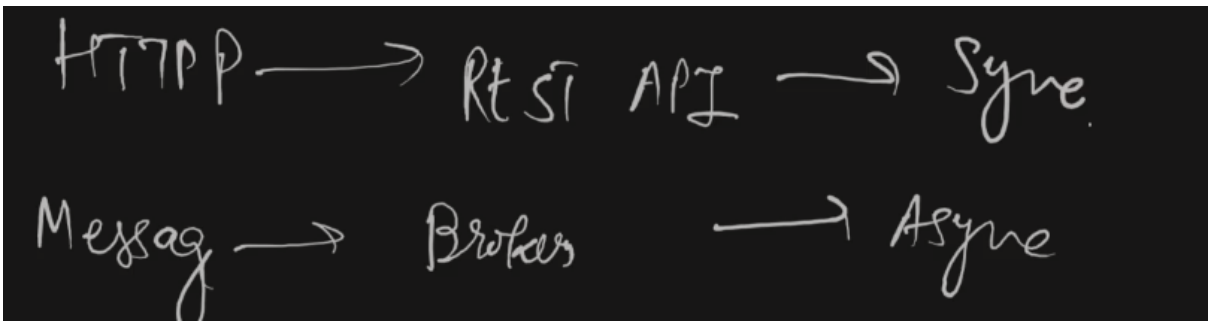
Every module is separate and connected with a single protocol.
Now we can scale,change and do many things without affecting other modules.



All are loosely coupled , they are independent to each other.
All modules can be built using different tech stacks. Or we can say we can use polygon architecture.

All microservices connected using a single protocol .
Most useful and popular protocol is HTTP PROTOCOL and REST-API are created and all api's are exposed and each microservices are able to call the other microservices using api's . This connection is synchronous.
Other ways is to connect them is messaging system (broker) they make async connections

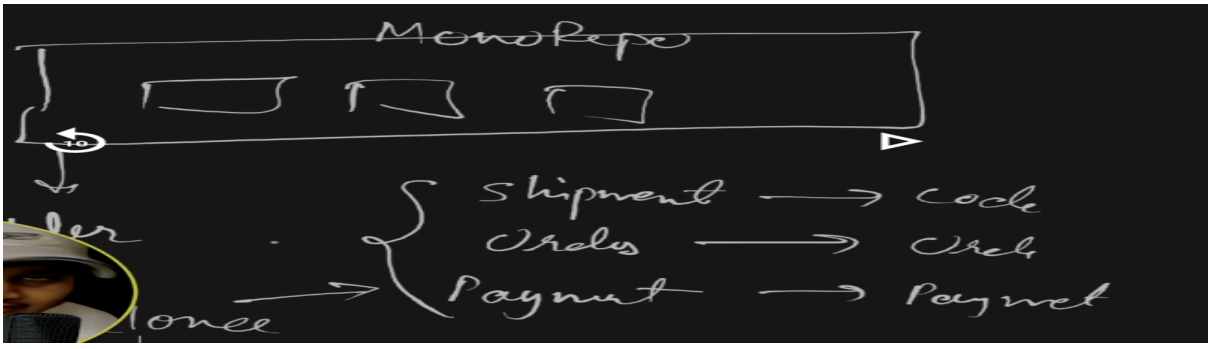
We can create one produce messaging service and other is to consume messaging service and based on that they performed accordingly



Few disadvantage of microservices- all application is been de-structured into the different microservices , now to maintain all those microservices And to work with all those microservices and to debug an issue and trace any of the bugs is very difficult because all those are different microservices and calling each other using any of the other protocols and to trace each and everything is very difficult.
To handle all error scenarios in which microservices fail we have to have many monitoring tools involved and we have to make sure all microservices are monitored correctly and whatever issue comes out we should be figured out asap.

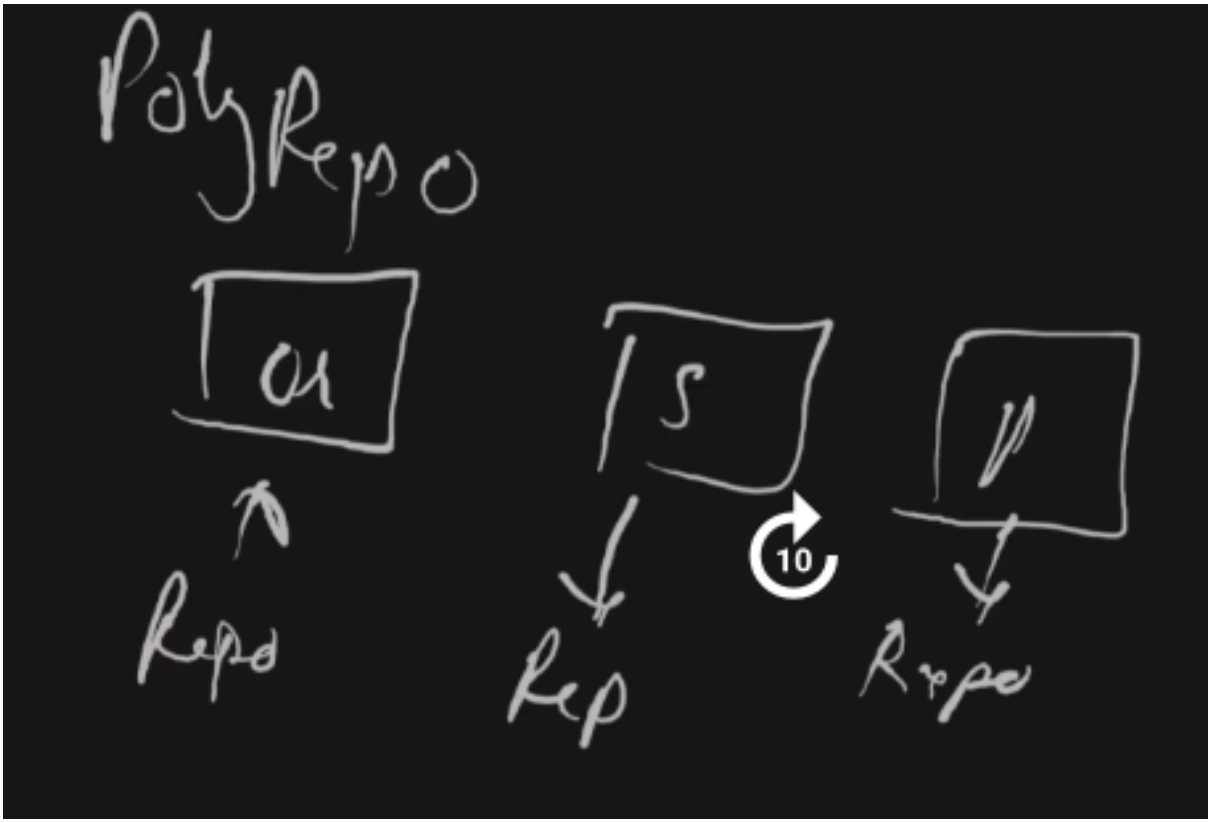
Microservice can be maintain using two types of repository

MonoRepo - all different services are different all together but everything maintain within one repository
Adv - maintaining everything in a folder manner,so clone one repo which contains all these folders.
disAdv - handle CI/CD pipeline . we have to create CI/CD for all these particular services within one repository , there are tools who doesn't allowed multiple CI/CD pipeline for single repository



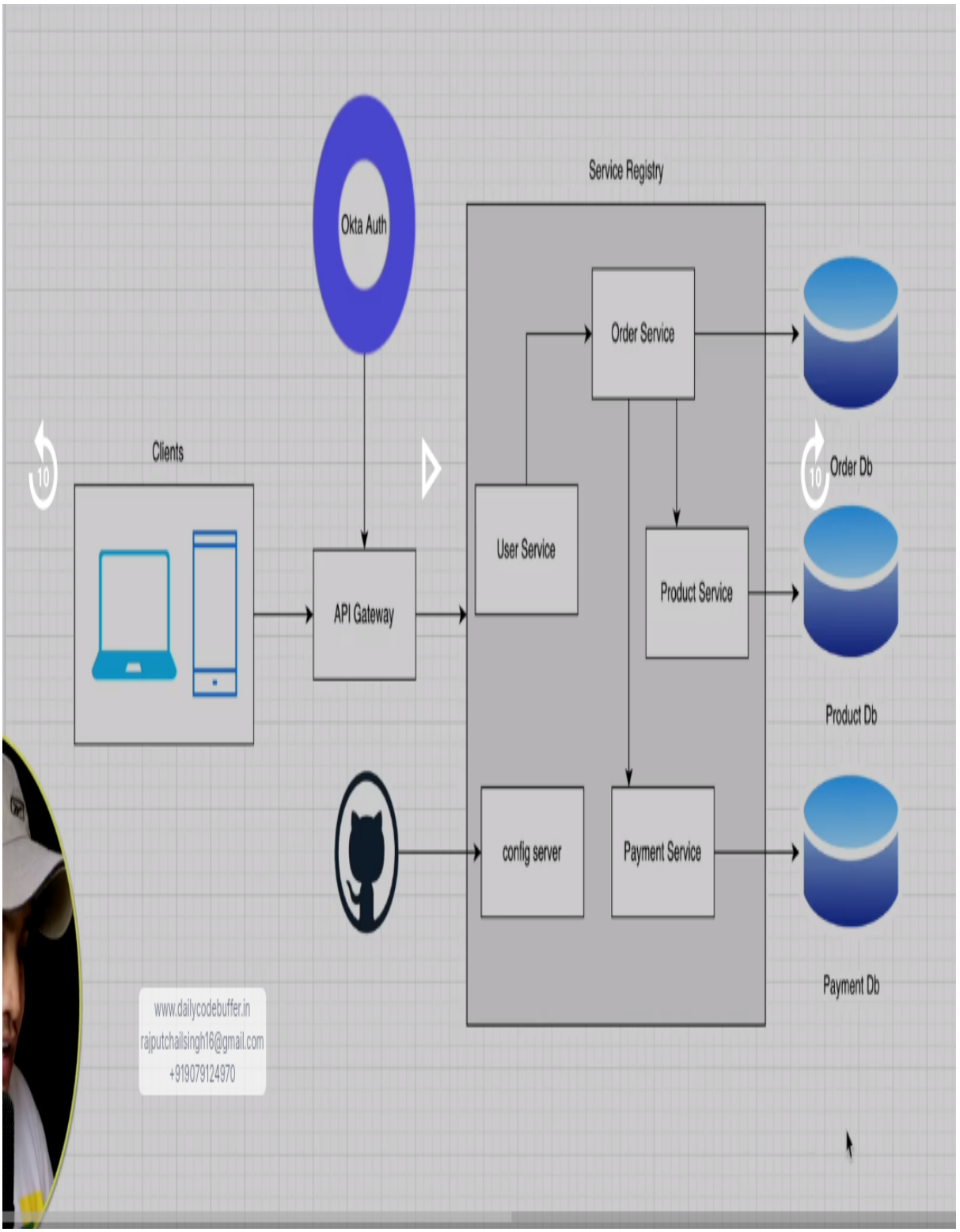
PolyRepo - for each and every service there are different repositories .

Adv - you can easily create CI/CD pipelines for each microservices and can work differently.
disAdv - you have to clone all repositories separately and maintain code .



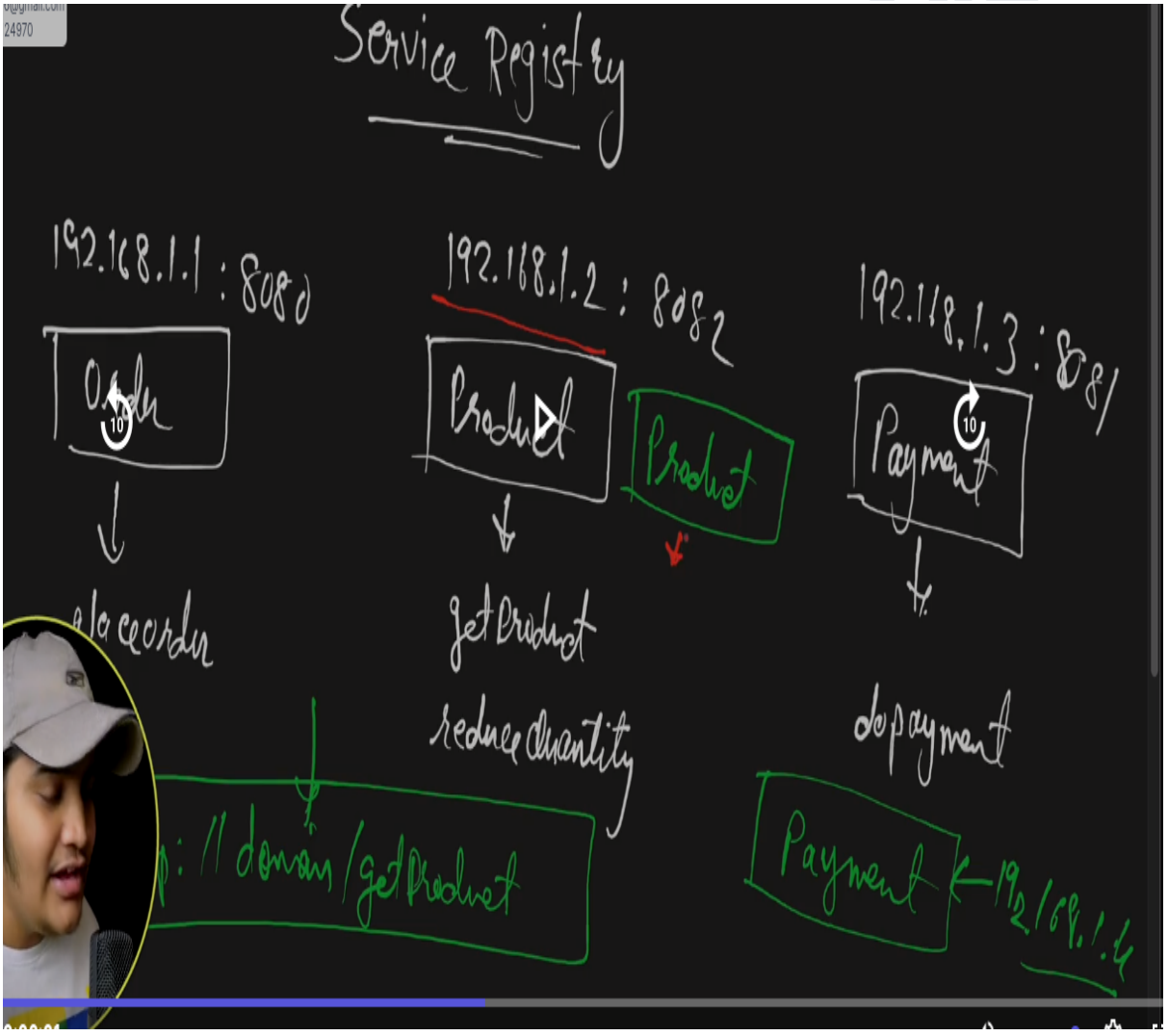
Project Architecture

Per service database design pattern



What is Service - Registry

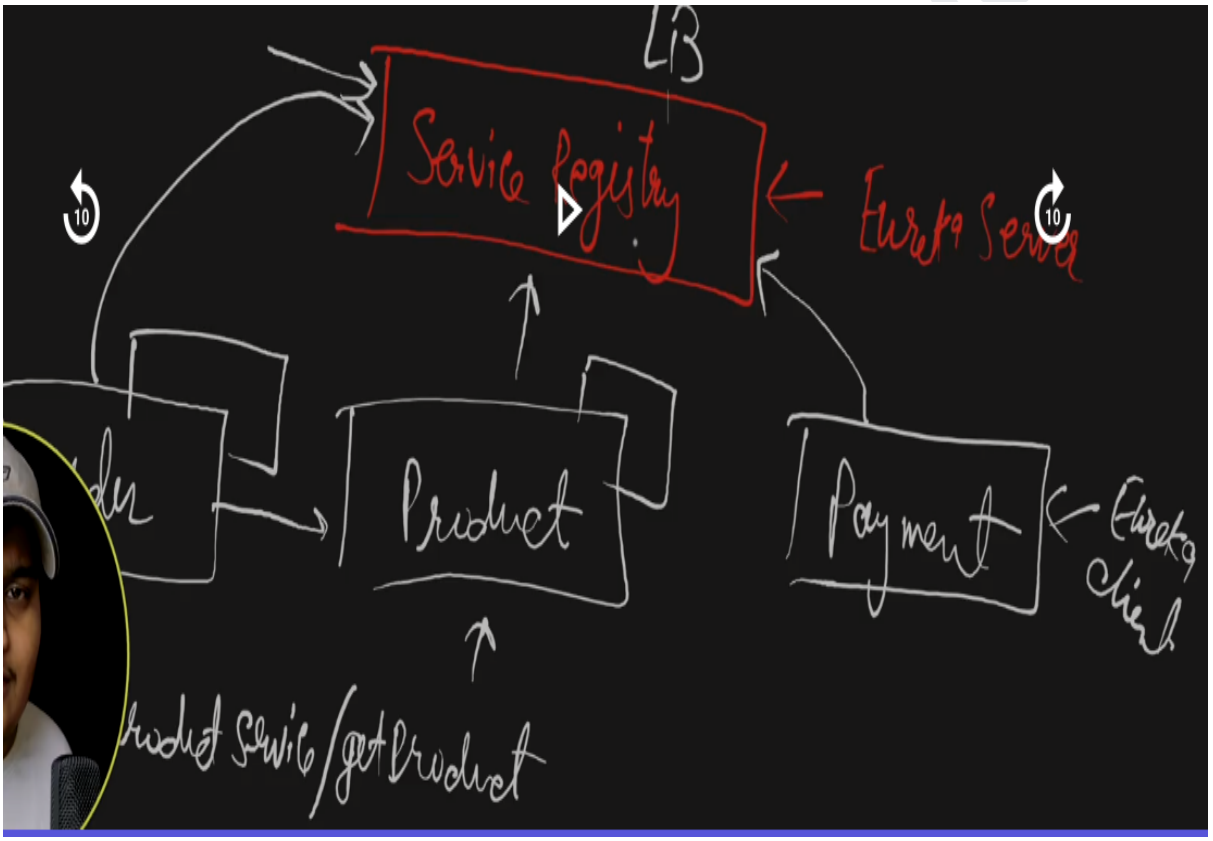
There might be a multiple services deployed on multiple places and there might be multiple ip addresses that we need to call ideally whenever we need to call product service using http we call method using http://domain /getproduct
Now domain can be multiple and different so how we will be able to configure those multiple things for that there has to be way where we are not worried where they are deployed and what ip address they have we only should be focused about i want details from particular services for that reason and also to get availability of different services , service-registry pattern is used .



We will create a service registry and we will implement a service registry using eureka server ,and also implement different services and make them

eureka clients . all the services implemented within eureka client and main service-registry implemented using eureka server.
All services try to connect with the service registry . The service registry knows everything about all services.
Service registry will allow us each and every endpoint using service-name / application-name itself.
So whenever i want to call any service i do not worry about the ip address , port and more information , i can just call by it’s name only.

service registry get the data using load balancer to us all those information



E-COMMERCE APPLICATION - MICROSERVICES

1) **Product-Service(client)**
2) **Order-Service(client)**
3) **Payment-Service(client)**
4) **Service-Registry(sever)**
5) **Config-server(client)**
6) **Api-Gateway(client)**

Service-Registry →

1) Dependency required - **spring cloud , eureka server**
2) Annotate main class with - `@EnableEurekaServer`

Main Class -

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {

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

Yaml -

```
server:
 port: 8761
#default port - 8761
#config for eureka server
eureka:
 instance:
   hostname: localhost
 client:
   register-with-eureka: false
   fetch-registry: false
```

Run Project and go to localhost:8761

Note - Now our server is ready to connect for that we have to do some config in our client

Register every Client to the service-registry

```
eureka:
 instance:
   prefer-ip-address: true
 client:
   fetch-registry: true
   register-with-eureka: true
   service-url:
     defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

Config-Server→handle repetitive code

1) Dependency Required - config-server , Eureka Discovery client
2) Annotate Main class with - `@EnableConfigServer`
3) `Yaml config of config-server`

```
server:
 port: 9296

spring:
 application:
   name: CONFIG-SERVER
 cloud:
   config:
     server:
       git:
         url: url of github reppo
         clone-on-start: true


eureka:
 instance:
   prefer-ip-address: true
 client:
   fetch-registry: true
   register-with-eureka: true
   service-url:
     defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

As we can see to register every client we need to do eureka config which is repetitive so we will go to config server and write repetitive code in yaml of config-server project and upload on github we will do config server config in every client so that every client service can pull repetitive configuration from github itself.

```
   4) To make capable to every client to pull eureka config we
      need to add this config in every client
spring:
 config:
   import: configserver:http://localhost:9296
   //url of config-server
```

Common yml config for all client →

```
server:
 port: port-number

#database config
spring:
 datasource:
   driver-class: com.cj.jdbc.Driver
   username: root
   url:
jdbc:mysql://${live_host:localhost}:3306/database-name?serverTime
zone=UTC
   password: user

 #jpa config
 jpa:
   properties:
     hibernate:
       dialect: org.hibernate.dialect.MySQL8Dialect
       format_sql: true
   hibernate:
     ddl-auto: update
   show-sql: true
```

```
eureka:
 instance:
   prefer-ip-address: true
 client:
   fetch-registry: true
   register-with-eureka: true
   service-url:
     defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

Product-Service →

Main class -

```
@SpringBootApplication
public class ProductServiceApplication {

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

}
```

Yaml config -

```
server:
 port: 9090

#database config
spring:
 datasource:
   driver-class: com.cj.jdbc.Driver
   username: root
   url:
jdbc:mysql://${live_host:localhost}:3306/product?serverTimezone=UT
C
```

```
    password: user

  #jpa config
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
        format_sql: true
    hibernate:
      ddl-auto: update
    show-sql: true

  application:
    name: PRODUCT-SERVICE

eureka:
  instance:
    prefer-ip-address: true
  client:
    fetch-registry: true
    register-with-eureka: true
    service-url:
      defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

Entity-product

```
package product.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class Product {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "product_id")
    private Long productId;
    @Column(name = "product_name")
    private String productName;
    @Column(name = "price")
    private double price;
    @Column(name = "quantity")
    private long quantity;
}
```

Model-product request

```
@Data
public class ProductRequest {

    private String productName;
    private double price;
    private long quantity;

}
```

Model-product response

```
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ProductResponse {

    private Long productId;
    private String productName;
    private double price;
    private long quantity;

}
```

Product-controller

```java
package product.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import product.entity.Product;
import product.model.ProductRequest;
import product.model.ProductResponse;
import product.service.ProductService;

@RestController
@RequestMapping("api/product")
public class ProductController {

    @Autowired
    private ProductService productService;

    @PostMapping("/")
    public ResponseEntity<Long> addProduct(@RequestBody
ProductRequest productRequest){
        Long productId = productService.addProduct(productRequest);
        return new ResponseEntity<>(productId, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductResponse>
getProductById(@PathVariable("id") Long productId){
        ProductResponse productResponse =
productService.getProductById(productId);
        return new ResponseEntity<>(productResponse,HttpStatus.OK);
    }


}
```

Product-service

```java
package product.service;

import product.model.ProductRequest;
import product.model.ProductResponse;

public interface ProductService {
```

```
    Long addProduct(ProductRequest productRequest);

    ProductResponse getProductById(Long productId);
}
```

Product-service-impl

```
@Service
@Log4j2
public class ProductServiceImpl implements ProductService{

    @Autowired
    private ProductRepository productRepository;
    @Override
    public Long addProduct(ProductRequest productRequest) {
        log.info( "adding product");
        Product product = Product.builder()
                .productName(productRequest.getProductName())
                .quantity(productRequest.getQuantity())
                .price(productRequest.getPrice())
                .build();

        productRepository.save(product);
        log.info("product added");
        return product.getProductId();
    }

    @Override
    public ProductResponse getProductById(Long productId) {
        log.info("get product of product id :" + productId);
        Product product = productRepository.findById(productId)
                .orElseThrow(() -> new
ProductCustomException("Product Not Found","NOT_FOUND"));

        ProductResponse productResponse = new ProductResponse();
        BeanUtils.copyProperties(product,productResponse);
        return productResponse;
    }
}
```

Product repository

```
@Repository
public interface ProductRepository extends
JpaRepository<Product,Long> {
}
```

Custom-exception

```
@Data
public class ProductCustomException extends  RuntimeException{
    private String statusCode;

    public ProductCustomException(String message, String
statusCode) {
        super(message);
        this.statusCode = statusCode;
    }
}
```

CustomHandler

```
@ControllerAdvice
public class ExceptionHandler extends
ResponseEntityExceptionHandler {


@org.springframework.web.bind.annotation.ExceptionHandler(Product
CustomException.class)
    public ResponseEntity<ExceptionResponse>
handleProductServiceException(ProductCustomException exception){

        return new ResponseEntity<>(new
ExceptionResponse().builder()
                .message(exception.getMessage())
                .statusCode(exception.getStatusCode())
                .build(), HttpStatus.NOT_FOUND);
    }

}
```

Product -custom-exception

```
@Data
public class ProductCustomException extends  RuntimeException{
    private String statusCode;

    public ProductCustomException(String message, String
statusCode) {
        super(message);
        this.statusCode = statusCode;
    }
}
```

exceptionResponse

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ExceptionResponse {

    private String message;
    private String statusCode;
}
```

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.7.4</version>
      <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>ProductService</groupId>
  <artifactId>ProductService</artifactId>
```

```
  <version>0.0.1-SNAPSHOT</version>
  <name>ProductService</name>
  <description>Demo project for Spring Boot</description>
  <properties>
      <java.version>17</java.version>
      <spring-cloud.version>2021.0.4</spring-cloud.version>
  </properties>
  <dependencies>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-jpa</artifactId>
      </dependency>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter</artifactId>
      </dependency>

      <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <scope>runtime</scope>
      </dependency>
      <dependency>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
          <optional>true</optional>
      </dependency>
      <dependency>
          <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId
>
      </dependency>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
      </dependency>
  </dependencies>
  <dependencyManagement>
      <dependencies>
          <dependency>
```

```
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
  </build>

</project>
```


Service-Registry (Eureka Server) →

Main class -

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {

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

}
```

Yaml config -

```
server:
 port: 8761


eureka:
 instance:
   hostname: localhost
 client:
   register-with-eureka: false
   fetch-registry: false
```

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-parent</artifactId>
     <version>2.7.4</version>
     <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>Service-Registry</groupId>
  <artifactId>Service-Registry</artifactId>
```

```
  <version>0.0.1-SNAPSHOT</version>
  <name>Service-Registry</name>
  <description>Demo project for Spring Boot</description>
  <properties>
      <java.version>17</java.version>
      <spring-cloud.version>2021.0.4</spring-cloud.version>
  </properties>
  <dependencies>
      <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter</artifactId>
      </dependency>
      <dependency>
          <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactI
d>
      </dependency>

      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
      </dependency>
  </dependencies>
  <dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${spring-cloud.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
  </dependencyManagement>

  <build>
      <plugins>
          <plugin>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>
      </plugins>
  </build>
```

```
</project>
```


Order-Service →

Main class -

```
@SpringBootApplication
public class OrderServiceApplication {

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

}
```

order entity -

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "order_id")
    private Long orderId;
    @Column(name = "product_id")
    private Long productId;
    @Column(name = "quantity")
    private long quantity;
    @Column(name = "order_date")
    private Instant orderDate;
    @Column(name = "order_status")
    private String orderStatus;
    @Column(name = "total_amount")
    private long amount;
}
```

Order request –

```
@Data
```

```java
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class OrderRequest {

    private Long productId;
    private long totalAmount;
    private long quantity;
    private PaymentMode paymentMode;

}
```

Payment mode enum –

```java
public enum PaymentMode {

    CASH,PAYPAL,DEBIT_CARD,CREDIT_CARD,APPLE_PAY
}
```

Order repo –

```java
@Repository
public interface OrderRepository extends
JpaRepository<Order,Long> {
}
```

Order service –

```java
public interface OrderService {
    Long placeOrder(OrderRequest orderRequest);
}
```

Order service impl –

```java
@Service
@Log4j2
public class OrderServiceImpl implements OrderService {
```

```
    @Autowired
    private OrderRepository orderRepository;

    @Override
    public Long placeOrder(OrderRequest orderRequest) {
        //Order Entity -save data with status order created
        Order order = Order.builder()
                .amount(orderRequest.getTotalAmount())
                .orderStatus("CREATED")
                .productId(orderRequest.getProductId())
                .orderDate(Instant.now())
                .quantity(orderRequest.getQuantity())
                .build();

        order=orderRepository.save(order);
        log.info("order placed successfully with order id" +
order.getOrderId());
        //product service - block product (reduce quantity)
        //payment service- complete(success) else
        //cancelled

        log.info("placing order request : " + orderRequest);
        return order.getOrderId();
    }
}
```

Yaml –

```
server:
 port: 9091

#database config
spring:
 datasource:
    driver-class: com.cj.jdbc.Driver
    username: root
    url:
jdbc:mysql://${LIVE_HOST:localhost}:3306/order?serverTimezone=UTC
    password: user

 #jpa config
 jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
```

```
        format_sql: true
    hibernate:
      ddl-auto: update
    show-sql: true

    #eureka config
  application:
    name: ORDER-SERVICE

    eureka:
      instance:
        prefer-ip-address: true
      client:
        fetch-registry: true
        register-with-eureka: true
        service-url:
          defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

To configure config server in each service

Step 1) config server

Main file -

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

}
```

Yaml file -

```
server:
 port: 9296

spring:
 application:
   name: CONFIG-SERVER

 cloud:
   config:
     server:
       git:
         uri:
https://github.com/chail-a-singh/e-commerce-config-server
         clone-on-start: true

eureka:
 instance:
   prefer-ip-address: true
 client:
   fetch-registry: true
   register-with-eureka: true
   service-url:
```

```
    defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

Pom.xml -

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId
>
</dependency>
```

Product service - client

Pom.xml

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactI
d>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

yaml

```
 config:
   import: configserver:http://localhost:9296

#eureka:
#  instance:
```

```
#    prefer-ip-address: true
#  client:
#    fetch-registry: true
#    register-with-eureka: true
#    service-url:
#      defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

(Do same for every client service)

Write a code for reduce quantity in product service

Controller

```
@PutMapping("/reduceQuantity/{id}")
public ResponseEntity<Void> reduceQuantity(@PathVariable("id")
Long productId, @RequestParam long quantity){
    productService.reduceQuantity(productId,quantity);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Service

```
void reduceQuantity(Long productId, long quantity);
```

serviceImpl

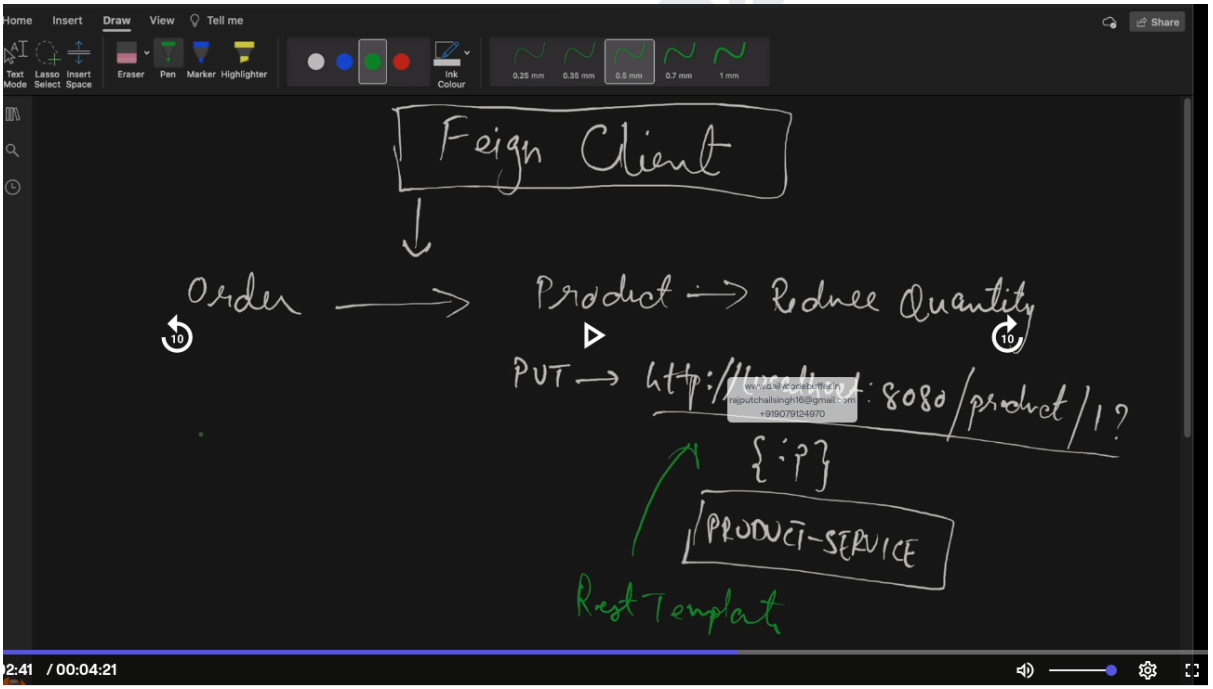
```
@Override
public void reduceQuantity(Long productId, long quantity) {
    log.info("Reduce quantity {} for id : {}" +
quantity,productId);
    Product product=productRepository.findById(productId)
            .orElseThrow(() -> new ProductCustomException("Product
not found","PRODUCT_NOT_FOUND"));

    if (product.getQuantity() < quantity){
        throw new ProductCustomException("Product Does Not have
sufficient quantity","INSUFFICIENT_QUANTITY");
    }
    product.setQuantity(product.getQuantity()-quantity);
    productRepository.save(product);
    log.info("Product Quantity Updated");
}
```

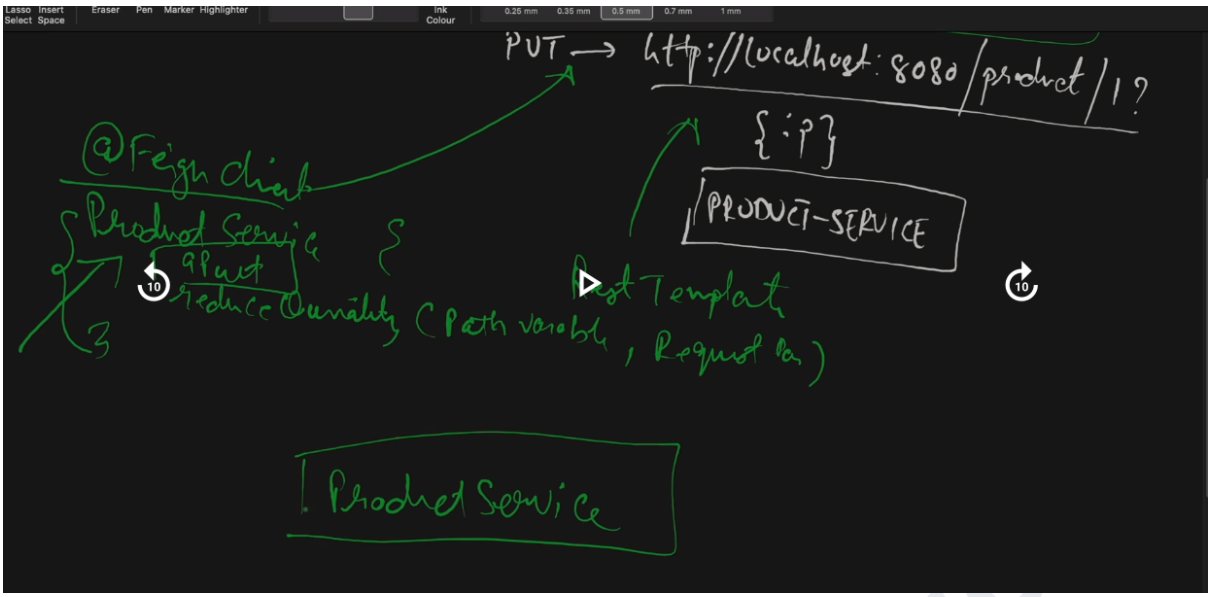
FEIGN-CLIENT

We have a product service where we have implemented an api for reducing quantity.
Now I need to call this api in my order service.
Both the services are different so how can i call , here comes the feign client.
(You can use RestTemplate as well)
Feign client and Rest Template are Api clients with which we can call the other api's.

Feign client is the rest client which allows us as a java developer to use the api's call using a declared function.

In the rest template we call using the application name without worry about the port .
In feign client we declare service as a @FeignClient and in that we declare the method and that interface react as a feign client who provides us a method.





→ Calling reduce quantity api from placeorder service using feign client.

Step 1) add dependency open feign in order service
Step 2) add @EnableFeignClient in main file of order service
Step 3) create separate package external in order project and create an interface class with name ProductService and calling the declaration method
Step 4) autowire this interface in orderimpl and request the id and quantity

Pom.xml - order

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

Main class - order

```
@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {
   public static void main(String[] args) {
      SpringApplication.run(OrderServiceApplication.class, args);
   }

}
```

Product -Service(interface) - order

```
package order.external.client;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestParam;

@FeignClient(name = "PRODUCT-SERVICE/api/product")
public interface ProductService {

    @PutMapping("/reduceQuantity/{id}")
    ResponseEntity<Void> reduceQuantity(@PathVariable("id") Long
productId, @RequestParam long quantity);

}
```

Order service impl - order

```
@Service
@Log4j2
public class OrderServiceImpl implements OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductService productService;
orderRequest
    @Override
    public Long placeOrder(OrderRequest orderRequest) {
        //Order Entity -save data with status order created

        log.info("placing order request : {}",orderRequest);
        //calling reduce quantity using feing client

productService.reduceQuantity(orderRequest.getProductId(),orderReq
uest.getQuantity());
        log.info("creating order with status CREATED");
        Order order = Order.builder()
                .amount(orderRequest.getTotalAmount())
                .orderStatus("CREATED")
                .productId(orderRequest.getProductId())
                .orderDate(Instant.now())
```

```
                .quantity(orderRequest.getQuantity())
                .build();

        order=orderRepository.save(order);
        log.info("order placed successfully with order id" +
order.getOrderId());
        //product service - block product (reduce quantity)

        //payment service- complete(success) else
        //cancelled

        log.info("placing order request : " + orderRequest);
        return order.getOrderId();
    }
}
```

OBJ

Error Decode added into order service

```
package order.config;

import feign.codec.ErrorDecoder;
import order.external.decoder.CustomErrorDecoder;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FeignConfig {

   ErrorDecoder errorDecoder(){
       return new CustomErrorDecoder();
   }
}
```

CustomErrorDecoder

```
package order.external.decoder;

import com.fasterxml.jackson.databind.ObjectMapper;
import feign.Response;
import feign.codec.ErrorDecoder;
import lombok.extern.log4j.Log4j2;
import order.external.exception.CustomException;
import order.external.response.ExceptionResponse;

import java.io.IOException;

@Log4j2
public class CustomErrorDecoder implements ErrorDecoder {

   @Override
   public Exception decode(String s, Response response) {
       ObjectMapper objectMapper = new ObjectMapper();
       log.info("::{}",response.request().url());
       log.info("::{}",response.request().headers());

       try {
           ExceptionResponse exceptionResponse =
objectMapper.readValue(response.body().asInputStream(),
```

```
                    ExceptionResponse.class);

            return new
CustomException(exceptionResponse.getMessage(),
exceptionResponse.getStatusCode(), response.status());

        } catch (IOException e) {
            throw new CustomException("Internal server
error","INTERNAL_SERVER_ERROR",500);
        }
    }
}
```

CustomException

```
package order.external.exception;

import lombok.Data;

@Data
public class CustomException extends RuntimeException{

    private String statusCode;
    private int status;

    public CustomException(String message, String statusCode,int
status) {
        super(message);
        this.statusCode = statusCode;
        this.status=status;
    }

}
```

ExceptionHandler

```
package order.external.exception;

import order.external.response.ExceptionResponse;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
```

```
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEnti
tyExceptionHandler;


@ControllerAdvice
public class ExceptionHandler extends
ResponseEntityExceptionHandler {

   @org.springframework.web.bind.annotation.ExceptionHandler(
CustomException.class)
   public ResponseEntity<ExceptionResponse>
handleCustomException(CustomException exception){

       return new ResponseEntity<>(new
ExceptionResponse().builder()
                .message(exception.getMessage())
                .statusCode(exception.getStatusCode())
                .build(),
HttpStatus.valueOf(exception.getStatus()));
   }
}
```

Exception Response

```
package order.external.response;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ExceptionResponse {

   private String message;
   private String statusCode;
}
```

Installing Zipkins Via Docker

1. Zipkin - is a very efficient tool for distributed tracing in the microservices ecosystem. Distributed tracing, in general, is the latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business use case.

Let's say from our application, we have to call 4 different services/components for a transaction. Herewith distributed tracing enabled, we can measure which component took how much time.

Distributed tracing is useful during debugging when lots of underlying systems are involved and the application becomes slow in any particular situation. In such cases, we first need to identify which underlying service is actually slow. Once the slow service is identified, we can work to fix that issue. Distributed tracing helps in identifying that slow component in the ecosystem.

Internally it has 4 modules –

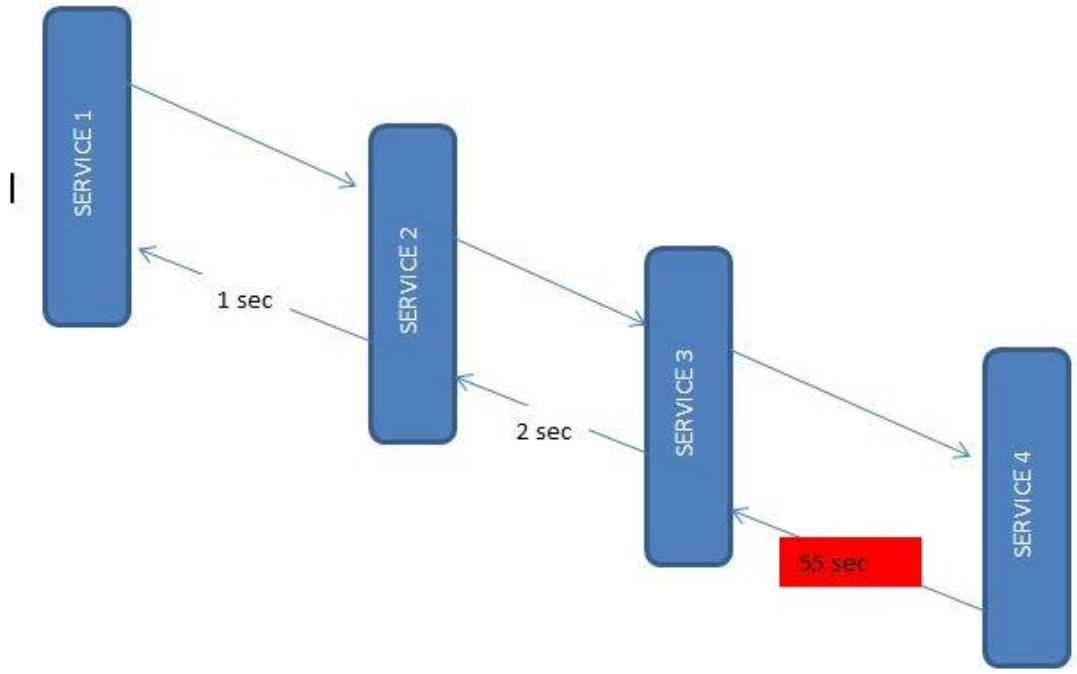
1. **Collector – Once any component sends the trace data, it arrives at the Zipkin collector daemon. Here the *trace data is validated, stored, and indexed for lookups by the Zipkin collector*.**
2. **Storage – This module stores and indexes the lookup data in the backend. Cassandra, Elastic search and mysql are supported.**
3. **Search – This module provides a simple JSON API for finding and retrieving traces stored in the backend. The primary consumer of this API is the Web UI.**
4. **Web UI – A very nice UI interface for viewing traces.**

2. Sleuth - sleuth is another tool from the Spring cloud family. It is used to generate the *trace id*, *span id* and add this information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and ELK **etc. to store, index and process log files.**

Step 1) install docker in the machine and run the docker
Step 2) run this command to install zipkin image using docker

docker run -d -p 9411:9411 openzipkin/zipkin

Step 3) go to browser - localhost:9411 → zipkin dashboard



Now, adding zipkin and sleuth in project

Step 1) add both (zipkin and sleuth) dependency in the pom.xml in every client

Zipkin

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

Sleuth

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

Step 2) you will get a trace id and span id in the console while request any url
Trace id - unique throughout the request
Span id - unique per segment/scope of the application

Step 3) go to zipkin dashboard and find a trace by hitting run query also you can search by trace id

PAYMENT - SERVICE

MAIN-CLASS -

```
@SpringBootApplication
public class PaymentServiceApplication {

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

ENTITY -

```
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
import java.time.Instant;
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class TransactionDetails {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = "transaction_id")
   private Long transactionId;
   @Column(name = "order_id")
   private Long orderId;
   @Column(name = "payment_mode")
   private String paymentMode;
   @Column(name = "reference_number")
   private String referenceNumber;
   @Column(name = "payment_date")
   private Instant paymentDate;
   @Column(name = "payment_status")
   private String paymentStatus;
   @Column(name = "total_amount")
   private long amount;


}
```

MODEL -

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentRequest {

    private Long orderId;
    private long amount;
    private String referenceNumber;
    private PaymentMode paymentMode;
}
```

ENUM

```
public enum PaymentMode {

    CASH,PAYPAL,DEBIT_CARD,CREDIT_CARD,APPLE_PAY
}
```

CONTROLLER -

```
package payment.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import payment.model.PaymentRequest;
import payment.service.PaymentService;

@RestController
@RequestMapping("/api/payment")
public class PaymentController {

    @Autowired
    private PaymentService paymentService;


    @PostMapping("/")
    public ResponseEntity<Long> doPayment(@RequestBody
PaymentRequest paymentRequest){
        return new ResponseEntity<>(
                paymentService.doPayment(paymentRequest),
HttpStatus.OK
        );

    }

}
```

SERVICE -

```
public interface PaymentService {
    Long doPayment(PaymentRequest paymentRequest);
}
```

SERVICE-IMPL -

```
@Service
@Log4j2
public class PaymentServiceImpl implements PaymentService{
    @Autowired
    private TransactionsDetailsRepository
transactionsDetailsRepository;

    @Override
    public Long doPayment(PaymentRequest paymentRequest) {
        log.info("recording payment details: {}",paymentRequest);

        TransactionDetails transactionDetails =
TransactionDetails.builder()
                .paymentDate(Instant.now())

.paymentMode(paymentRequest.getPaymentMode().name())
                .paymentStatus("SUCCESS")
                .orderId(paymentRequest.getOrderId())

.referenceNumber(paymentRequest.getReferenceNumber())
                .amount(paymentRequest.getAmount())
                .build();

        transactionsDetailsRepository.save(transactionDetails);
        log.info("Transaction completed with id :
{}",transactionDetails.getTransactionId());

        return transactionDetails.getTransactionId();
    }
}
```

REPOSITORY-

```
@Repository
public interface TransactionsDetailsRepository extends
JpaRepository<TransactionDetails,Long> {
}
```

Xml -

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-parent</artifactId>
     <version>2.7.4</version>
     <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>PaymentService</groupId>
  <artifactId>PaymentService</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>PaymentService</name>
  <description>Demo project for Spring Boot</description>
  <properties>
     <java.version>17</java.version>
     <spring-cloud.version>2021.0.4</spring-cloud.version>
  </properties>
  <dependencies>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
     </dependency>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter</artifactId>
     </dependency>
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
     </dependency>
     <dependency>
        <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactI
d>
     </dependency>
```

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-zipkin</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </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>
  <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
```

```
                </exclude>
            </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

Yaml -

```
server:
 port: 9092

#database config
spring:
 datasource:
   driver-class: com.cj.jdbc.Driver
   username: root
   url:
jdbc:mysql://${live_host:localhost}:3306/payment?serverTimezone=U
TC
   password: user

 #jpa config
 jpa:
   properties:
     hibernate:
       dialect: org.hibernate.dialect.MySQL8Dialect
       format_sql: true
   hibernate:
     ddl-auto: update
   show-sql: true

 application:
   name: PAYMENT-SERVICE

 config:
   import: configserver:http://localhost:9296

#eureka:
#  instance:
#    prefer-ip-address: true
#  client:
#    fetch-registry: true
#    register-with-eureka: true
```

```
#    service-url:
#      defaultZone:
${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

Adding do payment into order service using feign client

1)

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

2)

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentRequest {

    private Long orderId;
    private long amount;
    private String referenceNumber;
    private PaymentMode paymentMode;
}
```

3)

```
package order.external.client;

import order.external.request.PaymentRequest;
import org.springframework.cloud.openfeign.FeignClient;
```

```java
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;

@FeignClient(name = "PAYMENT-SERVICE/api/payment/")
public interface PaymentService {

    @PostMapping("/")
    public ResponseEntity<Long> doPayment(@RequestBody
PaymentRequest paymentRequest);
}
```

4)

```java
package order.service;

import lombok.extern.log4j.Log4j2;
import order.entity.Order;
import order.external.client.PaymentService;
import order.external.client.ProductService;
import order.external.exception.CustomException;
import order.external.request.PaymentRequest;
import order.model.OrderRequest;
import order.model.OrderResponse;
import order.repository.OrderRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.time.Instant;

@Service
@Log4j2
public class OrderServiceImpl implements OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductService productService;

    @Autowired
    private PaymentService paymentService;
```

```
    @Override
    public Long placeOrder(OrderRequest orderRequest) {
        //Order Entity -save data with status order created

        log.info("placing order request : {}",orderRequest);
        //calling reduce quantity using feign client
        //product service - block product (reduce quantity)

productService.reduceQuantity(orderRequest.getProductId(),orderRe
quest.getQuantity());
        log.info("creating order with status CREATED");
        Order order = Order.builder()
                .amount(orderRequest.getTotalAmount())
                .orderStatus("CREATED")
                .productId(orderRequest.getProductId())
                .orderDate(Instant.now())
                .quantity(orderRequest.getQuantity())
                .build();

        order=orderRepository.save(order);
        log.info("calling payment service to complete payment");
        PaymentRequest paymentRequest = PaymentRequest.builder()
                .orderId(order.getOrderId())
                .paymentMode(orderRequest.getPaymentMode())
                .amount(orderRequest.getTotalAmount())
                .build();
        String orderStatus = null;

        try{
            paymentService.doPayment(paymentRequest);
            log.info("payment done successfully , changing order
status to placed ");
            orderStatus="PLACED";

        }catch (Exception e){
            log.error("Error occurring in payment , changing order
status to PAYMENT FAILED");
            orderStatus="PAYMENT_FAILED";
        }

        order.setOrderStatus(orderStatus);
        orderRepository.save(order);
        log.info("order placed successfully with order id" +
order.getOrderId());

        //payment service- complete (success) else
```

```
        //cancelled
        log.info("placing order request : " + orderRequest);
        return order.getOrderId();
    }
```

Implement get order details in order service

Controller

```
@RestController
@RequestMapping("/api/order")
@Log4j2
public class OrderController {

    @Autowired
    private OrderService orderService;

    @GetMapping("/{orderId}")
    public ResponseEntity<OrderResponse>
getOrderDetails(@PathVariable Long orderId) {

        OrderResponse orderResponse=
orderService.getOrderDetails(orderId);

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

}
```

Order response

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class OrderResponse {

    private Long orderId;
    private Instant orderDate;
    private String orderStatus;
    private long amount;
}
```

Order service -

```
public interface OrderService {
    OrderResponse getOrderDetails(Long orderId);
}
```

Order service impl -

```
@Service
@Log4j2
public class OrderServiceImpl implements OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductService productService;

    @Autowired
    private PaymentService paymentService;


    @Override
    public OrderResponse getOrderDetails(Long orderId) {
```

```
        log.info("get order details for orderId : {}",orderId);

        Order order =
orderRepository.findById(orderId).orElseThrow(() -> new
CustomException("Order Not found ","NOT_FOUND",404));
        OrderResponse orderResponse = OrderResponse.builder()
                    .orderId(order.getOrderId())
                    .orderStatus(order.getOrderStatus())
                    .amount(order.getAmount())
                    .orderDate(order.getOrderDate())
                    .build();
        return orderResponse;
    }
}
```

Fetching product data for get order details api

Step 1) create static inner class of product Details in order response

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class OrderResponse {

    private Long orderId;
    private Instant orderDate;
    private String orderStatus;
    private long amount;

    private ProductDetails productDetails;

    @Data
    @Builder
    @NoArgsConstructor
    @AllArgsConstructor
    //static inner class
    public static class ProductDetails {

        private Long productId;
        private String productName;
        private double price;
        private long quantity;
```

```
    }
}
```

Step 2) define rest template bean in main class

```
@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {

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

  @Bean
  @LoadBalanced
  public RestTemplate restTemplate(){
      return new RestTemplate();
  }

}
```

Step 3) add product service dependency in order service

```
<dependency>
    <groupId>ProductService</groupId>
    <artifactId>ProductService</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>compile</scope>
</dependency>
```

Step 4) implement in order service impl

```
@Service
@Log4j2
public class OrderServiceImpl implements OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductService productService;

    @Autowired
    private PaymentService paymentService;

    @Autowired
    private RestTemplate restTemplate;


@Override
    public OrderResponse getOrderDetails(Long orderId) {
        log.info("get order details for orderId : {}",orderId);

        Order order =
orderRepository.findById(orderId).orElseThrow(() -> new
CustomException("Order Not found ","NOT_FOUND",404));

        log.info("invoking product service to fetch the product
for id:{}",order.getProductId());

        ProductResponse productResponse =
restTemplate.getForObject(

"http://PRODUCT-SERVICE/api/product/"+order.getProductId(),Produc
tResponse.class
        );

        OrderResponse.ProductDetails
productDetails=OrderResponse.ProductDetails
                .builder()
                .productName(productResponse.getProductName())
                .productId(productResponse.getProductId())
                .build();
```

```
        OrderResponse orderResponse = OrderResponse.builder()
                .orderId(order.getOrderId())
                .orderStatus(order.getOrderStatus())
                .amount(order.getAmount())
                .orderDate(order.getOrderDate())
                .productDetails(productDetails)
                .build();
        return orderResponse;
    }
}
```

Fetching payment details for get order details api

Step 1) add static inner class in order response in order service

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class OrderResponse {

    private Long orderId;
    private Instant orderDate;
    private String orderStatus;
    private long amount;

    private ProductDetails productDetails;

    private PaymentDetails paymentDetails;

    @Data
    @Builder
    @NoArgsConstructor
    @AllArgsConstructor
    //static inner class
    public static class ProductDetails {

        private Long productId;
        private String productName;
        private double price;
        private long quantity;

    }
```

```
    //inner class for payment response

    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    @Builder
    public static class PaymentDetails {

        private Long paymentId;
        private String status;
        private PaymentMode paymentMode;
        private long amount;
        private Instant paymentDate;
        private Long orderId;
    }
}
```

Step 2) add payment response in order service

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentResponse {

    private Long paymentId;
    private String status;
    private PaymentMode paymentMode;
    private long amount;
    private Instant paymentDate;
    private Long orderId;
}
```

Step 3) add payment details in order impl

```
@Service
@Log4j2
public class OrderServiceImpl implements OrderService {

    @Autowired
    private OrderRepository orderRepository;
```

```
    @Autowired
    private ProductService productService;

    @Autowired
    private PaymentService paymentService;

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public Long placeOrder(OrderRequest orderRequest) {
        //Order Entity -save data with status order created

        log.info("placing order request : {}",orderRequest);
        //calling reduce quantity using feign client
        //product service - block product (reduce quantity)

productService.reduceQuantity(orderRequest.getProductId(),orderRe
quest.getQuantity());
        log.info("creating order with status CREATED");
        Order order = Order.builder()
                .amount(orderRequest.getTotalAmount())
                .orderStatus("CREATED")
                .productId(orderRequest.getProductId())
                .orderDate(Instant.now())
                .quantity(orderRequest.getQuantity())
                .build();

        order=orderRepository.save(order);
        log.info("calling payment service to complete payment");
        PaymentRequest paymentRequest = PaymentRequest.builder()
                .orderId(order.getOrderId())
                .paymentMode(orderRequest.getPaymentMode())
                .amount(orderRequest.getTotalAmount())
                .build();
        String orderStatus = null;

        try{
            paymentService.doPayment(paymentRequest);
            log.info("payment done successfully , changing order
status to placed ");
            orderStatus="PLACED";

        }catch (Exception e){
```

```
            log.error("Error occurring in payment , changing order
status to PAYMENT_FAILED");
            orderStatus="PAYMENT_FAILED";
        }

        order.setOrderStatus(orderStatus);
        orderRepository.save(order);
        log.info("order placed successfully with order id" +
order.getOrderId());

        //payment service- complete (success) else
        //cancelled
        log.info("placing order request : " + orderRequest);
        return order.getOrderId();
    }

    @Override
    public OrderResponse getOrderDetails(Long orderId) {
        log.info("get order details for orderId : {}",orderId);

        Order order =
orderRepository.findById(orderId).orElseThrow(() -> new
CustomException("Order Not found ","NOT_FOUND",404));

        log.info("invoking product service to fetch the product
for id:{}",order.getProductId());

        ProductResponse productResponse =
restTemplate.getForObject(

"http://PRODUCT-SERVICE/api/product/"+order.getProductId(),Produc
tResponse.class
        );

        log.info("getting payment information from payment service
");

        PaymentResponse paymentResponse =
restTemplate.getForObject(

"http://PAYMENT-SERVICE/api/payment/order/"+order.getOrderId(),Pa
ymentResponse.class
        );
```

```
        OrderResponse.ProductDetails
productDetails=OrderResponse.ProductDetails
                .builder()
                .productName(productResponse.getProductName())
                .productId(productResponse.getProductId())
                .build();


        OrderResponse.PaymentDetails paymentDetails =
OrderResponse.PaymentDetails
                .builder()
                .paymentId(paymentResponse.getPaymentId())
                .status(paymentResponse.getStatus())
                .paymentDate(paymentResponse.getPaymentDate())
                .paymentMode(paymentResponse.getPaymentMode())
                .build();


        OrderResponse orderResponse = OrderResponse.builder()
                .orderId(order.getOrderId())
                .orderStatus(order.getOrderStatus())
                .amount(order.getAmount())
                .orderDate(order.getOrderDate())
                .productDetails(productDetails)
                .paymentDetails(paymentDetails)
                .build();
        return orderResponse;
    }
}
```

Step 4) add payment response in payment service

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentResponse {

    private Long paymentId;
    private String status;
    private PaymentMode paymentMode;
    private long amount;
    private Instant paymentDate;
    private Long orderId;
}
```

Step 5) define byOrderId in payment transaction repository

```
@Repository
public interface TransactionsDetailsRepository extends
JpaRepository<TransactionDetails,Long> {
    TransactionDetails findByOrderId(Long orderId);

}
```

Step 6) payment controller

```
@GetMapping("/order/{orderId}")
public ResponseEntity<PaymentResponse>
getPaymentDetailsByOrderId(@PathVariable String orderId){
    return new
ResponseEntity<>(paymentService.getPaymentDetailsByOrderId(orderI
d),HttpStatus.OK);
}
```

Step 7) payment service

```
public interface PaymentService {
    Long doPayment(PaymentRequest paymentRequest);

    PaymentResponse getPaymentDetailsByOrderId(String orderId);
}
```

Step 8) payment serviceimpl

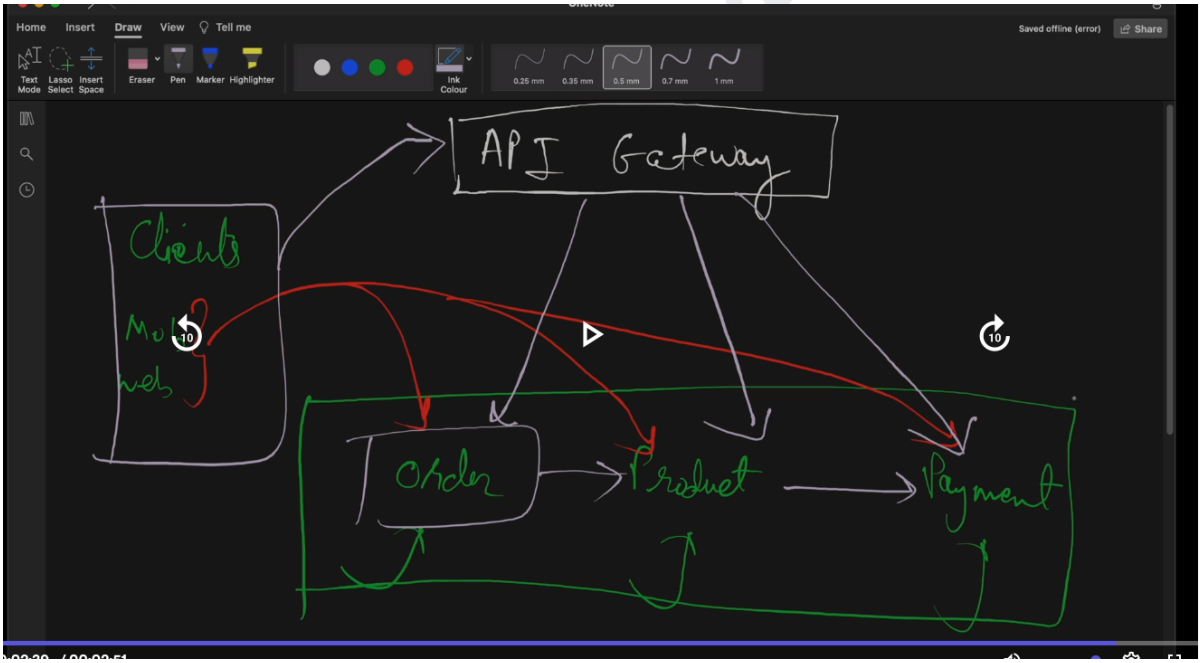
```
@Service
@Log4j2
public class PaymentServiceImpl implements PaymentService{

    @Autowired
    private TransactionsDetailsRepository
transactionsDetailsRepository;

    @Override
    public Long doPayment(PaymentRequest paymentRequest) {
        log.info("recording payment details: {}",paymentRequest);

        TransactionDetails transactionDetails =
TransactionDetails.builder()
                .paymentDate(Instant.now())
```

```
.paymentMode(paymentRequest.getPaymentMode().name())
                .paymentStatus("SUCCESS")
                .orderId(paymentRequest.getOrderId())

.referenceNumber(paymentRequest.getReferenceNumber())
                .amount(paymentRequest.getAmount())
                .build();

        transactionsDetailsRepository.save(transactionDetails);
        log.info("Transaction completed with id :
{}",transactionDetails.getTransactionId());

        return transactionDetails.getTransactionId();
    }

    @Override
    public PaymentResponse getPaymentDetailsByOrderId(String
orderId) {

        log.info("getting payment details for order
id:{}",orderId);

        TransactionDetails transactionDetails =
transactionsDetailsRepository.findByOrderId(Long.valueOf(orderId)
);

        PaymentResponse paymentResponse =
                PaymentResponse.builder()

.paymentId(transactionDetails.getTransactionId())

.paymentMode(PaymentMode.valueOf(transactionDetails.getPaymentMod
e()))

.paymentDate(transactionDetails.getPaymentDate())
                        .orderId(transactionDetails.getOrderId())

.status(transactionDetails.getPaymentStatus())
                        .amount(transactionDetails.getAmount())
                        .build();
        return paymentResponse;
    }
}
```

API GATEWAY

Till now we are calling every microservice directly which is running on different servers or ports.
Now any public request coming from the client(mobile,web etc)
Generally all these requests should be protected from the public network to the private network.
Whatever request coming from clients can't directly come to our microservices.
It has to be protected , it has to come via one of the services and that service should be responsible to handle all those checks(authorization , authentication etc).
There should be only one single point of contact from where all those services will contact to and after that those services should interact with internal services.
For that reason we can have an
.

It will act as a gateway for all the requests coming from the clients.in that way it will work.



API GATEWAY - SERVICE

Step 1) create spring boot project with dependencies

-Spring cloud (cloud bootstrap)
-Spring cloud routing (gateway)
-Web (spring reactive web)
-Zipkin client
-Sleuth
-Eureka Discovery client
-Lombok
-Actuator
-config client

Step 1) yaml config

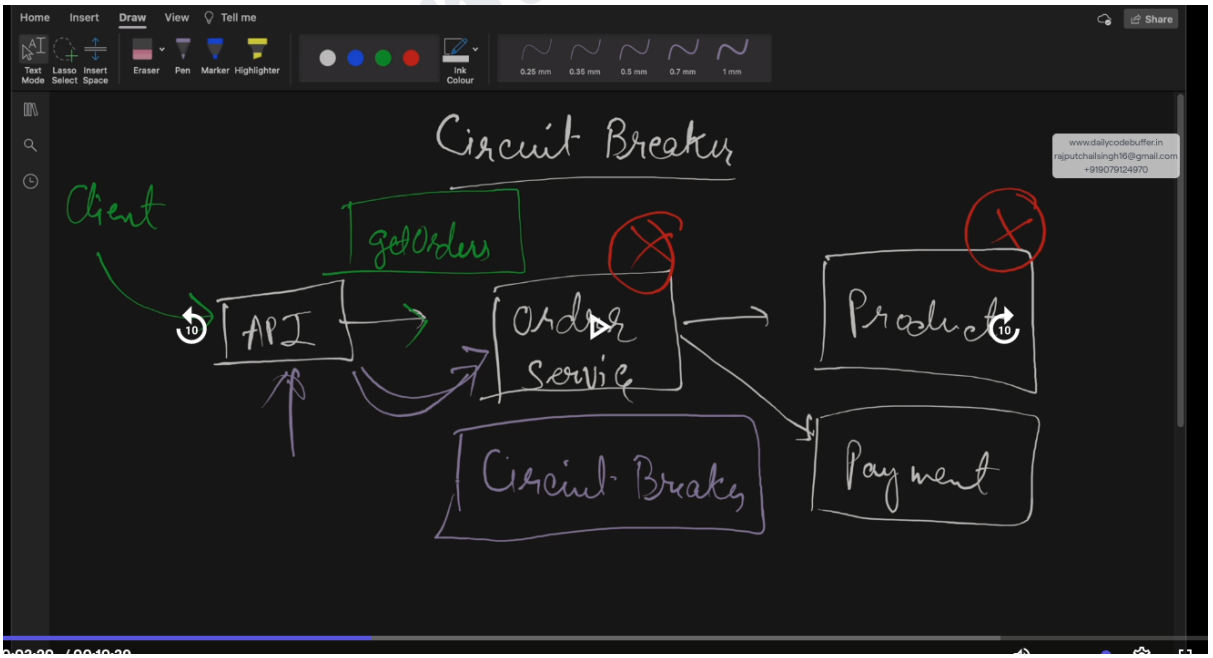
```
server:
 port: 9093

spring:
 application:
   name: API-GATEWAY

 config:
   import: configserver:http://localhost:9296

 cloud:
   gateway:
     routes:
       - id : PRODUCT-SERVICE
         uri: lb://PRODUCT-SERVICE
         predicates:
             - Path=/api/order/**
       - id : ORDER-SERVICE
         uri: lb://ORDER-SERVICE
         predicates:
             - Path=/api/order/**
       - id : PAYMENT-SERVICE
         uri: lb://PAYMENT-SERVICE
         predicates:
           - Path=/api/payment/**
```

Circuit Breaker

All the microservice are interlinked and all those access through one contact point that is **API-GATEWAY** .
Now if one of the internal services is not down and another service is dependent on this service, an error will arise.
Since the api gateway doesn't know the internal dependency of each service , it just passes the request .
Until the service is down, the api gateway will pass all the requests to the service to get the data that is just a waste of resources and waste of time as well. We already know the particular service is down so why send all the requests?

Here comes the solutions.

So what we can do is , we have a mechanism , through which it will constantly check if the particular service is up or down.
If it is down then it will not send all the requests to the service and it will wait for sometime to start that particular service and when the service gets up it will start sending the request .
In this scenario the circuit breaker will help us.
There are few libraries who allow us to use this .
Earlier we had **hystrix** (now it is deprecated) , now we have **Resilience4J**
So **Resilience 4J library** allows us to implement the circuit breaker



Circuit breakers have some different **stages** , based on that it will try to handle all the scenarios.
So suppose i want to call the order service from api gateway
So I will add one circuit breaker at api gateway level and that circuit breaker will handle all scenarios for order service .

If I want to handle product service , I will handle circuit breakers at order service level.

Generally circuit breaker have 3 states

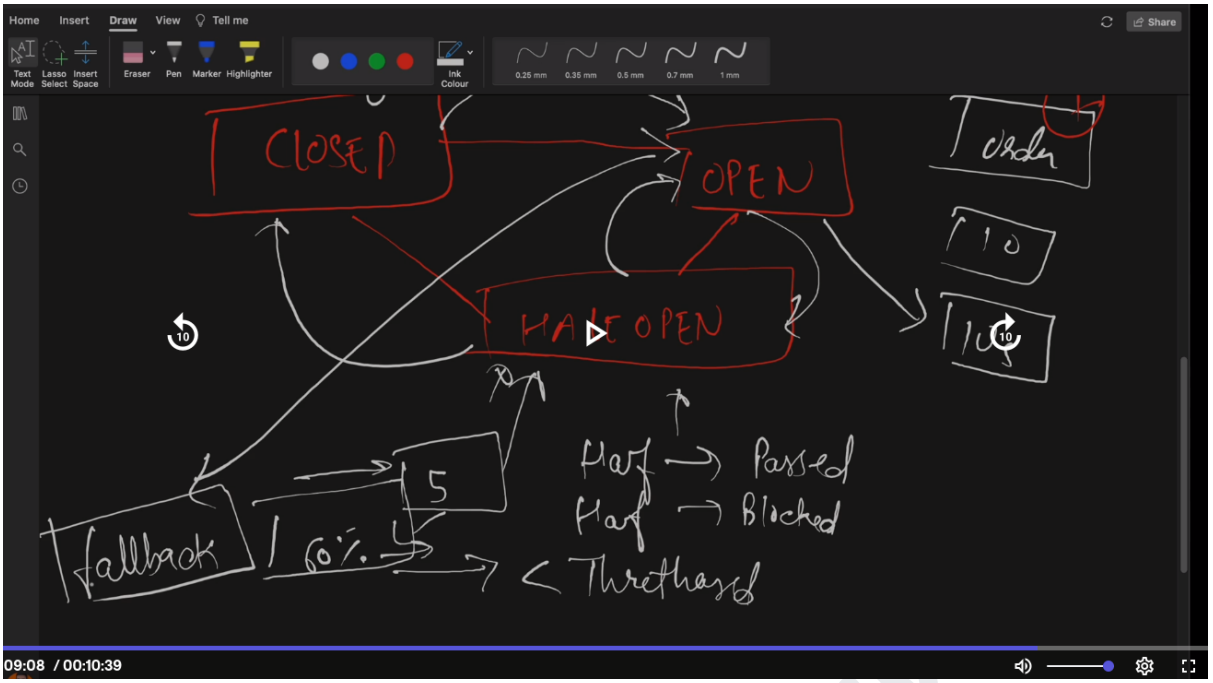
1) **close**
2) **Open**
3) **half-open**

These are all connected in the **triangle**.
By default if all are in working mode , the state is closed and that is a healthy scenario.

If the order service is down and a request coming from the api gateway will fail.
At that time we can configure that if this number of requests fails then change the state from closed to open.

When the circuit is open so the request will not go to the service and we also configure the timing , after particular time it will change to half open
Whenever it is half open so half of the request will pass through and half will be blocked .
Based on the success rate we can change the state from half open to closed and now all requests can pass.
If success rate is below the threshold then it will change from half open to open state .and will wait for given time and doing the same process

Whenever the circuit breaker is open we can send the fallback method which will send the response to the client



There might be multiple scenarios that what fall back method needs to do and what fallback data needs to send .
If the scenario is complicated then we can directly send the message that this service is down.
Or if a client requests to get the data and service is down so we are not able to connect with the database at that time we can get the data from cache and send it to the client .
After handling all the scenarios our request won't be fail

In this way circuit breakers are helpful to us.

IMPLEMENTING CIRCUIT BREAKER IN API GATEWAY

Step 1) add Resilience4J dependency in api gateway project

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

<artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience
4j</artifactId>
</dependency>
```

Step 2) Fallback controller - api gateway

```
@RestController
public class FallBackController {

    @GetMapping("/orderServiceFallBack")
    public String orderServiceFallback(){
        return "Order Service is down";
    }

    @GetMapping("/productServiceFallBack")
    public String productServiceFallback(){
        return "Product Service is down";
    }

    @GetMapping("/paymentServiceFallBack")
    public String paymentServiceFallback(){
        return "Payment Service is down";
    }

}
```

Step3) main class

```
@SpringBootApplication
public class ApiGatewayApplication {

  public static void main(String[] args) {
```

```
        SpringApplication.run(ApiGatewayApplication.class, args);
    }

    @Bean
    public Customizer<Resilience4JCircuitBreakerFactory>
defaultCustomizer(){
        return factory -> factory.configureDefault(
                id -> new Resilience4JConfigBuilder(id)
                        .circuitBreakerConfig(
                                CircuitBreakerConfig.ofDefaults()
                        ).build()
        );

    }
}
```

Step 4) yaml

```
server:
  port: 9093

spring:
  application:
    name: API-GATEWAY

  config:
    import: configserver:http://localhost:9296

  cloud:
    gateway:
      routes:
        - id : ORDER-SERVICE
          uri: lb://ORDER-SERVICE
          predicates:
              - Path=/api/order/**
          filters:
            - name: CircuitBreaker
              args:
                name: ORDER-SERVICE
                fallbackuri: forward:/orderServiceFallBack
        - id: PRODUCT-SERVICE
          uri: lb://PRODUCT-SERVICE
          predicates:
            - Path=/api/order/**
```

```
        filters:
          - name: CircuitBreaker
            args:
              name: PRODUCT-SERVICE
              fallbackuri: forward:/productServiceFallBack
      - id : PAYMENT-SERVICE
        uri: lb://PAYMENT-SERVICE
        predicates:
          - Path=/api/payment/**
        filters:
          - name: CircuitBreaker
            args:
              name: PAYMENT-SERVICE
              fallbackuri: forward:/paymentServiceFallBack
```

IMPLEMENTING CIRCUIT BREAKER IN ORDER SERVICE

Step 1) add dependency

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

<artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience
4j</artifactId>
</dependency>
```

Step 2) Yaml

```
resilience4j:
 circuitbreaker:
   instances:
     external:
       event-consumer-buffer-size: 10
       failure-rate-threshold: 50
       minimum-number-of-calls: 5
       automatic-transition-from-open-to-half-open-enabled: true
       wait-duration-in-open-state: 5s
       permitted-number-of-calls-in-half-open-state: 3
```

```
        sliding-window-size: 10
        sliding-window-type: COUNT_BASED
```

Step 3) payment service interface in order service

```
package order.external.client;

import
io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import order.external.exception.CustomException;
import order.external.request.PaymentRequest;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;

@CircuitBreaker(name = "external",fallbackMethod = "fallback")
@FeignClient(name = "PAYMENT-SERVICE/api/payment/")
public interface PaymentService {

    @PostMapping("/")
    public ResponseEntity<Long> doPayment(@RequestBody
PaymentRequest paymentRequest);

    default void fallback(Exception e){
        throw new CustomException("PAYMENT SERVICE is not available
" , "UNAVAILABLE" , 500);
    }


}
```

Step 4) product service interface in order service

```
package order.external.client;

import
io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import order.external.exception.CustomException;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestParam;

@CircuitBreaker(name = "external",fallbackMethod = "fallback")
@FeignClient(name = "PRODUCT-SERVICE/api/product")
public interface ProductService {

    @PutMapping("/reduceQuantity/{id}")
    ResponseEntity<Void> reduceQuantity(@PathVariable("id") Long
productId, @RequestParam long quantity);

    default void fallback(Exception e){
        throw new CustomException("product service is not available
" , "UNAVAILABLE" , 500);
    }

}
```

IMPLEMENTING RATE LIMITER IN API GATEWAY USING RESILLIENCE4J AND REDIS

To avoid detox attacks we need to implement rate limiter in api gateway,we don't want api's to be vulnerable.
In ratelimiter we will implement how many requests per user per second so for each and every api's we will have a max number of requests allowed.

We will implement rate limiter using Resilience and Redis

Redis is the in-memory database .

We will have to add redis in the application stack as well.

Using docker we can add a redis image.

Step 1) run redis

```
docker run --name latestredis -d -p6379:6379 redis
```

Step 2) add dependency in api gateway

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

Step 3) define userkeysolver bean in main file

```
@Bean
KeyResolver userKeySolver(){
  return exchange -> Mono.just("userKey");

}
```

Step 4) yaml

```
server:
 port: 9093

spring:
 application:
   name: API-GATEWAY

 config:
   import: configserver:http://localhost:9296

 cloud:
   gateway:
     routes:
       - id : ORDER-SERVICE
         uri: lb://ORDER-SERVICE
         predicates:
             - Path=/api/order/**
         filters:
           - name: CircuitBreaker
             args:
               name: ORDER-SERVICE
               fallbackuri: forward:/orderServiceFallBack
           - name: RequestRateLimiter
             args:
               redis-rate-limiter.replenishRate: 1
               redis-rate-limiter.burstCapacity: 1
```

```
        - id: PRODUCT-SERVICE
          uri: lb://PRODUCT-SERVICE
          predicates:
            - Path=/api/product/**
          filters:
            - name: CircuitBreaker
              args:
                name: PRODUCT-SERVICE
                fallbackuri: forward:/productServiceFallBack
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 1
                redis-rate-limiter.burstCapacity: 1
        - id : PAYMENT-SERVICE
          uri: lb://PAYMENT-SERVICE
          predicates:
            - Path=/api/payment/**
          filters:
            - name: CircuitBreaker
              args:
                name: PAYMENT-SERVICE
                fallbackuri: forward:/paymentServiceFallBack
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 1
                redis-rate-limiter.burstCapacity: 1
```

ReplenishRate - how many requests should be allowed per second.
Burst Capacity - how many request should be allowed in that particular one duration second

SPRING SECURITY

We have created multiple microservices and with all those microservices currently we are able to directly access it but there has to be a way that all the microservices are secured behind the particular authentication gateway.
All the requests that come to application are to be authenticated and authorised then only particular requests should be allowed.

We will implement – (spring security + oauth2 authentication)

Oauth2 - previously whenever we were developing a monolithic application we just used to add spring security and we were just doing session based authentication and for that particular user session was created and within that session interval the user was allowed to perform the operation.
Now when we are developing the microservices architecture , to create the session each and different services is not possible so what we do is to go ahead toward the Oauth2 authentication .
Oauth2 gives us authentication based on authentication code so there has to be an authentication server which will take the request , that request has the credential details and this authentication server will return the Auth Code now this code will check that credentials are valid or not and it will have all the authorised information as well.

That authentication code will be the unique JWT code .

This consist of 3 things
Header
Body
Signature

Whatever the auth code is created that will have code itself and duration as well for which particular duration the code is valid .
After that we need to generate the new code and that code will help me to be passed .
When we generate auth code , it is client responsibility to send this code for each and every request

Whenever the client will send request with this code , the services will handle those auth code decode the jwt

As the authentication server to give the authentication and everything we are going to use okta

Okta is a third party authentication library
Okta will give us a library that we will use and it will give us all the oauth2 implementation.

Whenever we are doing any request which ideally it auth2 tells that i am going to do a request on behalf of a user internally.
It will take consent from us.

Create Account on Okta
Okta configuration here
Registered user on okta

Adding Security in API Gateway

Step 1) **add dependencies in api gateway(pom.xml)**

Spring security

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

okta

```
   <dependency>
     <groupId>com.okta.spring</groupId>
     <artifactId>okta-spring-boot-starter</artifactId>
     <version>2.1.6</version>
   </dependency>
```

Step 2) **api gateway configuration - yaml**

```
okta:
 oauth2:
   issuer: https://dev-32888642.okta.com/oauth2/default
   audience: api://default
   client-id: 0oa6q0ym0oqRPhY4b5d7
   client-secret: _TPXZSXK21lohMrFZo5MY2bYYCEzpNDxUxs6n9SW
   scopes: openid, profile, email, offline_access
```

Step 3) **Controller**

```
@RestController
@RequestMapping("/authenticate")
public class AuthenticationController {

   @GetMapping("/login")
   public ResponseEntity<AuthenticationResponse> login(
           @AuthenticationPrincipal OidcUser oidcUser ,
```

```
            Model model,
            @RegisteredOAuth2AuthorizedClient("okta")
OAuth2AuthorizedClient client){

        AuthenticationResponse authenticationResponse =
                AuthenticationResponse.builder()
                        .userId(oidcUser.getEmail())

.accessToken(client.getAccessToken().getTokenValue())

.refreshToken(client.getRefreshToken().getTokenValue())

.expiresAt(client.getAccessToken().getExpiresAt().getEpochSecond()
)
                        .authorityList(oidcUser.getAuthorities()

.stream().map(GrantedAuthority::getAuthority)
                                .collect(Collectors.toList()))
                        .build();

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

Step 4) **authentication response**

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class AuthenticationResponse {

    private String userId;
    private String accessToken;
    private String refreshToken;
    private long expiresAt;
    private Collection<String> authorityList;
}
```

Step 5) **okta oauth websecurity**

```
@Configuration
@EnableWebFluxSecurity
public class OktaOAuth2WebSecurity    {

    @Bean
    public SecurityWebFilterChain
securityFilterChain(ServerHttpSecurity http){
        http
                .authorizeExchange()
                .anyExchange().authenticated()
                .and()
                .oauth2Login()
                .and()
                .oauth2ResourceServer()
                .jwt();

        return http.build();

    }


}
```

Adding Security in Order Service

Step 1) add dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>com.okta.spring</groupId>
  <artifactId>okta-spring-boot-starter</artifactId>
  <version>2.1.6</version>
</dependency>
```

Step 2) yaml

```
security:
 oauth2:
   resourceserver:
     jwt:
       issuer-uri: https://dev-32888642.okta.com/oauth2/default
   client:
     registration:
       internal-client:
         provider: okta
         authorization-grant-type: client_credentials
         scope: internal
         client-id: 0oa6q0ym0oqRPhY4b5d7
         client-secret: _TPXZSXK211ohMrFZo5MY2bYYCEzpNDxUxs6n9SW

     provider:
       okta:
         issuer-uri: https://dev-32888642.okta.com/oauth2/default
```

Step 3) web security class

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig {
   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
      http
            .authorizeRequests(authorizeRequest ->
authorizeRequest
                  .anyRequest().authenticated())

.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
      return http.build();
   }
}  }

}
```

Order service

Main file

```
@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {

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

  @Autowired
  private OAuth2AuthorizedClientRepository
oAuth2AuthorizedClientRepository;

  @Autowired
  private ClientRegistrationRepository
clientRegistrationRepository;

  @Bean
  @LoadBalanced
  public RestTemplate restTemplate(){
     RestTemplate restTemplate=new RestTemplate();
     restTemplate.setInterceptors(Arrays.asList(new
RestTemplateInterceptor(

clientManager(clientRegistrationRepository,oAuth2AuthorizedClient
Repository)
     )));

     return restTemplate;
  }

  @Bean
  public OAuth2AuthorizedClientManager clientManager(
     ClientRegistrationRepository clientRegistrationRepository,
     OAuth2AuthorizedClientRepository
oAuth2AuthorizedClientRepository){
     OAuth2AuthorizedClientProvider
oAuth2AuthorizedClientProvider
           = OAuth2AuthorizedClientProviderBuilder
           .builder()
           .clientCredentials()
```

```
            .build();

    DefaultOAuth2AuthorizedClientManager
oAuth2AuthorizedClientManager
            =new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository,
            oAuth2AuthorizedClientRepository
    );


oAuth2AuthorizedClientManager.setAuthorizedClientProvider(oAuth2A
uthorizedClientProvider);

    return oAuth2AuthorizedClientManager;

  }
```

Websecurity class

```
@EnableWebSecurity
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {

       http
               .authorizeRequests(authorizeRequest ->
authorizeRequest
                       .anyRequest().authenticated())

.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);

       return http.build();
   }

}
```

interceptor

```
@Configuration
public class OAuthRequestInterceptor implements
RequestInterceptor {


    @Autowired
    private OAuth2AuthorizedClientManager
oAuth2AuthorizedClientManager;

    @Override
    public void apply(RequestTemplate template) {
        template.header("Authorization","Bearer "
        +oAuth2AuthorizedClientManager
                .authorize(OAuth2AuthorizeRequest

.withClientRegistrationId("internal-client")
                            .principal("internal")
                            .build())
                .getAccessToken().getTokenValue());

    }
}
```

Rest template interceptor

```
package order.external.interceptor;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpRequest;
import
org.springframework.http.client.ClientHttpRequestExecution;
import
org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.http.client.ClientHttpResponse;
import
org.springframework.security.oauth2.client.OAuth2AuthorizeRequest
;
import
org.springframework.security.oauth2.client.OAuth2AuthorizedClient
Manager;
import org.springframework.web.client.RestTemplate;

import java.io.IOException;
```

```
public class RestTemplateInterceptor implements
ClientHttpRequestInterceptor {

    private OAuth2AuthorizedClientManager
oAuth2AuthorizedClientManager;

    public RestTemplateInterceptor(OAuth2AuthorizedClientManager
oAuth2AuthorizedClientManager){

this.oAuth2AuthorizedClientManager=oAuth2AuthorizedClientManager;
    }


   @Override
   public ClientHttpResponse intercept(HttpRequest request,
byte[] body, ClientHttpRequestExecution execution) throws
IOException {

        request.getHeaders().add("Authorization","Bearer
"+oAuth2AuthorizedClientManager

.authorize(OAuth2AuthorizeRequest.withClientRegistrationId("inter
nal-client")
                           .principal("internal")
                           .build())
                .getAccessToken().getTokenValue());


       return execution.execute(request,body);
   }
}
```

Adding Security in Product Service

Websecurity class

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {

        http
                .authorizeRequests(authorizeRequest ->
authorizeRequest
                        .anyRequest().authenticated())

.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);

        return http.build();
    }
}
```

Controller

```
@RestController
@RequestMapping("api/product")
public class ProductController {

    @Autowired
    private ProductService productService;

    @PostMapping("/")
    @PreAuthorize("hasAuthority('Admin')")
    public ResponseEntity<Long> addProduct(@RequestBody
ProductRequest productRequest){
        Long productId = productService.addProduct(productRequest);
        return new ResponseEntity<>(productId, HttpStatus.CREATED);
    }
```

```
    @GetMapping("/{id}")
    @PreAuthorize("hasAuthority('Admin')  ||
hasAuthority('Customer') || hasAuthority('SCOPE_internal')" )
    public ResponseEntity<ProductResponse>
getProductById(@PathVariable("id") Long productId){
        ProductResponse productResponse =
productService.getProductById(productId);
        return new ResponseEntity<>(productResponse,HttpStatus.OK);
    }

    @PutMapping("/reduceQuantity/{id}")
    public ResponseEntity<Void> reduceQuantity(@PathVariable("id")
Long productId, @RequestParam long quantity){
        productService.reduceQuantity(productId,quantity);
        return new ResponseEntity<>(HttpStatus.OK);
    }

}
```

Pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>com.okta.spring</groupId>
  <artifactId>okta-spring-boot-starter</artifactId>
  <version>2.1.6</version>
</dependency>
<dependency>
```


Adding Security in Payment Service

Yaml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>com.okta.spring</groupId>
  <artifactId>okta-spring-boot-starter</artifactId>
  <version>2.1.6</version>
</dependency>
```

Websecurity class

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

   @Bean
   SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
       http
               .authorizeRequests(authorizeRequests ->
authorizeRequests.antMatchers("/api/payment/*")
                       .hasAuthority("SCOPE_internal")
                       .anyRequest()
                       .authenticated()
               )

.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);

       return http.build();

   }
}
```


Unit Testing

Step 1) dependency add - Junit
Junit comes by default when we generate spring boot projects.

This junit is the junit5 , that is junit jupiter dependency.

We should create junit for all the functionality that we have created.

Under Junit we will use the **Mockito** framework(library) and **wiremock** library to create test cases in our application.

Mockito - used to mock the method .

Mock → if you writing test case for getting order , within this getting order functionality we have multiple calls to database and other services as well.
But here we just want to test getting order func.only , we are not trying to test the other calls that its been doing from that particular method for that either they are getting call from db or from other rest point we mimic those scenarios programmatically
Rather than directly calling the database method what we tell them is wehenver we calling this method return that mock object .
We are just mimicking the behaviour and passing the data
Rather than calling actuatl method it will mimic that .

Wiremock - when we want to mimic the rest end point we use wiremock
In mockito we mimic method while in wiremock we mimic endpohnt

We will create two type of test cases - unit test for order service (whaterver method are in order service we will do test case for that)
Order controller - for entire controller we will write integrate test case for endpoints using wire mock

To added test case we need junit and mockitp library which is there default in springoboot project

Other library we need is wiremock
Spring security test library also need
Also to store test case data we use h2 in memory database

```
<dependency>
  <groupId>com.github.tomakehurst</groupId>
  <artifactId>wiremock-jre8</artifactId>
  <version>2.33.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>
```

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Now we have added resource directory in test scope and crate applicairon .yaml for test case condfig

Yaml config

```
server:
 port: 9091

#database config
spring:
 config:
   import: optional:configserver:http://localhost:9296
 datasource:
   driver-class: org.h2.Driver
   username: sa
   url: jdbc:h2:mem:order
   password: password

 #jpa config
 jpa:
   database-platform: org.hibernate.dialect.H2Dialect

   #eureka config

 security:
   oauth2:
     resourceserver:
       jwt:
         issuer-uri: https://dev-32888642.okta.com/oauth2/default
     client:
       registration:
         internal-client:
           provider: okta
           authorization-grant-type: client_credentials
           scope: internal
           client-id: 0oa6q0ym0oqRPhY4b5d7
```

```
            client-secret: _TPXZSXK21lohMrFZo5MY2bYYCEzpNDxUxs6n9SW

        provider:
          okta:
            issuer-uri:
https://dev-32888642.okta.com/oauth2/default

eureka:
 client:
    enabled: false
```

Docker

Docker is a way to package all our applications and all the related libraries and whatever thing that we can use within our applications into a single bundle.

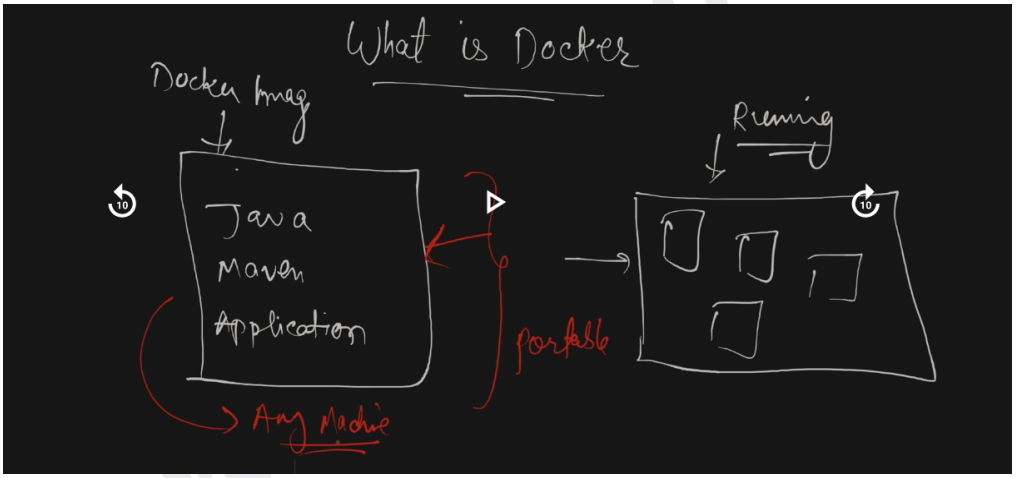
We can share all the things into a single bundle to others as a docker image rather than giving them each one separately.

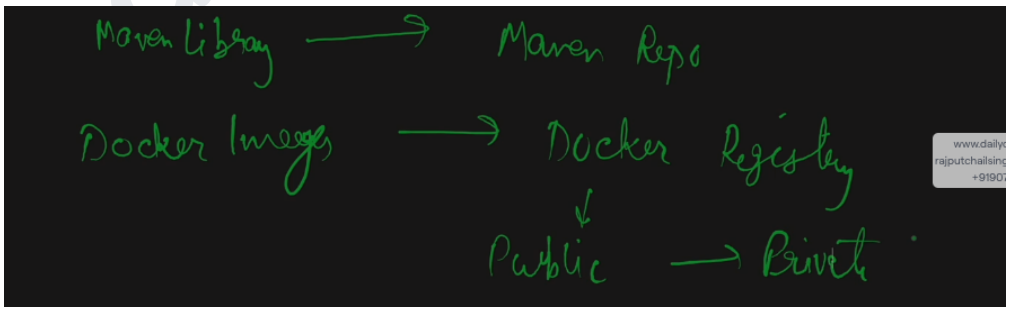
We can take docker images and go to any machine and we can deploy this application without worrying about the configuration that much of portable docker is.

Images live in the container registry just like the maven library lives in the maven repository.

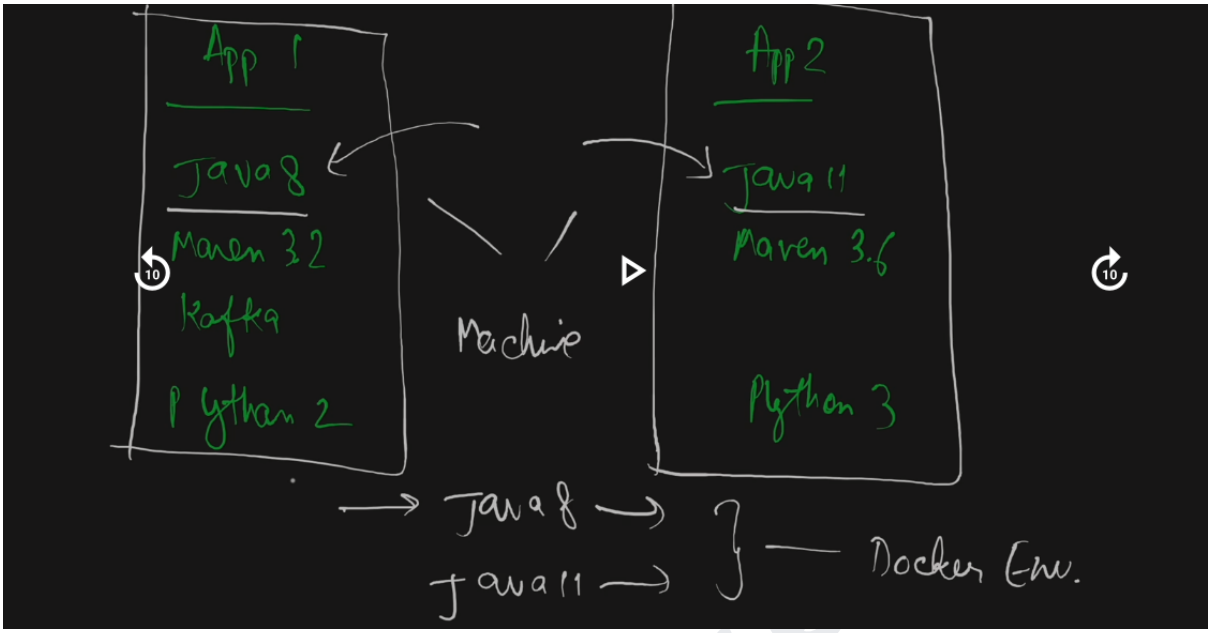
All those docker images there will be in docker registry , registry can be public and private both.

By default docker gives a public docker hub registry where all images live.
We can create a private registry as well.





Before docker , let's have two applications with different configurations so it will be difficult to manage in the system but using docker it is easy to configure because all the images are available in the docker environment.

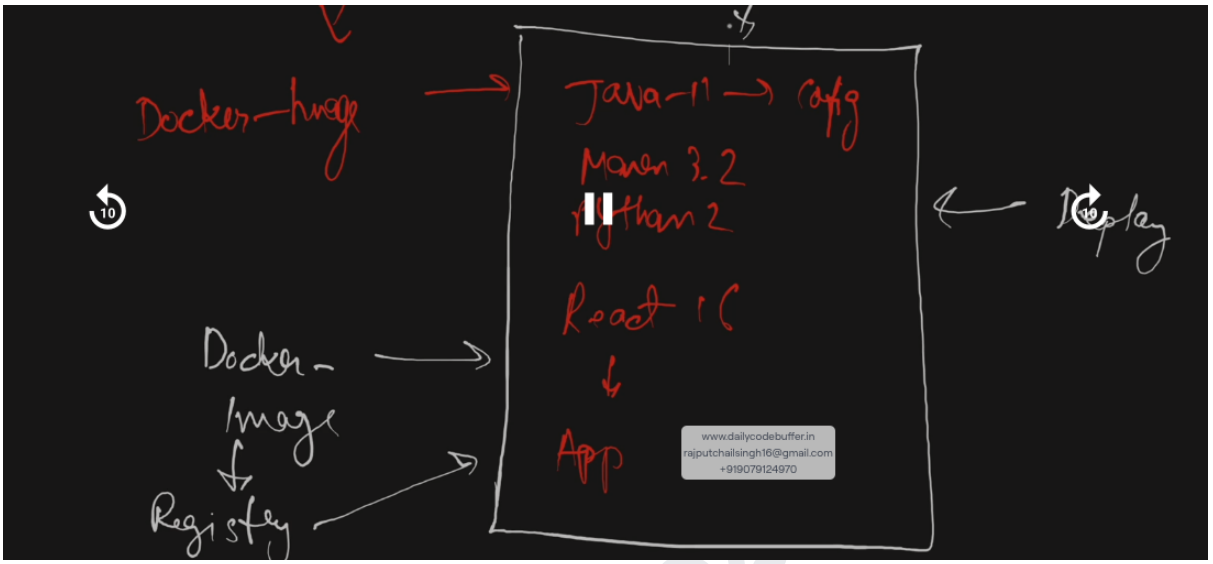


So it will be very easy when you are working with multiple environments using docker.
For eg.
I am a new joinee in the company and if I need to start working on an application .
I will have to configure all requirements , that would be a hectic process so rather if i get a docker image which combines all the requirements , it will be very easy to work on.
Other than development we do alot of deployment . we moved our application to a different environment so for that as well docker is very useful.

Before Docker, whenever we have deployed the application , as a developer I need to deploy into any of the environments so I need to connect with the infrastructure team or devops team to deploy applications.
And I will give them all the requirements to be configured.
Now there might be a chance that some of the configuration misinterpreted that would create problems and for multiple environments we need to make sure proper SOP’s and Documentation are maintained.

What if we are using docker

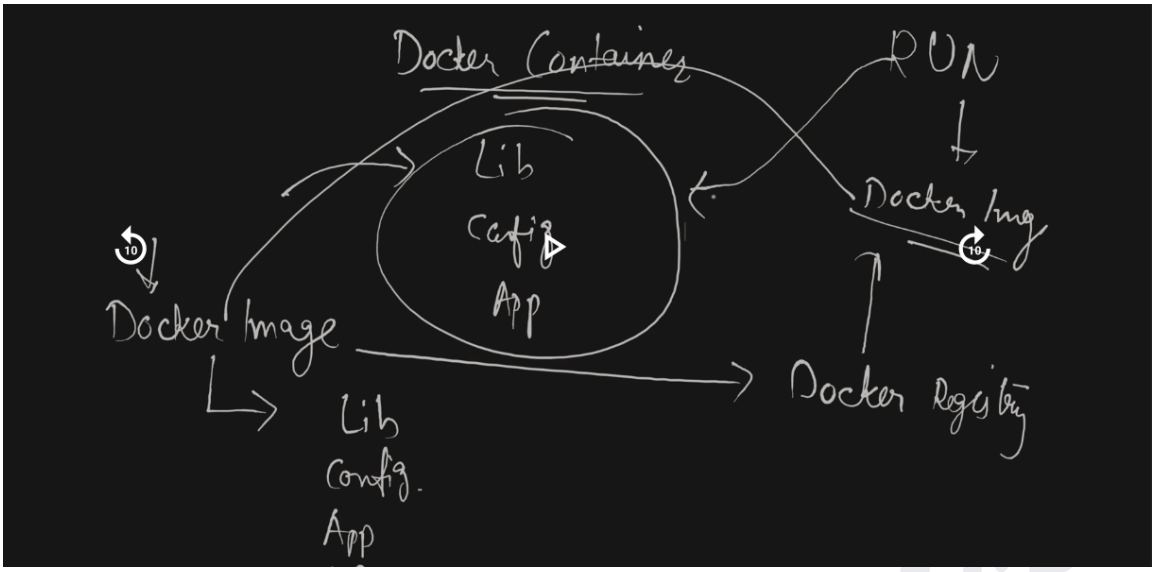
We can package each and every requirement into a single image .
This docker image is created , we can keep this in the registry and wherever we want , we can directly take this image and deploy easily.



Docker Container

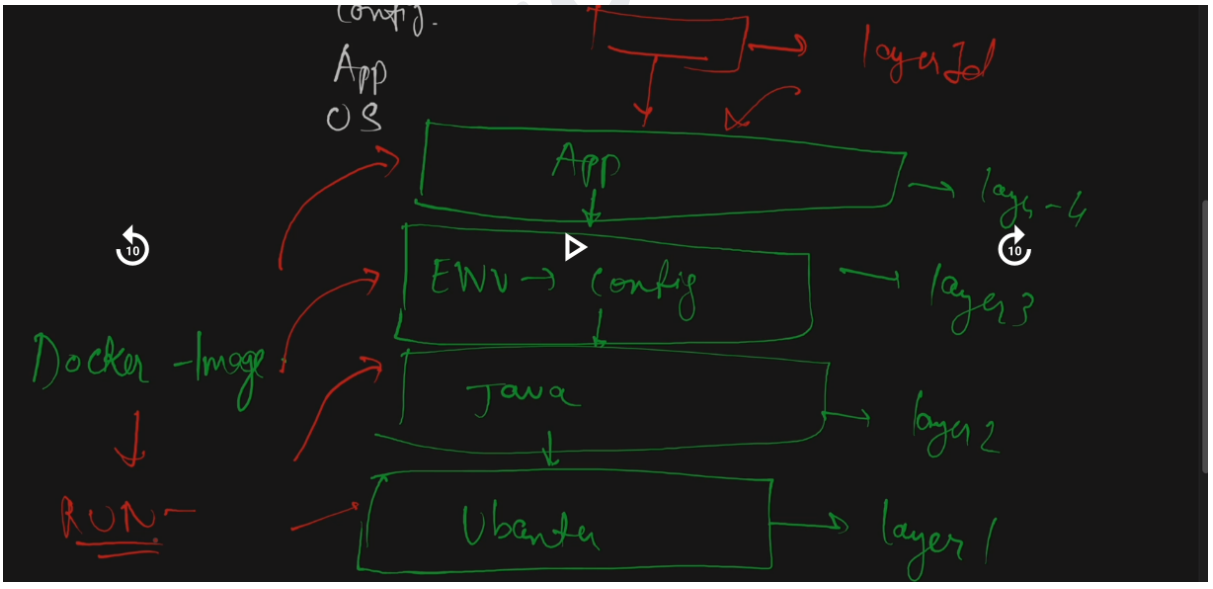
Docker container is the package itself which includes each and everything with all the libraries , configuration and applications.
Everything combines together and forms the docker container.
And this docker container will be running on the docker environment.
Docker container is created based on the docker image.

There will always be docker images which consist of all the libraries , configuration and applications.
This image lives in the docker registry and in whichever environment our application is running in that environment we can take this image from the registry and we can run this image.
Whenever we can run this docker image that is called container.
Running entity of docker is called container.



Every docker image consists of different layers , whatever configuration we added into the image it created different layers.And at the end you will get the application layer at the top and you will get layer id as well.

Those layers combined together will make a single docker image and that image we can run.
And that is going to be called docker container.
That single entity will be running on the docker environment.



Steps to Install Docker

Remove pre-installed docker → `sudo apt remove docker-desktop`

For a complete cleanup, remove configuration and data files at `$HOME/.docker/desktop`, the symlink at `/usr/local/bin/com.docker.cli`, and purge the remaining systemd service files.

```
$ rm -r $HOME/.docker/desktop
$ sudo rm /usr/local/bin/com.docker.cli
$ sudo apt purge docker-desktop
```

Install Docker Desktop

Recommended approach to install Docker Desktop on Ubuntu:

1. Set up Docker's package repository.
2. Download the latest DEB package.
3. Install the package with apt as follows:

```
$ sudo apt-get update
$ sudo apt-get install ./docker-desktop-<version>-<arch>.deb
```

-

Launch Docker Desktop

To start Docker Desktop for Linux, search **Docker Desktop** on the **Applications** menu and open it. This launches the whale menu icon and opens the Docker Dashboard, reporting the status of Docker Desktop.

Alternatively, open a terminal and run:

```
$ systemctl --user start docker-desktop
```

After you've successfully installed Docker Desktop, you can check the versions of these binaries by running the following commands:

```
$ docker compose version
Docker Compose version v2.5.0

$ docker --version
Docker version 20.10.14, build a224086349
```

```
$ docker version
Client: Docker Engine - Community
Cloud integration: 1.0.24
Version:           20.10.14
API version:       1.41
...
```

To enable Docker Desktop to start on login, from the Docker menu, select **Settings** > **General** > **Start Docker Desktop when you log in**.

Alternatively, open a terminal and run:

```
$ systemctl --user enable docker-desktop
```

To stop Docker Desktop, click on the whale menu tray icon to open the Docker menu and select **Quit Docker Desktop**.

Alternatively, open a terminal and run:

```
$ systemctl --user stop docker-desktop
```

Uninstall Docker Desktop

To remove Docker Desktop for Linux, run:

```
$ sudo apt remove docker-desktop
```

For a complete cleanup, remove configuration and data files at `$HOME/.docker/desktop`, the symlink at `/usr/local/bin/com.docker.cli`, and purge the remaining systemd service files.

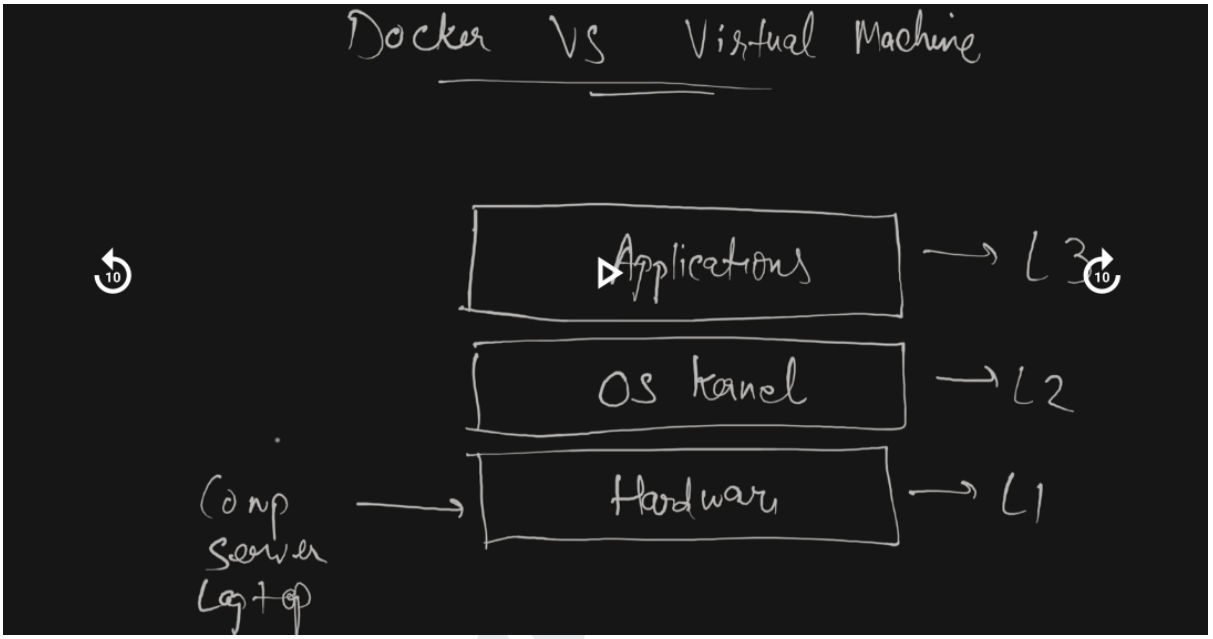
```
$ rm -r $HOME/.docker/desktop
$ sudo rm /usr/local/bin/com.docker.cli
$ sudo apt purge docker-desktop
```

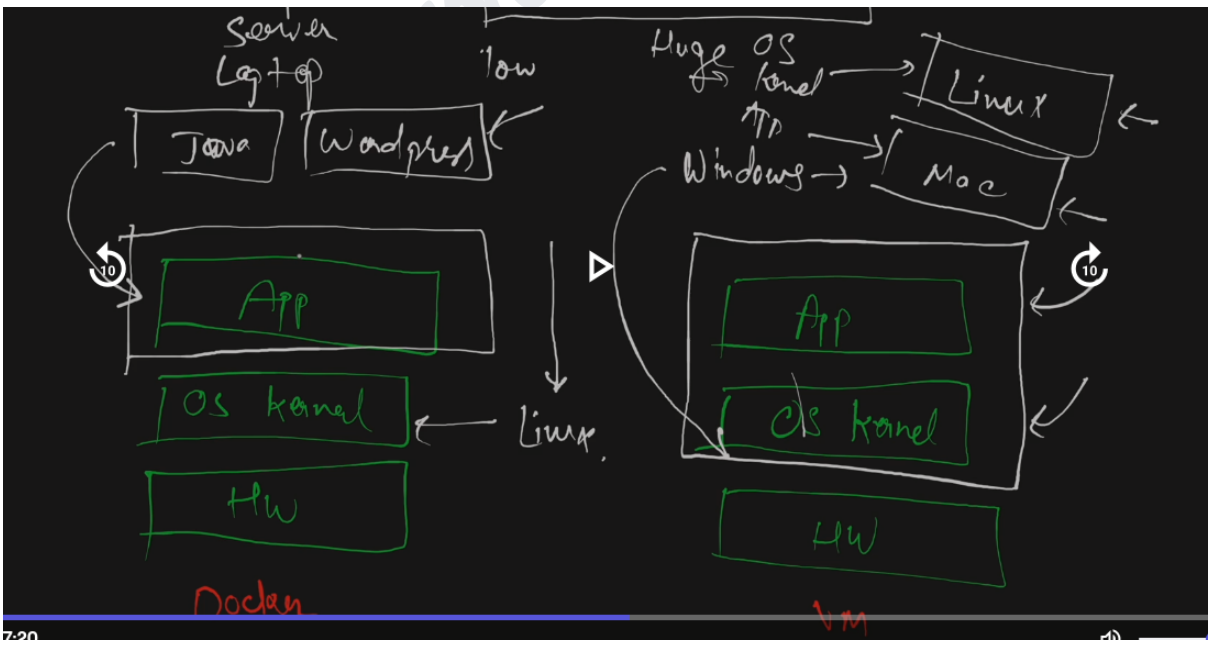
Docker vs Virtual Machine

Both docker and virtual machines provide us virtualization to use for our applications.

Different layers that we get for the operating system.

1) Hardware layer - bottom layer
2) Kernel layer
3) Application layer - top[layer





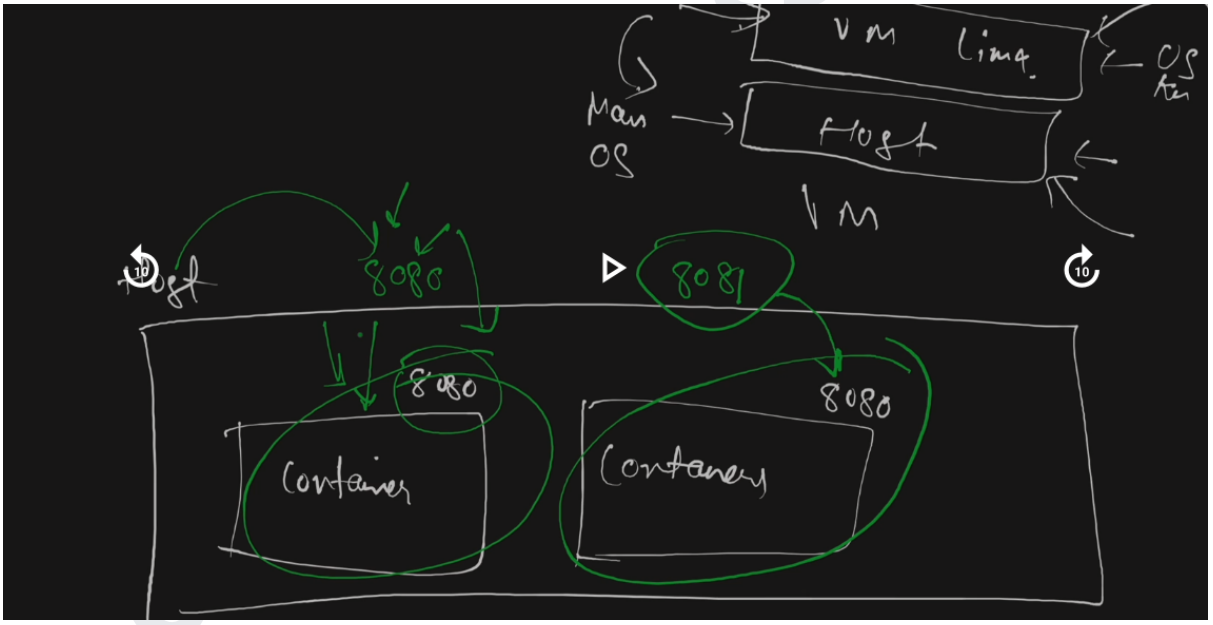
Virtual machines will create a layer of application + os kernel on top of the host machine
So if you have windows installed in your machine , there are different applications that allow us to create virtual machines like oracle virtualbox , vmware.

It allows us to install different os all together . If I want to install linux or mac on the top of the windows it will allow using a virtual box. We can do it because it consists of os kernel and application level both.

But docker will use one application layer only , it will use the same os kernel level and on top of that it will allow us to create different virtual environments For the different applications .

If my OS and kernel is linux then i can only use docker images that are based on linux environments.

That's why the size of the images is less because it does not contain any operating system it will use the same os. Whereas virtual machines use separate os so size is huge.



In virtual → the host machine and virtual machine (linux) have their different kernel and os and we can access the host machine using host and we can access virtual machine using the virtual machine environment.

With docker → we have the host environment And within host env we have different containers deployed .

Every container will have their own ip address and port number .

Every container is isolated from each other so both containers can run on the same port because they are isolated .But for the host machine it can have a different machine .
Hosts can't have multiple port numbers so here comes the host port and internal docker port concept .

I can define the configuration in such a way that my container which is run on port 8080 will access from my host port 8080.

But for container 2 it also have 8080 port but it can't access because host with port 8080 already in use so here i can define 8081 as a host port and from this port i want to access the container with port 8080

Different Commands of Docker

- docker pull redis → download redis images
- docker images → show all images
- docker run - -name redis-latest redis:latest **or** docker run –name redis-latest imageId
- docker ps → show all the running container
- docker stop containerId **or** docker stop container name → stop the container
- docker ps -a → show all the container (stop and run all)
- docker start containerId **or** docker start container name → start the container
- docker stop containerId
- docker rm containerId or docker rm container name → to remove first stop container
- docker images
- docker run –name redis-latest -p6379:6379 -d redis:latest → hostport:containerport
- docker pull redis:6.2.7 → different version of same container
- docker images
- dockerdocker images -a run - -name redis-old -p6378:6379 -d redis:6.2.7 → host port different only
- docker –help
-
- docker inspect imageId → details about image
- docker inspect containerId → details about container
- docker logs containerId → logs of container
- docker exec -it containerId /bin/sh → go into container and execute the command
- docker exec -it containerId /bin/bash
- docker exec -it containerId /bin/gsh
- ls
- pwd
- cd ..
- ls
- redis-cli
- KEYS *
- Exit → exit from container

- ls

Deleting all docker resources

- Docker ps
- Docker stop containerID1 containerId2 ..
- Docker ps -a
- Docker images
- Docker rmi imageId → remove the image
- Docker system prune -a → remove everything in one go

Building Docker Images

We are creating our own docker images using docker file for our application.

Create Dockerfile in root directory of service registesry

I want my image based on linux and it has java install in it
FROM Baseimg → FROM openJdk:11
Now i want to add jar file
Build application and jar fil ecrated
Copy the jar file inside imag and that jhar file start when i start fontaienre

Mvn clean install
Now i have to get this jar file and add in my image
I will refer this file ini one of the avribale
And that i will use rto define d

Docker file

```
FROM openjdk:11

ARG JAR_FILE=target/*.jar

COPY ${JAR_FILE} serviceregistry.jar

ENTRYPOINT ["java","-jar","/serviceregistry.jar"]

EXPOSE 8761
```

Create image -> Docker build -t chail/serviceregistry:0.0.1 .

Running docker images

Docker run -d -p8761:8761 –name serviceregistry imageId

Now containe start run
Go to browe localhost:8761
Run

sudo docker run -d -p9296:9296 -e
EUREKA_SERVER_ADDRESS=http://host.docker.internal:8761/eureka --name configserver
87a1914c1771
Docker logs containerId
Login to docker

Docker login -u username -p password
Docker push repositoryinfo
Docker push chail/service:latest

Docker Compose

Currently we hava multiple images available and for start container for all those images we did different commands and those used separately and all those command has different sets of env.and port info and everything.
So you have to handly all those command.
For start multiple images you have to go through with many commands.
There is a way where you can define each and every steps to do all those thing into single go .
You just need to configure into yaml file and by running that single yaml file it will do all you work in single go.

Run command → docker system prune -a → clean everything

Step 1) create all images for all microservices
Step 2) create docker compose file → go to project location in the system and run this command
touch docker-compose.yml
Step 3) config into this yml file
Version info —--- version: ‘1’
Services —--- services:

```
    Serviceregistry:
      Image: ‘chail/serviceregistry:0.0.1’
      Container_name: serviceregistry
      Port:
        -‘8761:8761’

    Configserver:
      Image: ‘chail/configserver:0.0.1’
      Container_name: configserver
      Port:
        -9296:9296’
      Environment:
            -EUREKA_SERVER_ADDRESS=http://serviceregistry:8761/eureka
      healthcheck :
          Test: [“CMD” , “Curl” , “-f” , “http://configserver:9296/actuator/health” ]
          Interval: 10s
          Timeout: 5s
          Retries: 5s
      Depends_on :
          - serviceregistry

    cloudgateway:
     Image: ‘chail/cloudgateway:0.0.1’
     Container_name: cloudgateway
      Port:
        -9090:9090’
     Environment:
```

```
                -EUREKA_SERVER_ADDRESS=http://serviceregistry:8761/eureka
              -CONFIG_SERVER_URL=configserver
        Depends_on :
            - configserver:
                Condition: service_healthy
```

Now run yml file in one go

Run all services
→ docker-compose -f docker-compose.yml up -d

Delete all container
→docker-compose -f docker-compose.yml down

DOCKER MAVEN JIB PLUGIN

It allows us to package applications into docker images.

Setup

In your Maven Java project, add the plugin to your `pom.xml`:

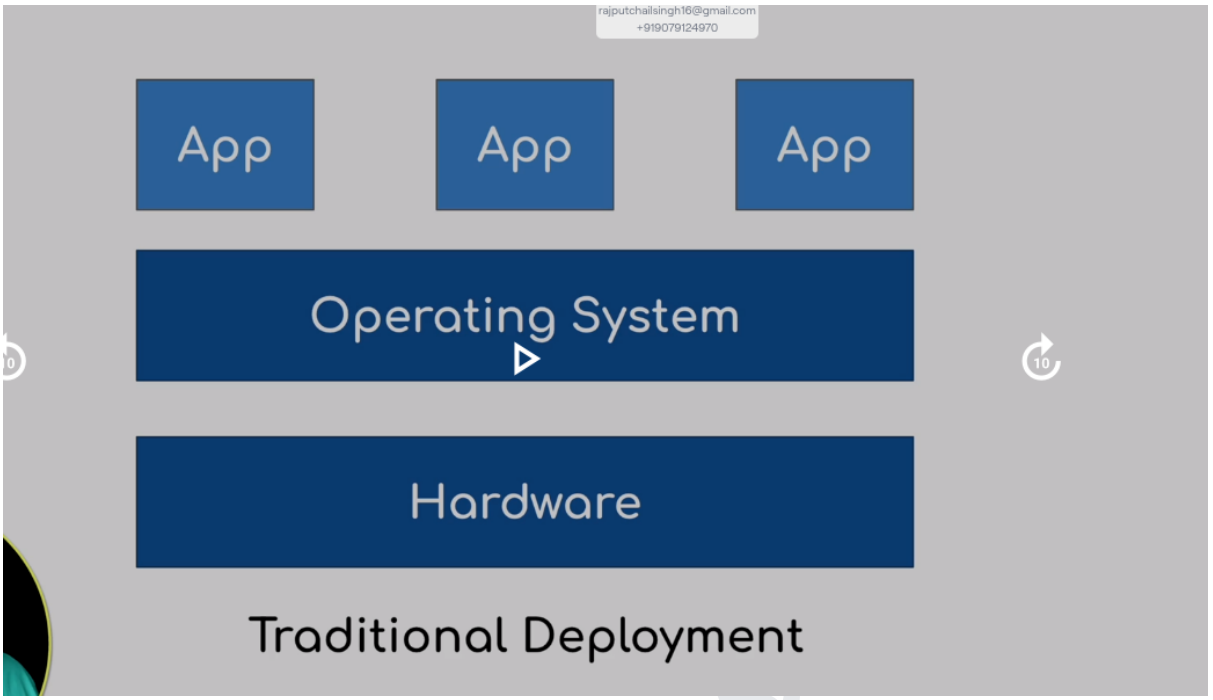
```
<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>jib-maven-plugin</artifactId>
  <version>3.3.0</version>
  <configuration>
      <from>
          <image>openjdk:11</image>
      </from>
      <to>

<image>registry.hub.dokcer.com/chail/serviceregistry</image>
          <tags>${project.version}</tags>
      </to>
  </configuration>
</plugin>
```

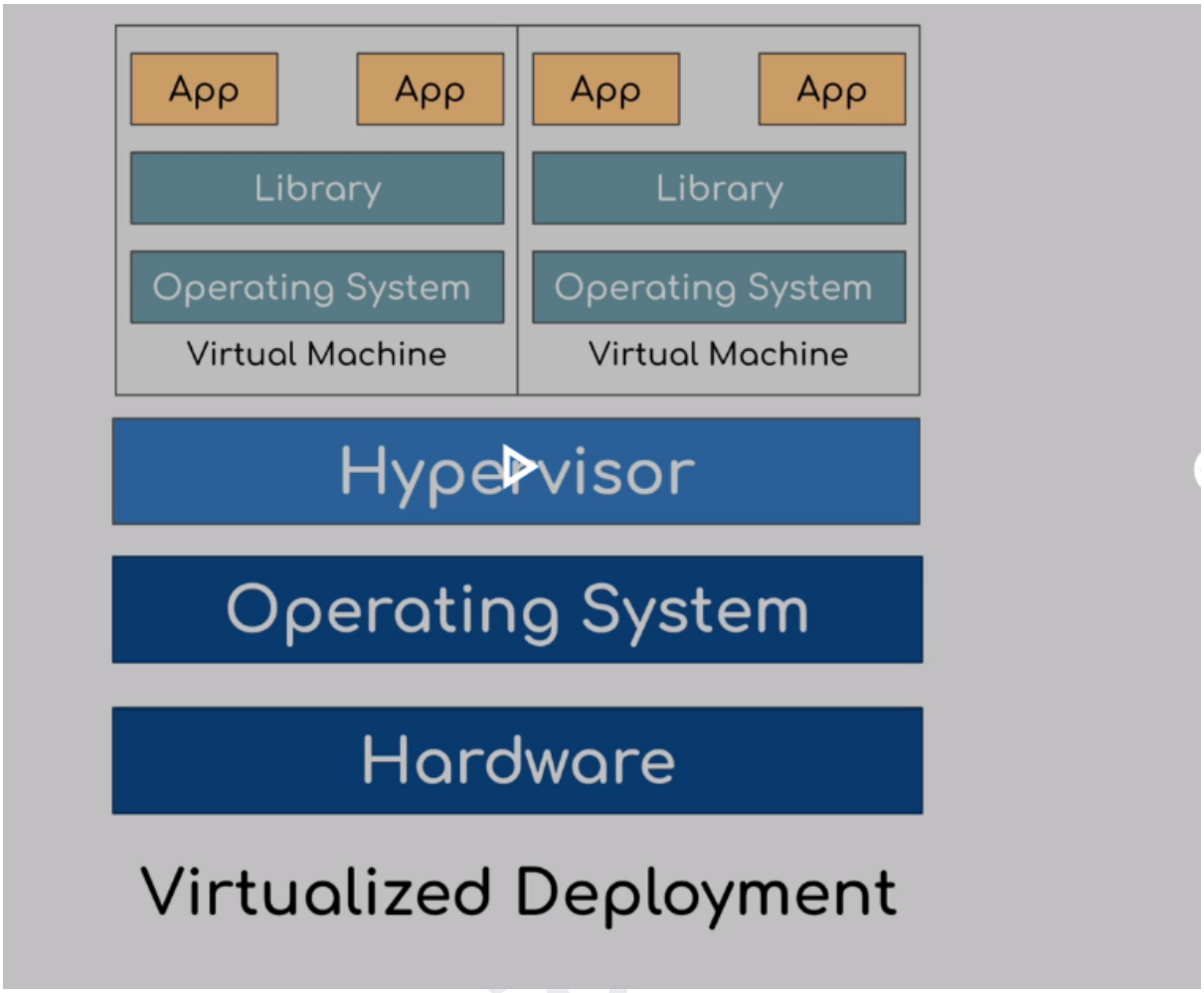
Set credential of docker into setting.xml file

```
<xml version=”1.0” encoding=”UTF-8”?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0
http://maven.apache.org/xsd/settings-1.1.0.xsd">
    <servers>
        <server>
            <id>registry.hub.docker.com</id>
            <username>//USERNAME//</username>
           <password>//PASSWORD// <password>
       <server>
  <servers>
<settings>
```

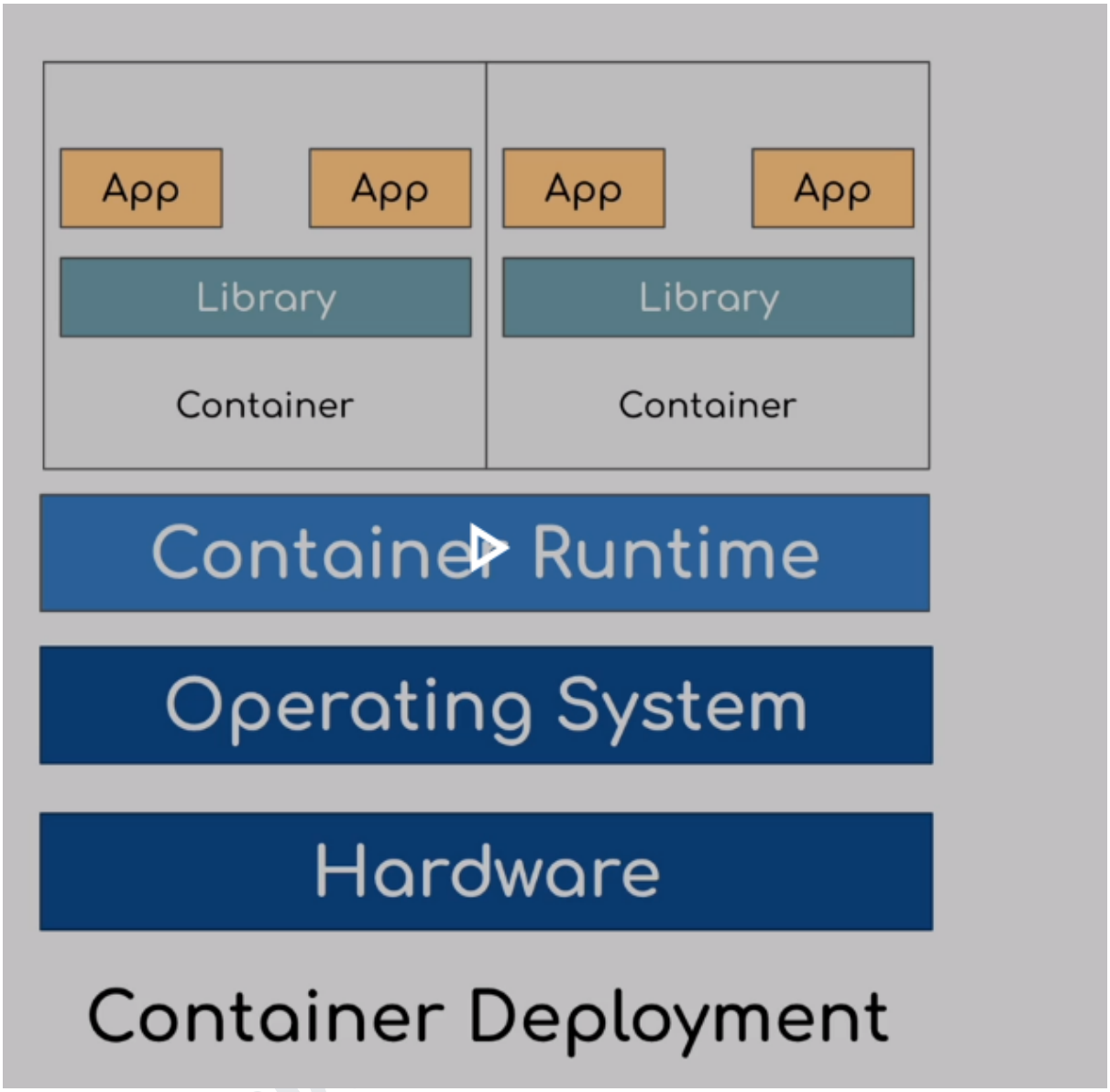
Now rebuild project → mvn clean install jib:build
It will create jar file and create docker image and push to docker hub automatically

KUBERNETE





Virtualized Deployment

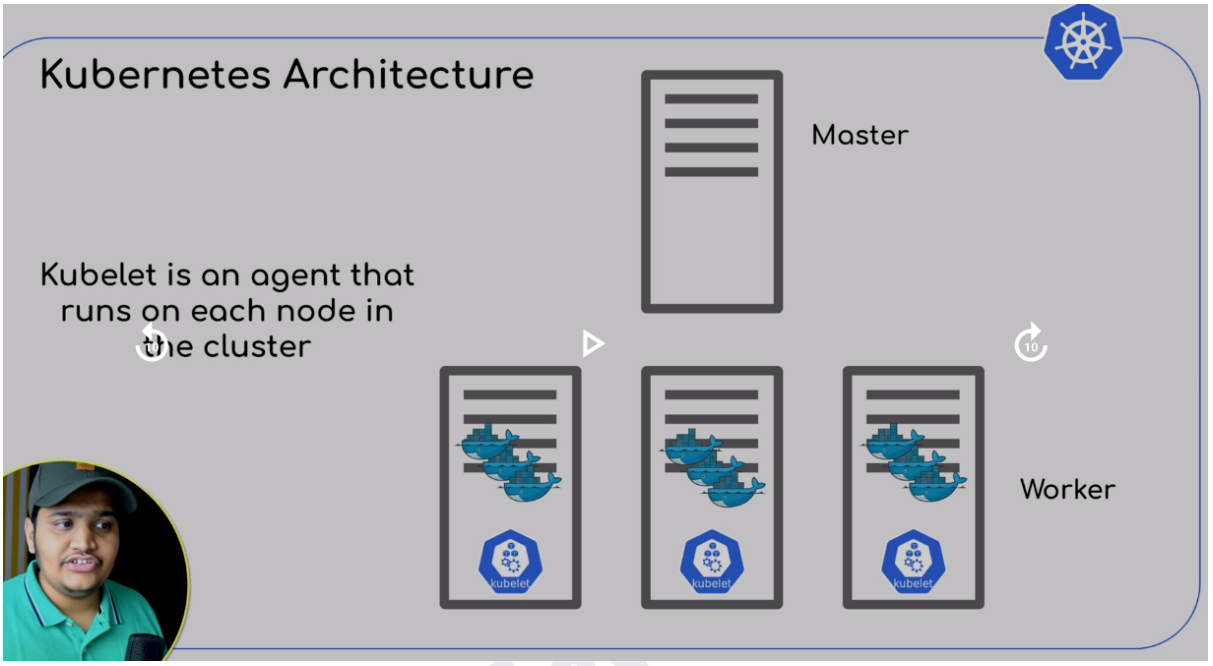


Using docker we have created a container and those containers we were having our application deployed.
Now suppose we have a huge application and in our env.we need to deploy hundreds of containers and to maintain the entire lifecycle of a container by doing it manually is difficult.

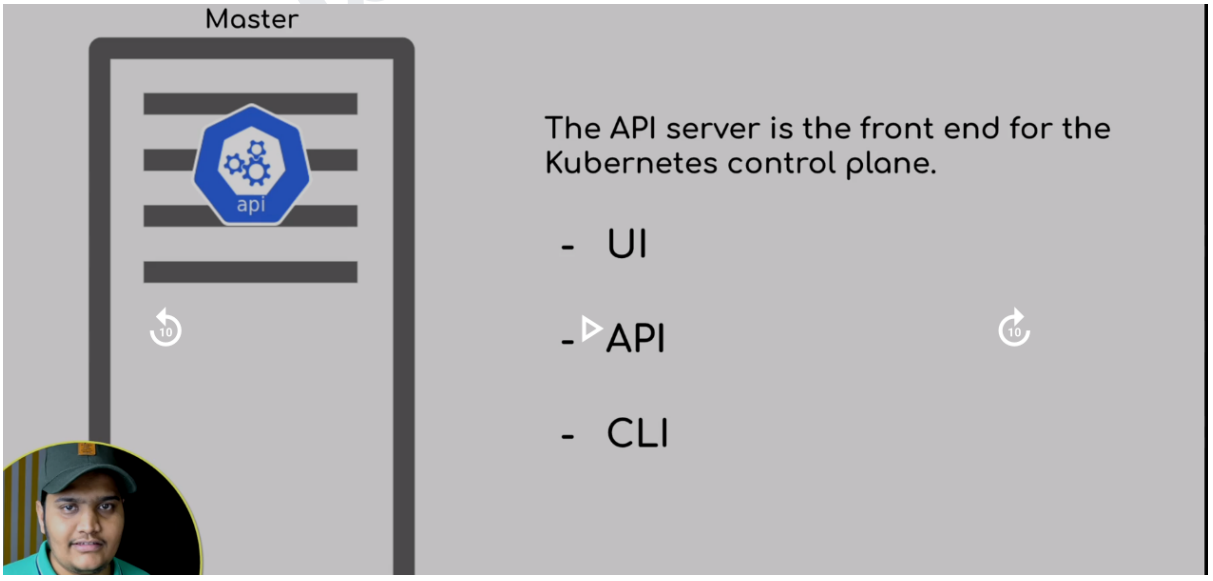
There is a kubernate tool which allows us to manage all by default.

Kubernetes Architecture cluster

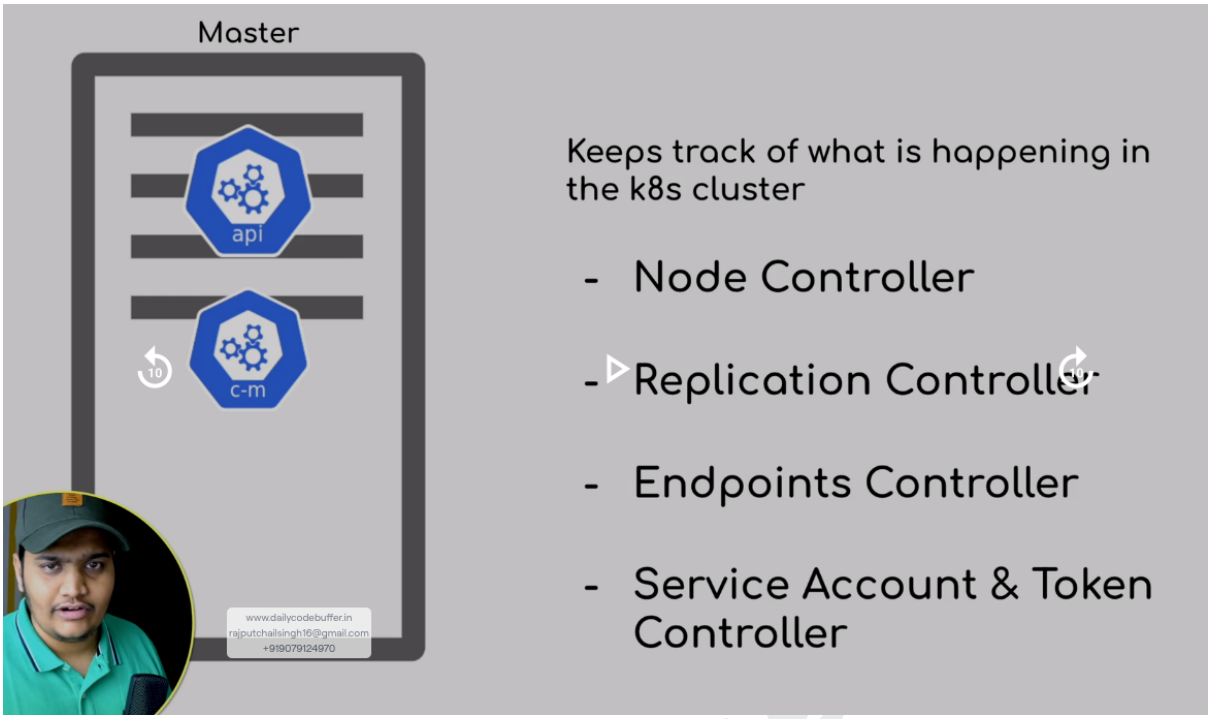
Master node - control everything , there should be atleast one master node
Worker node - work according to master , deploying container in it
With the container there is a kubernate component that is kublet which run on each and every node and make sure it has all the set of proper information and setup based on the master node.



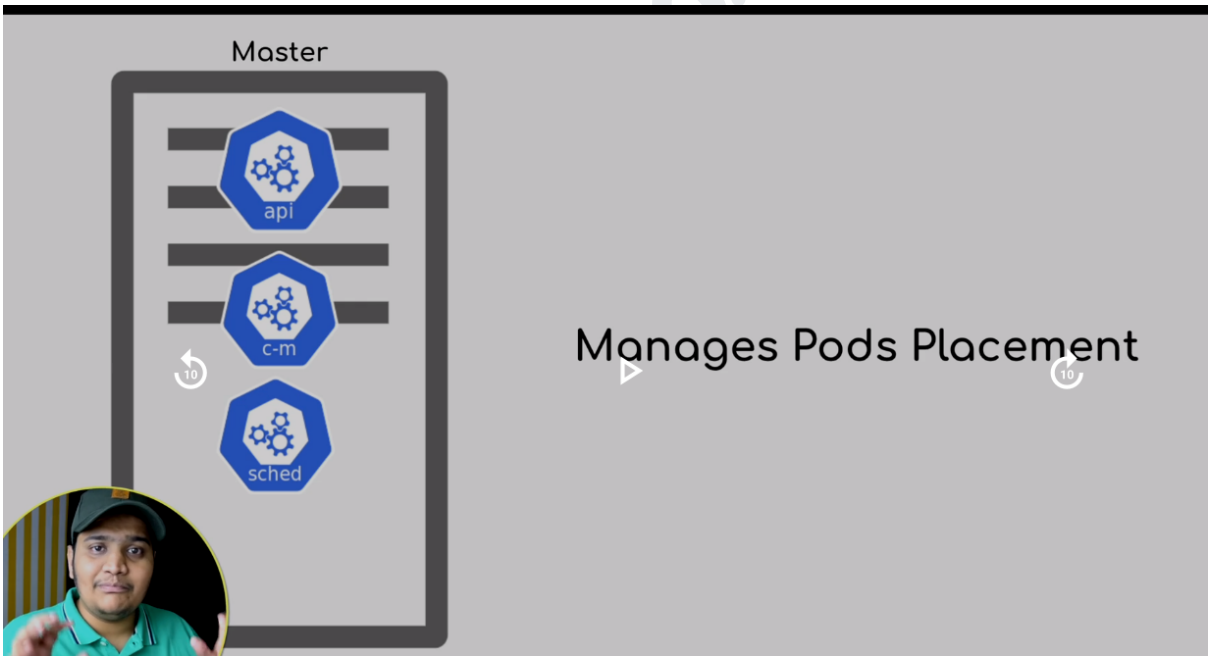
Master node will have API Component
We have access of this api server using ui,api,cli tool.
Api server is the front end for the kubernarte control plane



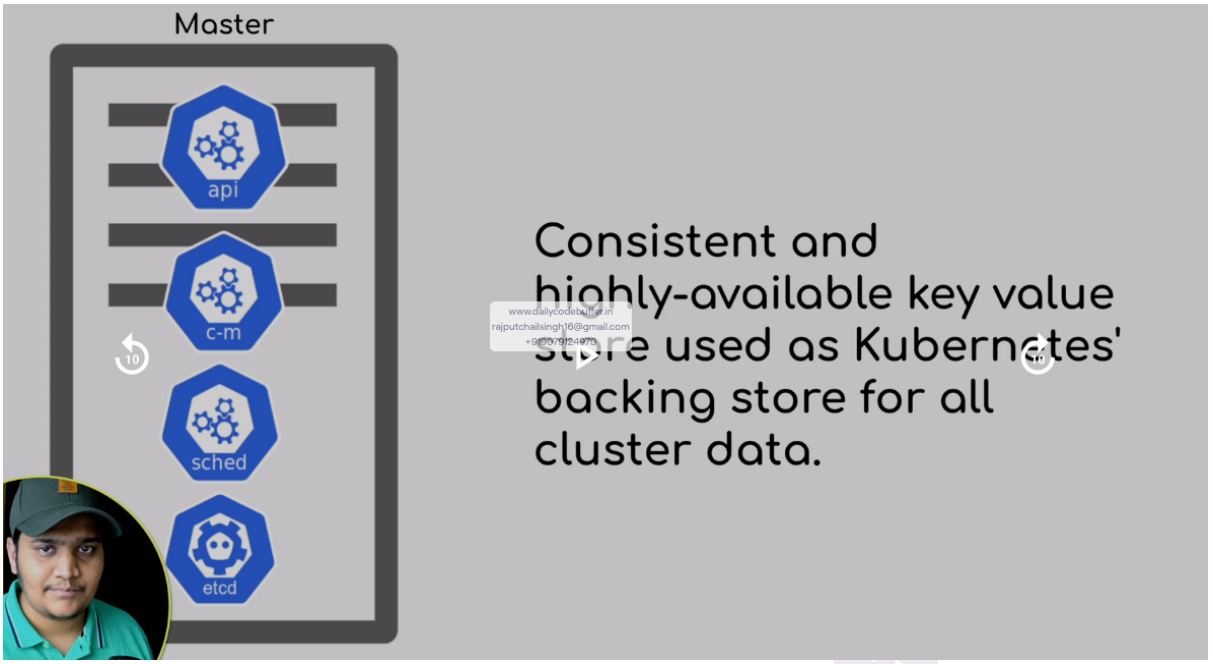
Next component is the control manager



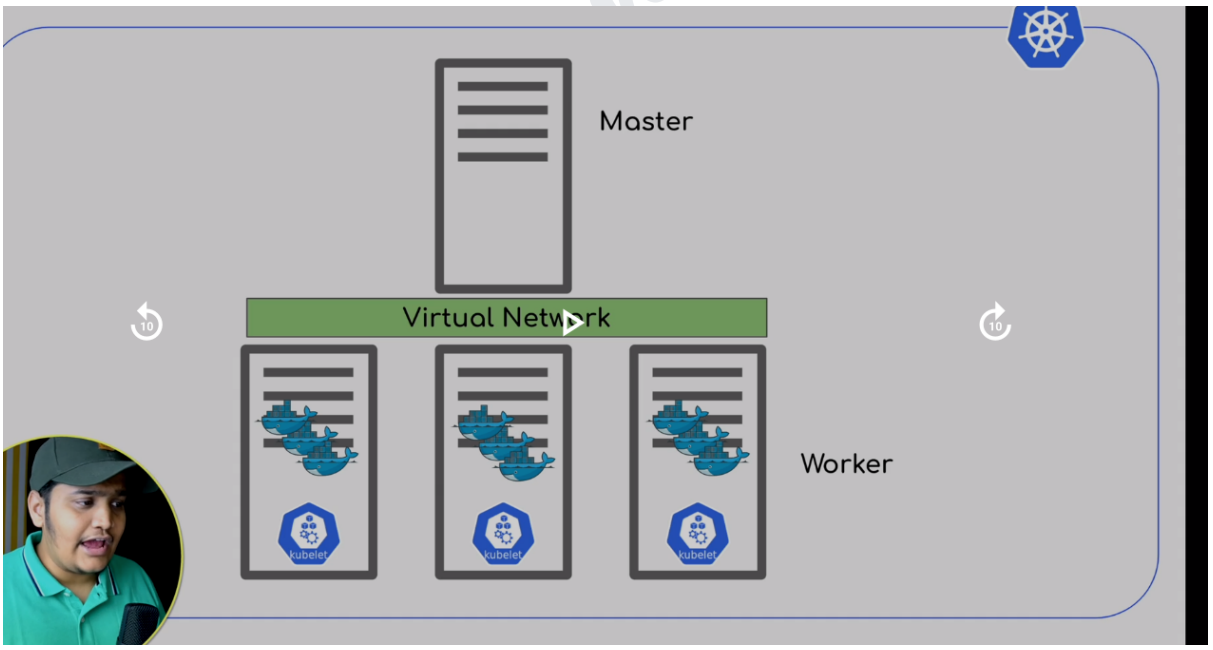
Next is scheduler



Next is etcd component



Master node and worker node have their own virtual network
Everything happens in that network within the cluster and they can interact each other .



Worker Node

Worker node will having pod component and pod is the single unit deployable component in the kubernate cluster.
Anything we will be doing in the kubernate cluster we will doing using the pod .
Whenever i want to depoy container or application within the cluster i need to wrap around with the pod and that i can deploy and
whenever i will create pod with the container it will assign an ip address to it so each and every pod will have its own ip address.
As These ip address are dynamic so we need component which can have static ip so different component can connect to it so for that reason we have service component
So for multiple pod we can create service component