Maven : (Project Management Tool) {External libraries}

↳ Compile, Run Tests, Packaging & Deploying

Maven Archetype : (Templating tool)
for web project

Dependencies (Libraries) : JAR file.

mvnrepository.com → Searching dependencies websites

pom.xml → to make changes in this file.
for adding other dependencies.

dependencies
GAV
Groupid
Artifact Id (project name)
version

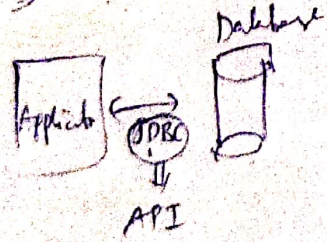
<dependencies>
<groupId> com.mysql </groupId>
<artifactId> mysql-connector-j </artifactId>
<version> 8.1.0 </version>
</dependencies>

Effective POM : (It is hidden)
Which is get created in Maven project

→ First it search in .m2 folder (hidden) in our system
→ If not there, it will go to Central.

JDBC : (Java database Connectivity) :

↳ To store data in Tabular format
RDBMS.



→ For connecting Application to Database using JDBC we need "JAR files"

Step-① : Import packages (java.sql)

Step-② : Load the driver (Coming from JAR file)

Step-③ : Register driver.

Step-④ : Create Connection

Step-⑤ : Create Statement

Step-⑥ : Execute statement

Step-⑦ : Close.

```
import java.util.sql.* ; //importing packages

public class DemoJDBC {
    psvm (      ) throws Exception {
                                  3306
                              mysql/oracle    //localhost:5432/demo"
        String url = "jdbc: postgresql = demo" ;
                                              databasename
    "jdbc:postgresql://localhost:5432/demo"
        String uname = "postgres"; //default username
        String pass = "kundan@04";
        String query = "select sname from student where sid=1";
    Class.forName("org.postgresql.Driver"); // load and register driver.
 (optional)
    Connection con = DriverManager.getConnection(url, uname, pass);
                                                          //Connection.

    Statement st = con.createStatement(); //create statement

ResultSet rs = st.executeQuery(query); // execute statement.
        rs.next();
        sop(rs.getString("sname"));
                           col_name.

    con.close() // closing the connection
```

String sql = "select * from student";

↳ It will check whether there is a next row.

while (rs.next()) {

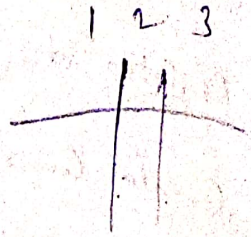
sop (rs.getInt(1));

sop (rs.getInt(2));

sop(rs.getString(3));

}

↳ Column number



CRUD operations:

(create)

String sql = "insert into student values(5, 66, 'John')";

If it is select query it returns resultset const.

insert/update

boolean status = st.execute(sql); // returns boolean. (insert, update/delete)

Update:

String sql = "update student set sname = 'Max' where sid = 5";

st.execute(sql); //

delete:

String sql = "delete from student where sid = 5";

st.execute(sql);

Problems with Statement: → Tony confusion of double & single quotes

→ This values are coming from users, it have a chance of SQL Injection

int sid = 101;

String sname = "Max";

int marks = 98;

String sql = "insert into student values('" + sid + "', '" + sname + "', " + marks + ")";

To overcome we introduce "Prepared Statement"

String sql = "insert into Student values (?, ?, ?)";

CallableStatement → Stored procedures

PreparedStatement st = con.prepareStatement(sql);

st.setInt(1, sid); → Col_number, data

st.setInt(2, marks);

st.setString(3, sname);

→ Enter in Order (which is in table)

Spring (POJO)

Pre-requisites: Java, DB connectivity (JDBC), Servlet, JSP, Maven, ORM (Hibernate)

Spring Core → Dependencies

Spring Boot ⇒ Convention over Configuration.

It give basic configuration

Dependency Injection = Inversion of Control (IoC)

Belongs to Application context.

Alien obj = getBean(Alien.class);

Spring Code:

class name

public {

ApplicationContext context = SpringApplication.run(FirstSpringApp.class, args);

Alien obj = context.getBean(Alien.class);

Spring is responsible for this class object

obj.code();

}

}

Alien.java

@Component

It indicates that Spring framework will responsible to create Alien object

public {

func() {

sop("code");

}

@ Autowired :

```
public class Alien
{
  @Autowired // we are not creating Object of laptop, it
  Laptop lap;                    was managed by spring framework.
  public void code() {
    sop(compile);
  }
}
```

Container → alien
obj1 → alien
obj2 → alien

Spring: Class is a Bean (Singleton beans = Object is created only once).

First we have to install dependencies of Spring ApplicationContext in pom.xml.

```
public class App {
  psvm() {
    // BeanFactory factory = new XmlBeanFactory(new FileSystemResource("Spring.xml"));
    ApplicationContext factory = new ClassPathXmlApplicationContext("Spring.xml");

    Alien obj = (Alien) factory.getBean("alien");
    obj.code();
  }
}
```

(Deprecated; It is an interface)

<bean id = "alien" class = "SpringDemo.Alien" />

→ <bean id = "alien" class = " " scope = "singleton" > </bean>
↳ by default

→ <bean id = "alien" class = " " scope = "prototype" > </bean>
↳ It will create multiple objects based on no of times we are requesting.

(obj1 → alien, obj2 → alien)

In singleton, if we not ask for object also, it will create one. But in prototype, if we doesn't call object, it doesn't create any object for us.

Laptop (laptop);
object name

Setter Injection:

Initializing age in xml file:

primitive:
I: <bean id="alien" class=" " >

It called setAge() method
(If we change method name then it will give error)

<property name="age" value="10" > </property>

<property name="laptop" ref="laptop" > </property>

If it is age1 → then method name is setAge1()

</bean>

Reference:
II: <bean id="laptop" class=" " > </bean>

~~<property name~~

Constructor Injection:

→ Initializing age with constructor:

<bean id="alien" class=" " >

<constructor-arg value="12" > </constructor-arg>

Autowire:

autowire = "default" (byType) (byName)

property name is com and bean is com

<bean id="alien" class=" " >

<property name="age" value="10" > </property>

<property name="com" ref="laptop" > </property>

(or) desktop (or) com

<bean id="laptop" (or) "com" class=" "

</bean>

<bean id="desktop" - -

primary="true"

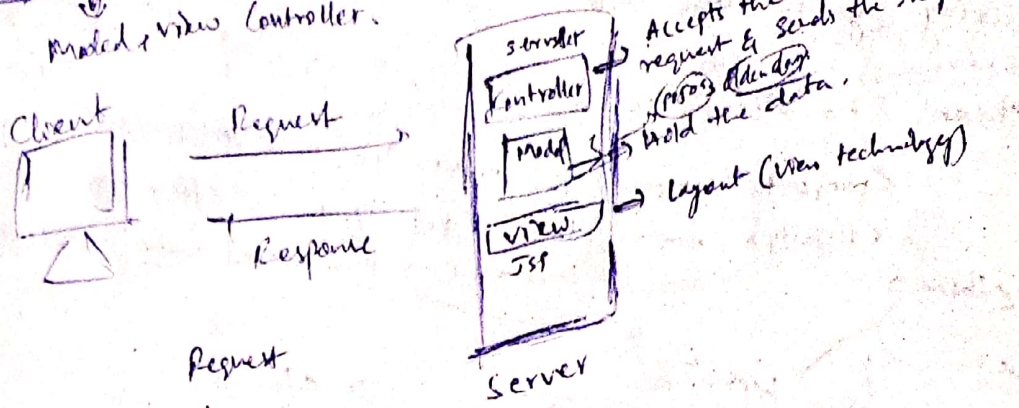
Object of Computer (Inheritance)

We can remove this because name and ref is com by mentioning autowire = byName

② If "byType" is there, and both com & desktop is there then primary="true"

Spring MVC:

Model, View, Controller.



Request

FrontController / Dispatcher Servlet

↓

User Controller

Creating a controller: (@Controller)

```
@Controller
public class HomeController
{ @RequestMapping("/")
  public String home(){
      String jsp(Home page Request) return "index.jsp";
  }
}
```

index.jsp

```
<html>
:
</>
```

Tomcat Jasper: We need for our application to run web applications

search in mvn and download dependencies in pom.xml.

Accepting user Input:

```
@RequestMapping("add")
public String add()
{
  return "result.jsp";
}
```

index.jsp

```
<html>
<body>
<form>
enter two inputs
button
</form>
```

result.jsp

```
@RequestMapping("add")
public String add(HttpServletRequest req)
{  int i = Integer.parseInt(req.getParameter("num1"));
   int j =                                  "num2"));
   int num3 = i+j;
   HttpSession session = req.getSession();
   session.setAttribute("num3", num3);
   return "result.jsp";
}
}
```

modifying code (By removing HttpServletRequest req object

~~public String add(int i, int j, Htt~~

```
public String add(@RequestParam("num1") int i, @RequestParam("num2")
                  int j, HttpSession session) {
   int num3 = i+j;
   session.setAttribute("num3", num3);
   return "result.jsp";
}
}
```

modifying code (By removing HttpSession session)

```
public ModelAndView add(@RequestParam("num1") int i, @RequestParam("num2")
                        int j){
   ModelAndView mv = new ModelAndView();
   mv.setViewName("result.jsp");
   int num3 = i+j;
   mv.addObject("num3", num3);
   return mv;
```

result.jsp

Result : ${num3}

Prefix & Suffix: (By removing '.jsp' extensions, API call doesn't find pages so, to run this) we want to add this.

Go to
Application properties file"

```
spring.mvc.view.prefix = /views/
spring.mvc.view.suffix = .jsp
```

Model and ModelMap: (Alternate of ModelandView) form client action

```
public String add(@Req... , Model m)
    int num3 = i+j;
    m.addAttribute("num", num3);
    return "result";
}
```

and method name but in @RequestMapping("add")

ModelMap. only data

ModelAttribute:

```
public String addAlien(@ModelAttribute("a1") Alien a, Model m)
{
    m.addAttribute("alien", a1);
    return "result";
}
```

ModelAttribute at MethodLevel:

```
@ModelAttribute
public void modelData(Model m)
{
    m.addAttribute("name", "Aliens");
}
```

Spring → lot of configurations

Spring Boot
↓
No need to do
lot of configurations

Spring MVC project: (we have to configure Front

1. Web.xml:

```
<web-app>
  <servlet>
    <servlet-name>telusko</servlet-name>
    <servlet-class>DispatcherServlet</servlet-class>   must be same
      org.springframework.web.servlet.DispatcherServlet
  <servlet-mapping>
    <servlet-name>telusko</servlet-name>
    <url-pattern>/</url-pattern>
```

telusko-servlet.xml:

```
<beans>-/views>
<beans>-jsp>-
```

Spring Boot:

Fetch data from server = GET
Post data to server = POST <form action='addAlien' method='POST'>
sending

@RequestMapping(value="addAlien", method=RequestMethod.POST)
@GetMapping((or))
@PostMapping((or))

Getmapping:

@GetMapping("getAliens") model m.
public String getAliens() {
  List<Aliens> aliens = Arrays.asList(new Alien(101,"Naveen"), new Alien(102,"Kiran"));
  return "showAliens";
m.addAttribute("result", aliens); }

Spring ORM

// Object Relational Mapping

JPA (Java Persistence API)

→ When we connect Java application with database, we use JDBC ... Hibernate.

for bigger application

achieve ORM.

→ add 4-5 dependencies in pom.xml.

1. Hibernate ORM Hibernate Core
2. Spring ORM // connect spring & hibernate.
3. Spring Transaction // handling Transaction
4. mysql Connector // to connect with db
5. c3p0 (...) // pooling

Configure Hibernate:

telusko-servlet.xml:

Table Database

Aliens Telusko.

Every table → have Dao layer // Data access object.

```
public class AlienDao {
    @Autowired
    private SessionFactory sessionFactory; // To fetch data first we use session. To use sessions, we use sessionFactory.

    @Transactional
    public List<Alien> getAliens()
    {
        Session session = sessionFactory.getCurrentSession();
        List<Alien> aliens = session.createQuery("from Alien", Alien.class).list();
        return aliens;
    }
}
```

```
@Entity  → declare class as Entity.
public class Alien {
    @Id  → primary key as 21
    private int aId;
    private String aname;
```

Add & Fetch:

Spring Data JPA Configuration: (CRUD operations)

① Mysql connector } dependencies.
② Spring Data JPA

→ Compulsory to provide configuration

application.properties:

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

url, uname & password

spring.datasource.url = jdbc:mysql://localhost:3306/telusko

spring.datasource.username = root

spring.datasource.password = kundan@04

JPA Get & Fetch:

Spring REST: (only data) = GET, POST, PUT, DELETE.

Representation State

→ We are not working actions, we are working with nouns.
→ We can't maintain session on this.

Postman setup:

```
@Controller
public class AlienController
{
    @GetMapping("aliens")
    public List<Alien> getAliens(){
        List<Alien> aliens = repo.findAll();
        return aliens;
    }
}
```

@Autowired
Alien Repo repo;

@ResponseBody // it use when we specify actual data not view

// fetch from db.

This is actual data not jsp file

localhost:8080/alien/104.
To fetch/get one alien
↓ It will not work

Jackson: (Java data to json)

PathVariable

→ By not mentioning @ResponseBody to all methods at method level, instead of that we can mention @Controller as @RestController at class level.

```
@GetMapping("alien/{aid}")
@ResponseBody
public Alien getAlien(int aid)
{
    Alien alien = repo.getOne(aid);
    return alien;
}
```

For one particular Alien:

```
@GetMapping("alien/{aid}")
@ResponseBody
public Alien getAlien
    (@PathVariable("aid") int aid)
{
    Alien alien = repo.findById(aid).
        orElse(new Alien(0,"0"));
    return alien;
}
```

Send data:
through postman.

→ Body (in this key and value)

```
@PostMapping("alien")
public Alien addAlien(Alien alien)
{
    repo.save(alien);
    return alien;
}
```

→ To get data in form of xml, we have to add one more jar file jackson-xml. and add in pom.xml file

Produces attributes:

By default when client req data, it will return in json format from server. If client mention xml format, then it will return in xml format.

→ If client doesn't mention in xml format, but client wants in xml format. Then we should mention that in Server (Controller) side.

```
@GetMapping(path="aliens", produces={"application/xml"})
public List<Alien> getAliens(){
    List<Alien> aliens = repo.findAll();
    return aliens;
}
```

// If we mentioned this, it will only give xml, not json

Accept data from client:

Give data in Json format (not in form data)

```
@PostMapping("alien") → @PostMapping(path="alien", consumes={"application/json"})
public Alien addAlien(@RequestBody Alien alien)
{ repo.save(alien);
  return alien;
}
```

// Json to object. (To accept Json format data)

Spring AOP: (To maintain log files and to maintain business logic code clean by not creating/specifying print() statements at every time. It comes into play)

Aspect = Aspect oriented programming

Aspect: One class can contain all the log methods.

Joint point: We have business logic methods, out of which we have to maintain logs for 2-3 methods. So, we have to connect them. Suppose, I want to maintain log for getAliens(). So, here getAliens() becomes my joint point.

This log becomes advice.

Pointcut: When we have getAliens(), we have to keep log what it is called pointcut. (Its a Expression & log - log() { Body & log })

Weaving: We have to connect actual method with the advice. This connection/weaving done at runtime.

{ Before Advice → Before Execution of method
After advice → After
During advice → During.

Steps:

@Aspect → To make aspect
@Component

public class LoggingAspect
{
@Before("Execution(public ~~List<Alien>~~ .getAliens())")
public void log() {
sop("getAliens method");
}
}

Joint point
↕
com.telusko.springmvcdemo.AlienController.getAliens()
↓
pointcut

Any return type

By default After (finally) advice

@After(,) → @AfterThrowing
↳ exception

Spring Security:

Authentication — Authorization (Access control like Admin, users, Content Creator all have their roles like admin can add, delete blogs. Content creator can edit their blogs, not others. User have only access to read.)

When we import spring security boot jar file pom.xml.

It will ask login details to access particular page.

'user' ; user → By default. } By spring security jar file.
password is generated.

To create user and password manually:

```
@EnableWebSecurity
@Configuration
public class AppSecurityConfig extends WebSecurityConfigurer
                                       Adapter {

    @Bean → object creation
    @Override
    protected UserDetailsService userDetailsService() {

        List<UserDetails> users = new ArrayList<>();
        users.add(User.withDefaultPasswordEncoder().username("navin")
                .password("1234").roles("USER").build())
                                          ↓
                                        role of

        return new InMemoryUserDetailsManager(users);navin

    }
```

Adding through java file manually

UserName & password will be verified from DB? [Service layer] ↑ [Controller]

@Bean
public AuthenticationProvider authProvider() {

DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
provider.setUserDetailsService(userDetailsService);
return provider; → provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance())

}

→ In order to setup the DB, we have to do some configuration on application.properties.

① spring.datasource.url =
spring.datasource.username = root // DB username
spring.datasource.password = Kundan@0411 // DB password
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver

Spring Security BCrypt Password Encoder:

Spring Security login form:

→ main
  ↳ webapp
    ↳ login.jsp
    ↳ logout.jsp

goto appSecurity and override

@Override
protected void configure( ) {
{ http
.csrf().disable()
.formLogin()
.loginPage("/login").permitAll()
.and()
.logout().invalidateHttpSession(true)
.clearAuthentication(true)
.logoutRequestMatcher(new

Spring Boot Security OAuth2

→ Add dependency in pom.xml for OAuth2.

→ @EnableWebSecurity
→ @EnableOAuth2Sso

Application.properties

url = /google
tokenName = oauth_token
scope = profile email
clientId =
clientSecret =

Dockers:

1) Virtualization:

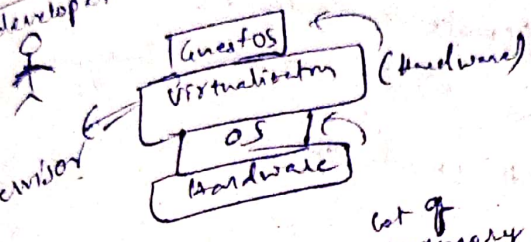
→ When the proj works on our machine and not on other machines due to configurations / versions.

Dockers & Virtualizations

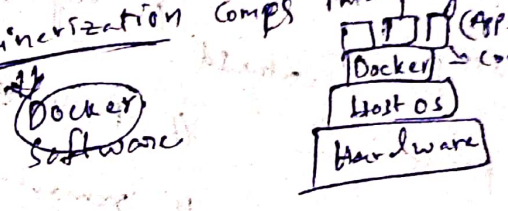
Comes to role.

Virtualization: faking hardware & OS (this layer is called Virtualization) It is a software.

developer



→ Virtualization occupies lot of memory and process will slows. To overcome this containerization comes into play.



Adv:

→ Isolation
→ Security
→ Portability from one machine to other machine easily
→ Consistency (works same in all machines)

Docker: (to achieve containerization)

→ First containers should be deleted then only image can be deleted

Command prompt:

> docker --version // verifying docker is present in system
> docker run hello-world // get the image. (we can push our own images to Docker hub)
image name.

If we don't have image, pull the image from Docker hub.

> docker ps // which containers are running
> docker ps -a // show me all, which executed already
> docker images. // images we have.
> docker rm <id> // remove containers
mentioned
> docker rmi <id> // remove images.
mentioned

> docker start ~ container Id

> docker create hello-world // create Image.

Step by step:

> docker search hello-world // searching the Image.

> docker pull hello-world // pull the Image means download the Image.

> docker create hello-world // create a container
Image name. => It will return Container Id.

> docker start ~ container Id // start the container

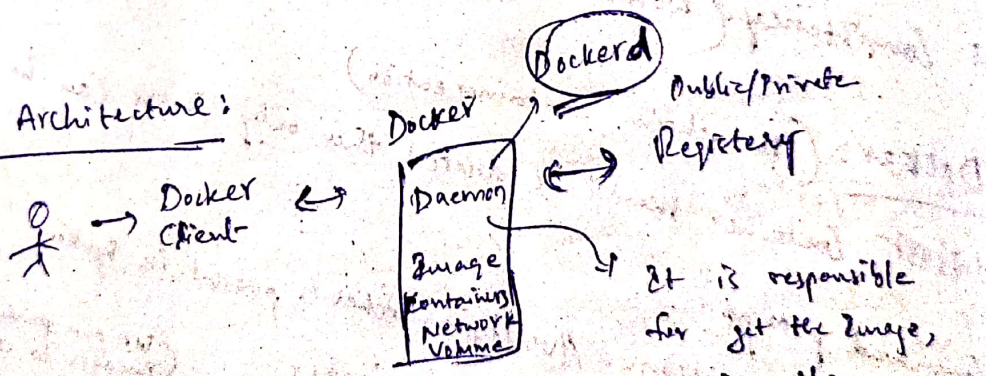
> docker stop ~ container Id // stop the container

> docker pause ~ // pause the container.

> docker rm ~ // remove the container

> docker ps -a // verify that removal of Containers. (It gives list of Containers)

Docker Architecture:



Running JDK docker Container:

By using Ishell > docker pull openjdk:22-jdk

> docker run -it openjdk:22-jdk

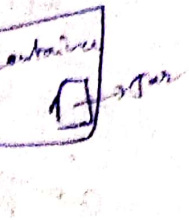
Packing the Spring Boot Web App:

→ Goto start.spring.io.

Application.properties
server.port = 8081

Containers | jar file.

Running spring Boot Web app on Docker:



> docker ps -a

> docker exec -it container id /bin/sh

> docker cp target/rest-demo-0.0.1-SNAPSHOT.jar gitted_ethel:/tmp → jar file

> docker commit --change 'ENTRYPOINT ["java", "-jar", "/tmp/rest-demo.jar"]' container name image name // Create Image of Container

→ machine & Docker has different network (means diff - port numbers)

> docker run -p 8081:8081 telusko/rest-demo

Docker file for Docker Images:

1) Doing manually (Every Time we have to create Image we will run docker file)
Achieve this docker file

DockerFile:

```
From openjdk:22-jdk
ADD target/rest-demo.jar rest-demo.jar
ENTRYPOINT ["java", "jar", "/rest-demo.jar"]
```

> docker build -t telusko/rest-demo . → same directory

8081:8081.

Web App with Postgres:

Docker Compose : → want to run on DB on seperate Container

→ mvn clean package.

Docker

It is a tool - use to multiple containers.

Application → Container 1

Database → Container 2

→ docker compose up. - build

docker-compose.yml

version : "3.7"

services :

app :

build :

ports :

"8080 : 8080"

↓ Host port No ↓ Container port no

networks :

- spring

postgres :

Both are working, but in diffrent network. We want to join in one network in config.

networks :

- spring

when restart containers it will lose data → to not lose data

To achieve that we use docker volumes

networks :

- spring

volume :

postgres-data

⇒ Accepting parameters in methods

use ⇒ @RequestParam

⇒ If Jpa doesn't give required data from db, we have to write query by using → JPQL (sql)

@Query (" ", nativeQuery = true)

↓ mention query

value = " "

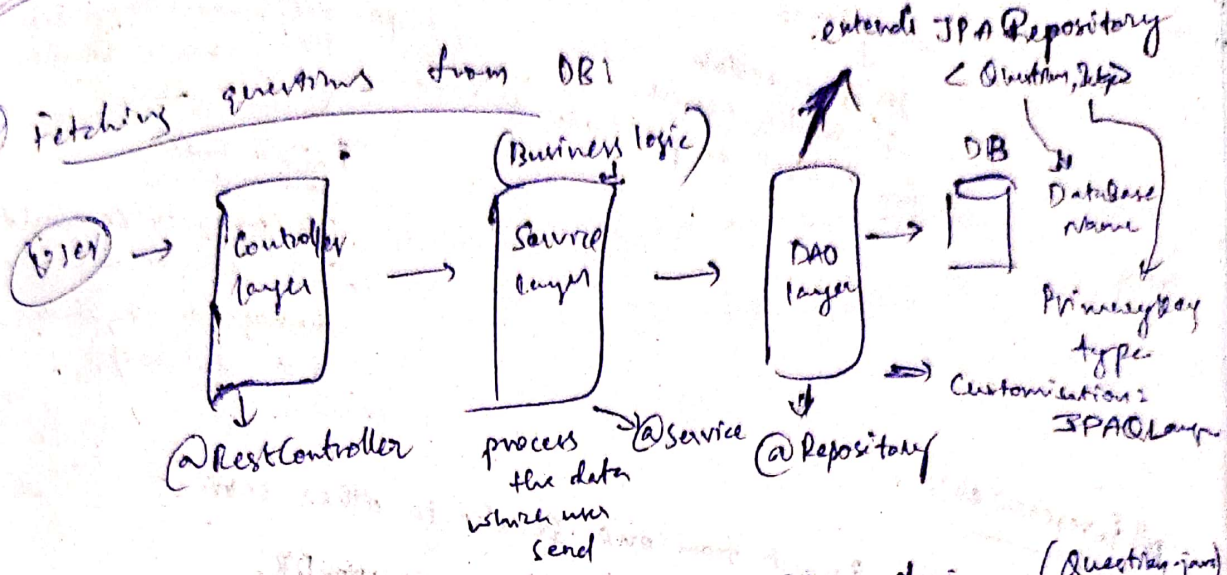
query

Micro Services:

→ localhost:8080/question/allQuestions → All questions

Application properties: setup work for postgres & configuration of postgres

Step 1: Fetching questions from DB:



Databases:

@Entity → To represent each table

@Id → Primary key.

@GeneratedValue (strategy: GenerationType.Sequence)
↳ Primary key auto generated.

Class design: (Question.java)

Class Name → Table Name

Class Fields → Table Columns

Each object → Each row (ORM)
↳ Object Relational mapping

(Variable in class should match the column name in table)

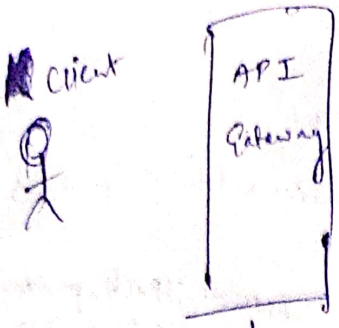
Adding questions ~~Post Map~~

Status codes:

100-199 → Informational response
200-299 → Successful responses
300-399 → Redirectional messages
400-499 → Client error responses
500-599 → Server error responses

If we want to return Questions + statuscode we want to use Response Entity

return new ResponseEntity<>
(, HttpStatus.OK)



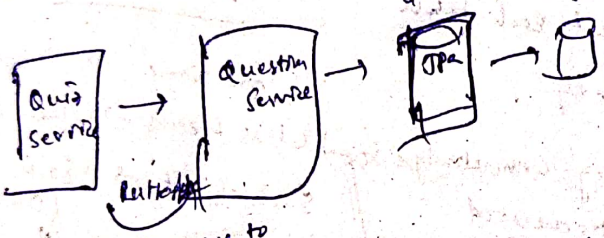
↓
It is responsible for sending request from client to particular micro service.

Service registry: It is responsible for communication between micro services.

Failed fast:
If one service would fail, then it will give message to user fast, not to wait.

Microservices:

How to send request from one service to other service



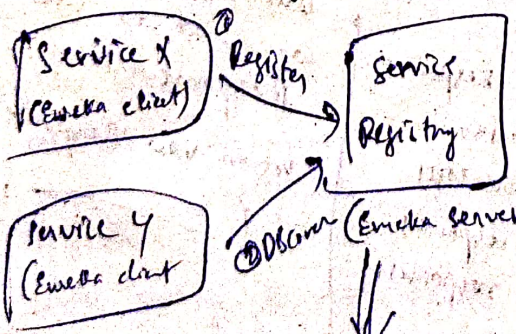
Sending req to one service to other service by using "Rest template" (Without help of Feign)

↓

@FeignClient ("QUESTIONSERVICE")

Feign (declarative way of requesting other service)

Service discovery (Eureka)

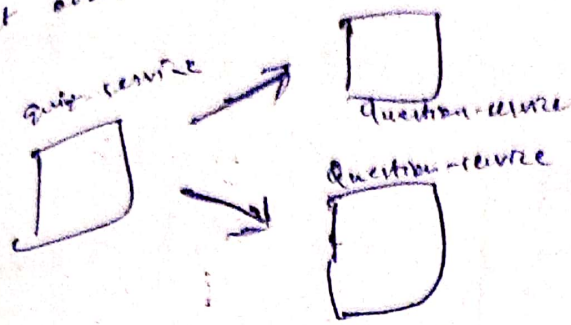


@EnableEurekaServer to enable it

All microservices want to register in Eureka server and search one service to other service naturally, by doing this we don't have to mention IP address and port number of services and with the help of Feign we can actually request directly to the service.

Load Balancer:

If there are multiple instances, and all are busy with the requests except one, then request goes to that particular instance.

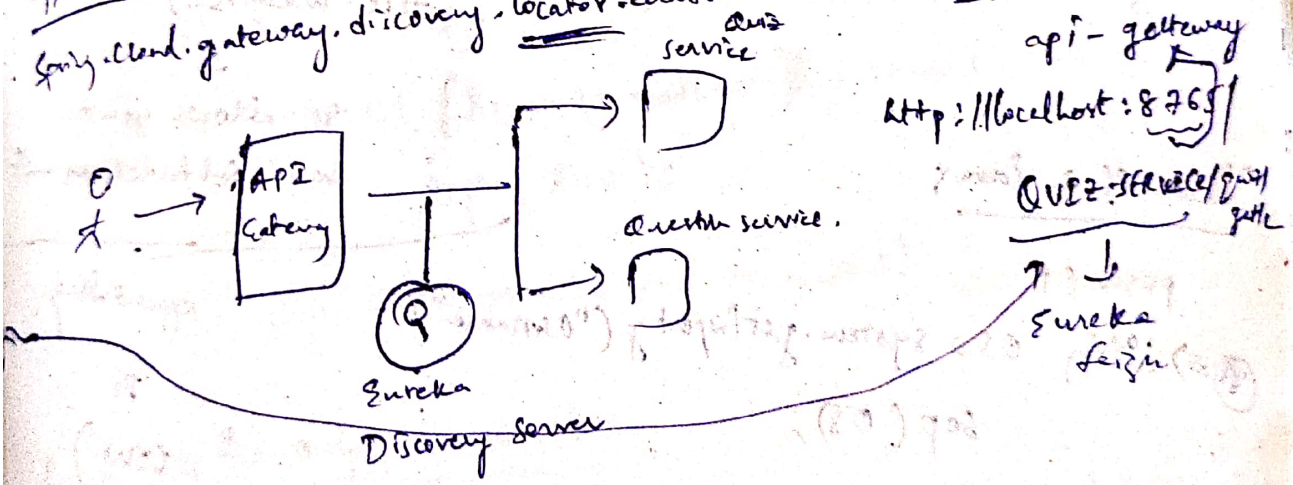


Firstly, Quiz-service sends request to two services, it will go where the service is free. If both services are free, then it will go randomly.

API Gateway:

application.prop:

spring.cloud.gateway.discovery.locator.enabled = true // to allow searching services



Spring docs.

Introduction to Spring: (Navin Reddy - 4 sections)

Spring Boot
Spring framework

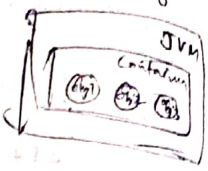
Injecting objects into Applications

IoC & DI: Inversion of control and Dependency injection. → Design pattern

It's our responsibility to create the objs & maintain the objs.

So, here we have give control to someone and focus on business logics.

Here, IoC container comes into play that it has objects, spring will create objects for us in IoC container.



Spring vs SpringBoot: we don't have to do much configurations.

To communicate with IoC Container we use Application Context + creation & container

Dependency Injection:

→ Any objects which is created by spring = Beans.

By default spring doesn't create Beans

By using Component, spring will create object/Bean in IoC Container.

Alien.java:
@Component
code() {
sop("coding");

main():

main() {
ApplicationContext context =
SpringApplication.run( )
inside run it returns ApplicationContext obj.

Alien obj = context.getBean(Alien.class);
obj.code();

@Autowired → will search object in IoC container (It creates objects automatically)
Xml Config:
Spring project: <beans resources/spring.xml : (create xml file)
<bean id="alien" class="com.telusko.Alien">
<bean id="lap" class="Laptop">
</beans>
</beans>

Singleton ← same class package
prototype ←

like @Component

main:

main() {
ApplicationContext context = new ClassPathXmlApplicationContext() // create a container object
Alien obj = context.getBean("alien");

Alien-class:
code() {
}

obj.code() (Alien)

Alien obj:
obj.code()

→ only one object

diff object created at every time

Laptop class:
<bean id="alien" class="Laptop" scope="prototype">

Singleton = Create obj on load
prototype = " " when getBean()

Setter Injection

We define in xml file.

<property name="age" value="21" />

<property name="lap" ref="lap1" />

<bean id="lapp" class="" />

Constructor Injection?

<constructor-arg value="21" />

<constructor-arg ref="lap1" />

<constructor-arg index="1" value="" />

<constructor-arg index="0" ref="" />

autowire="byName"

autowire="byType"

primary="true".

Lazy Initialized of Bean:

→ By default all the objects created when we initialize/declare as beans in xml and all ApplicationContext, even we those objects are not,

→ So, when we don't want to create object which we are not using those objects, lazy bean comes into play.

<bean id="comp" class="Desktop" lazy-init="true" />

→ we can initialize object as lazy then application process fast.

The desktop object is not created by default. Only when we use obj and called, then it will be created and present in container (singleton)

Id name in xml file

getType:

Alien obj1 = context.getBean("alien1", Alien.class); // to avoid typecasting

Desktop obj = context.getBean(Desktop.class);

Inner Bean:

<bean id="alien" class="Alien" autowire="byType">

<property name="com">

<bean id="com1" class="Laptop" />

</bean>

</property>

→ This laptop is only used or restricted to alien class by using Inner Bean

Java Based Config :

main :

```
main() {
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
Desktop dt = context.getBean("desktop", Desktop.class);
dt.compile();
}
```

```
@Configuration
public class AppConfig {
@Bean(name = {"com1", "Beast"})
public Desktop desktop() {
return new Desktop();
}
}
```

default name is method name

Scope Annotation : (By default all Beans are Singleton) → It load only once, object created once

```
@Scope("prototype")
```

Autowire :

```
Alien class implements desktop, Laptop
@Autowired → Not mandatory
@Bean
public Alien alien(Computer com){
@Qualifier("desktop")
}
```

Name of Bean

```
@Bean
desktop() {
}
```

```
@Bean @Primary
Laptop() {
}
```

@Component : Objects are created by Spring in IoC.

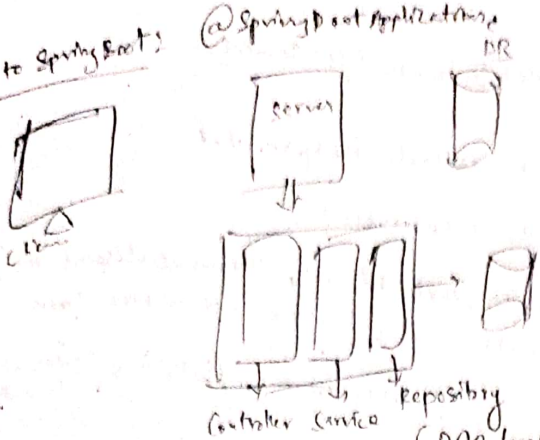
Bean name is class name (first letter small)

@Qualifier get preference than @Primary.

@Value("21") → Injecting a value.

```
private int age;
```

Spring to SpringBoot:



model → Entities package

We can use @Component, but @Service makes more sense.
" " " @Repository, but @Repository " " "

Spring JDBC:

→ When we connect first time it create connection, when we connect 2nd time again it creates connection. So, to avoid this we use data source

application.properties
- schema.sql
- Data.sql

spring.datasource.url
spring.datasource.username

JDBC template:

```
@Repository
StudentRepo {
private JDBCTemplate jdbc;
public void save(Student s) {
String sql = "insert into student(rollno, name, marks) values (?,??)";
int rows = jdbc.update(sql, s.getRollNo(), s.getName(), s.getMarks());
Sop(rows) (((← (inserting only 1 row)
public List<Student> findAll() {
String sql = "select * from students";
return jdbc.query(sql, mapper);
```

→ RowMapper<Student> mapper = new RowMapper<Student>(){ → it give one row at a time.

```
public Student mapRow(ResultSet rs, int rowNum)
{
Student s = new Student();
s.setRollNo(rs.getInt("rollno"));
s.setName(rs.getString("name"));
s.setMarks(rs.getInt("marks"));
return s;
}
```

Web App Intro:

Servlet → Servlet Container / Web Container (Ex: Tomcat)

Enabling/giving Servlet features is to extends HttpServlet

public class HelloServlet extends HttpServlet {
    public void service(HttpServletRequest req, HttpServletResponse res) {
        res.getWriter().println("Hello"); → // this will print on screen
    }
}

@WebServlet("/hello") → (external Tomcat)

res.setContentType("text/html");
pwt("<h2>Hello World</h2>");

main:

main() {
    Tomcat tomcat = new Tomcat();
    tomcat.start();
    tomcat.getServer().await();
}

8081
tomcat.setPort(8080);

Context context = tomcat.addContext("", null);

Tomcat.addServlet(context, "HelloServlet", new HelloServlet());
                              ↓
                           name of servlet

Context.addServletMappingDecoded("/hello", "HelloServlet");

Introduction to MVC: (Servlet & JSP)

Create a JSP page:

webapp/index.jsp: Hello world.

Create a Controller:

@Controller
public class HomeController {
    @RequestMapping("/")
    public String home() {
        return "index.jsp";
    }
                ↓
            page name

    @RequestMapping("add")
    public String add(HttpServletRequest req) {
        int num1 = Integer.parseInt(req.getParameter("num1"));
        num2 = " " " = (req.getParameter("num2"));
        int result = num1 + num2
        return "result.jsp";
    }

HttpSession session

session.setAttribute("res", result)

result.jsp:

Result is <%=session.getAttribute("res")%>
${res}.

To remove HttpServletRequest.

@RequestParam: (Simplify the previous code)

public String add (@RequestParam("num1") int num1, @RequestParam("num2") int num2, HttpSession session) {

int result = num1 + num2;

(num1 & num2 is the names in the URL.)

If these names are num1, num2 then no need to mention @RequestParam()

To remove session:

Model Object:

If you pass data in between pages, then we use model objects

public String add (int num1, int num2, Model model)

int res = num1 + num2;

model.addAttribute("result", res);

return "return-jsp";

ViewResolver:

return index; → Didn't mention extension.

return result;

prefix → views

suffix → extension

ModelAndView:

public ModelAndView add (int num1, int num2, ModelAndView mv) {

int result = num1 + num2;

mv.addObject ("result", result);

mv.setViewName ("result");

return mv;

ModelAttribute: Instead of using @RequestParam several times in parameters.

Input:

public String addAlien (@ModelAttribute Alien alien) {

return "result";

(or)

@ModelAttribute ("alien") Alien alien

optional.

result.jsp

${alien}

}

Method:

@ModelAttribute ("course")

public String CourseName () {

return "Java";

}

result.jsp

${course}

⇒ Save

Create a Spring MVC Project (We have to do all configurations)

web-xml

→ By default all requests are 'get requests'.

Lombok
@Data → (Don't have to create getters, setters etc)
@NoArgsConstructor
@

Rest API using SpringBoot

→ State = Data. JSON format
(JavaScript Object Notation.

Http methods → Get (view data/Read Data)
→ Post (Create data)
→ put (update)
delete

→ By default when we use @Controller, for methods will return view name. If we don't want view, then add @ResponseBody (Eg: index.jsp, home.jsp etc). If all methods return data, then we use @RestController on class level.

@CrossOrigin(origin:"link") → pls allow the link to get access

http://localhost:8080/jobposts/id.

@GetMapping("jobPost/{postId}") @PathVariable("postId") int postId
public JobPost getJob(int postId)
{ return service.getJob(postId);

@PostMapping("jobPost") → we annotate sending json format to server
public void addJob(@RequestBody JobPost jobPost){
service.addJob(jobPost);
}

@PutMapping("jobPost")
public JobPost updateJob(@RequestBody JobPost jobPost) {
service.updateJob(jobPost);
return service.getJob(

@DeleteMapping("jobPost/{postId}")
public String deleteJob(@PathVariable int postId){
service.deleteJob(postId);
return "deleted";

Connection to postgres

JPA?

spring.jpa.hibernate.ddl-auto=update //create table (Actually update, because if we assign create, then it will create table everytime we runs. So, update is correct here because, first time it creates, from next it won't create table).

spring.jpa.show-sql=true.

findAll(); ——→ (By using Domain Specific Language, JPA create methods)

findById()

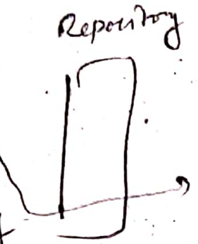
@Query("select s from Students where s.name = ?1")

List<Student> findByName(String name);

findBy + 'col_name'
'variableName'

repo.save(s1);

repo.delete(s3);



Spring Data Rest ?

→ Add Rest Repositories dependency

→ Spring data JPA

→ PostgresSQL, LomBok

→ No, Controller & Service layer in Data Rest.

Only model, Repository

Spring AOP ? (Seperate class for logging, security, validation, exceptions)

(Aspect Oriented Programming)