Master Hibernate + JPA:

JPA persistence API

JPQL
↳ Criteria queries
↳ Native queries

Query methodologies

// Create criteria query
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Courses> cq = cb.from(CourseClass)

TypedQuery<Courses> query = em.createQuery(cq);

@Component → Class managed by Spring Framework.
@Primary → When we have multiple components, we will use @primary

Maven: Manages jars needed by apps (application dependencies)
↳ enables consistent usage across all projects.

Dependency injection types: → Constructor based (A)
→ Setter based
→ Field

List<Courses> result = query.getResultList();

logging.level.org.springframework=debug

→ Spring Framework is divided into modules:
① Core: IOC container
② Testing: Mock objects, Spring MVC Test etc
③ Data access: Transactions, JDBC, JPA etc
④ Web servlet: Spring MVC
⑤ Web Reactive: Spring WebFlux
⑥ Integration: JMS etc

JPQL query:
→ Select c,s from Course c JOIN c.students s
→ select c,s from Course c Left join c.students s

② Repository

public class JdbcDao{
@Autowired
JdbcTemplate jdbc;

@Setting class

public List<person> findAll(){
return jdbc.query("select * from person", new BeanPropertyRowMapper(Person.class));
}

jdbc.queryForObject("select * from person where id=?", new Object[]{id})

Execute the findAll method by implements CommandLineRunner:

↳ run method.

Insert & update methods:

public int insert(Person person){
return jdbc.update("insert into person (id,name,location,birthdate) values (?,?,?,?)",
new Object[]{person.getId(), person.getName(), person.getLocation(), new Timestamp(person.getBirthDate().getTime())});

→ JPA is Interface, Hibernate is class. Hibernate implements JPA.

ORM → Object Relation Mapping.

@Column(name = "col_name") @Table(name = "table_name")

Junit: assertEquals(,)

assertTrue(), assertFalse() → for Boolean.

assertArrays(new int[]{1,2}, new int[]{1,2}); // true

@Test → Test class Annotation

@BeforeEach → Run before each test

@AfterEach → Run after each test.

@BeforeAll → It run before All tests. → class level method (static method)

@AfterAll → " after " tests. → " " "

@DirtiesContext → It reset the data after execution.

Create findById using JPA EntityManager:

import org.springframework. → @Repository

public class Repository{ Implements CourseRepository

@Autowired

EntityManager em)

public Course findById(long id){

return em.find(Course.class, id);

}

public void deleteById(Long id){

Course course = findById(id);

em.remove(course);

public Course save(Course course){

if(course.getId() == null){

em.persist(course);

}else{ em.merge(course);

}

return course;

}

→ em.persist(course) → Gives all the changes

→ em.flush() → The changes upto that point are send to DB.

→ em.refresh(course) → override and save the changes

LocalDateTime → @UpdateTimeStamp → Hibernate Annotations
private LocalDateTime lastUpdatedDate; not JPA Annotations
→ @CreationTimeStamp.
private LocalDateTime CreatedDate;

@NamedQuery (name = "query-get_all_courses", query = "Select c From Course c")
This is not repeatable Annotation, Instead define @NamedQueries

@Entity
public class Course {

@NamedQueries(value = {@NamedQuery ( ),
@NamedQuery ( ) })

Native queries: (to avoid select c from course c
to bring select * from courses).

→ Tuning parameters.
→ mass update in tables.

@Test
public void native_queries_to_update() {
Query query = em.createNativeQuery("Update course set
last_updated_date = ")
int noOfRowsUpdated = query.executeUpdate();

Course, Student, Passport, Reviews.
one to one.

OneToOne:

Student {
@OneToOne (fetch = FetchType.LAZY)
private Passport passport;
}

passport {
}

StudentRepository {
public void saveStudentWithPassport() {
Passport passport = new Passport("z123");
em.persist(passport); //
Student student = new Student("Mike");
student.setPassport(passport);
em.persist(student);

@Transactional

To avoid Eager fetching we use Lazy functions.

Eager Fetching: Even fetching the student details, the passport details are also executing.

One to One (Bidirectional);

Passport {

private Student student;

Student {

@OneToOne (fetch = FetchType.LAZY, mappedBy = "passport")

Opposite side

Student making mapped as owning side of relationship, passport is inverse

→ Because of duplication of data, either Student could have passport_id (or) vice versa (Owning side of relationship)

Many to One: One course has many reviews (Course's side of relationship, so add mapped by course.)

Course {

Review {

@OneToMany (mappedBy = "course")
private List<Review> reviews
= new ArrayList<>();

Many to One
private Course course;

By default it is lazy fetching

(Or OneToMany

(On ManyToOne, fetching is Eager fetch..

Many to Many: Join tables: Course-student

| Course_id | Student_id |

Many to many
By default
lazy fetch it only fetch student details.

Owning relationship because mappedBy is on Course side

Student {

Course {

@ManyToMany (mappedBy = "courses")
private List<Student> students
= new ArrayList<>();
(getters & setters & addCourse methods)

@ManyToMany (
private List<Course> courses
= new ArrayList<>()
addStudent)

→ make one of owning side of relationship, because @ManyToMany create two join tables makes duplicate data (It doesn't really matter which is owning side in ManyToMany)

@JoinTable (name = "STUDENT_COURSE",
joinColumns = @JoinColumn (name = "STUDENT_ID")
inverseJoinColumns = @JoinColumn (name = "COURSE_ID"))

insertStudentAndCourse()

{ Student student = new Student("Jack");
Course course = new Course("Microservices in 100 steps");
em.persist(student); // gets an id, when persist calls
em.persist(course);
student.addCourse(course);
course.addStudent(student);
em.persist(student); // owning side persist.

Inheritance Relationships → SingleTable
→ Table per class
→ Joined
→ MappedSuperClass.

@Inheritance (strategy = InheritanceType.SINGLE_TABLE)
public Employee extends Employee{
Employee {

@DiscriminatorColumn(name = "EmployeeType")
(giving new column name)

- TABLE_PE_CLASS.
- JOINED.

@MappedSuperclass (Remove Entity)

In repository define
partTimeEmployee & fullTimeEmployees

Repository:

EmployeeRepository{

public void insert(Employee employee)
em.persist( );

}
public List<Employee> retrieveAll(){
return em.createQuery("select e from
Employee e", Employee.class)
.getResultList();

Default strategy.
SingleTable: (Both ptEmployee & ftEmployee stored in single table) → (we will have lot of nullable colu

TablePerClass: (Individual tables were created) → (Common columns are repeated in both tables)

Joined: (Joined performed, and fetch the details) → (Really good, however it doesn't perform well because if going so many tables to get data

MappedSuperclass: (Completely eliminates Inheritance)

When to use and what to choose??
→ Data Integrity → Join column.
→ Permonance: SingleTable.

Transaction Management:

Permits Rollback

ACID → Atomicity, Consistency, Isolation, Durability.

Dirty Read → Reading the modified value.

Non repeatable Read → when I'm reading same transaction, I'm getting 2 diff values because of updations.

Phantom Read: Diff no. of rows.

Isolation levels: Read Uncommitted, Read Committed, Repeatable Read, Serializable

Caching with JPA & Hibernate:

Service → Data → DataBase

→ If user come first, then we will fetch the details from DB; from next time onwards we will fetch the details from cache in Data layer to avoid roundtrip from Data to DB everytime.

PersistenceContext → FirstLevelCache → SecondLevelCache → DB

FirstLevelCache: (Specific to single Transaction, @Transactional)

SecondLevelCache: (Stored Common Info of all users, Eg: Country, state, course details, @Transactional)

@Cacheable → SecondLevel cache
Course {

SecondLevel Cache:

→ Add dependency in pom.xml (eh cache)

application.properties:

→ Enable second level cache

→ specify the caching framework - Ehcache.

→ Only cache what I tell to cache.

→ What data to cache?

→ spring.jpa.properties.hibernate.cache.use_second_level_cache = true

→ " " cache.region.factory_class = org.hibernate.cache.ehcache.

→ " javax.persistence.sharedCache.mode = ENABLE_SELECTIVE

→ logging.level.net.sf.ehcache = debug.

Tips!

→ Soft delete: log of deleted cols.

If Course is deleted, then assign true.

@SQLDelete(Sql = "update Course set is_deleted=true where id=?")

Course {

private boolean isDelete;

@Where(clause = "is_deleted=false")

Performance Tuning:

→ Measure and Tune

→ Indexes (Add the right indexes on the DB)

→ Distributed Cache (Distributed load along multiple instances)

→ Enable SecondLevel Caching & Be careful about size of FirstLevel Caching

→ Eager vs Lazy Fetch: → Use lazy fetching mostly

→ Remember that all Mapping ToOne (@ManyToOne and @OneToOne) are EagerFetch By default.

N+1 problem:

To avoid N+1 problem, try to make it Eager.

@NamedEntityGraph() { setHint("javax.persistence.loadgraph", graph)

Spring Boot vs Spring MVC vs Spring

Spring Boot

(@Component, @Autowired, @ComponentScan etc)

Spring MVC

(@Controller, @RestController, @RequestMapping etc)

Spring Building web apps & Rest Api

When do we use JPA?

→ SQL DB

→ Static Domain model

→ Mostly CRUD

→ Mostly simple queries/mappings.

Embedded Services!

→ Install Java

→ Run jar file.

Actuator: (Monitor and manage apps in production)

→ Provides endpoints (beans, health, metrics, mappings).

{trace, debug, info, warning, error, off}

→ Logging.level.org.springframework=info

@ConfigurationProperties(prefix="currency-service")

@Component

class CurrencyServiceConfig {

url;

username;

getters & setters

Application.properties

- url=
- username=
- password=

Spring docs.

Build tool = Maven.

(Navin Reddy - Architect)

Introduction to Spring:

Spring Boot
Spring framework

Principle

Design pattern

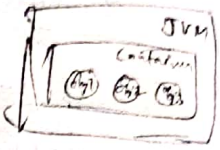
IOC & DI: Inversion of control and Dependency injection.

Injecting Objects into Applications.

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

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( )
```

(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: Singleton → same class prototype

resources/spring.xml : (Create xml file)

```
<beans>
  <bean id="alien" class="com.telusko.Alien">
  <bean id="lap" class="Laptop">
</beans>
```

@Component

main:

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

(Alien)

Alien obj:

```
obj.code()
```

-> only one object

Alien.class:

```
code() {
}
```

Laptop.class:

Singleton = Create obj onload

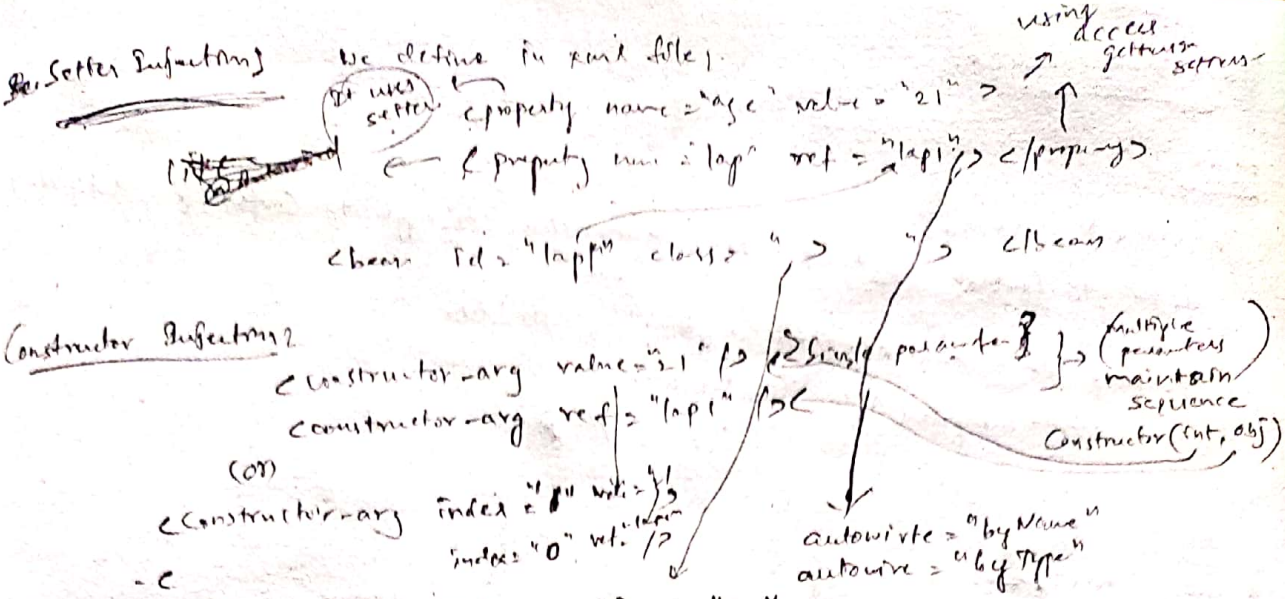
prototype = " when getBean()

diff object created: at every time

<bean id="alien1" class="..."

Scope=prototype

Setter Injection: We define in xml file.



Constructor Injection:

Lazy Initialized 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 these objects, lazy bean comes into play.

<bean id="com1" 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)

getType:

Id name in xml file

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

Desktop obj = context.getBean / desktop.("com");

Inner Bean:

<bean id=alien class="Alien" autowire="byType">
<property name="com">
<bean id="com1" class="laptop" /> -> This laptop is only used or restricted to alien class by using Inner Bean
</bean>
</property>

Java Based Config:

main:

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

"com" / "Beast"

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

default name is method name.
name = {"com", "Beast"}.

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

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

Autowire:

Alien class implements desktop laptop

@Autowired → Not mandatory.

```
@Bean
public Alien alien(Computer com){
    @Qualifier("desktop")
```

Name of Bean

(or)

@Primary.

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

@Components: Objects are created by spring in IoC.

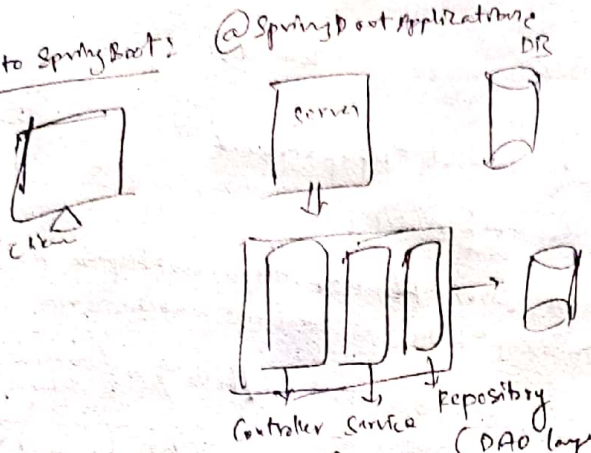
Bean name is class name (first letter small)

@Qualifier get preference than @primary.

@Value("21") → Injecting a value.

```
private int age;
```

Spring to SpringBoot: @SpringBootApplication



model → Entities.
package

(@Service)

We can use @Component, but @service makes much more sense.
" " " @Repository; but @Reposity " " "

Spring JDBC:

→ When we connect first time it create connection, so 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) (1 (querying 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"));   (Col_name)
    s.setName(rs.getString("name"));
    s.setMarks(rs.getInt("marks"));
    return s;
}
```

Web App Intro:

Servlet → Servlet Container / Web Container (Eg: 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");
print("<h2><b>hello world</b></h2>")
```

main:

```
main() {
    Tomcat tomcat = new Tomcat();         8081
    tomcat.start();                       tomcat.setPort(8080);
    { tomcat.getServer().await();
    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")                          HttpSession session
    public String add(HttpServletRequest req) {
        int num1 = Integer.parseInt(req.getParameter("num1"));
            num2 =   2        "        "          ("num2");
        int result = num1 + num2 ——→ session.setAttribute("res", result)
        return "result.jsp";
```

result.jsp:

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

To remove Annotations:

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 sessions:

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";
```

View Resolvers:

return model; return result; → Didn't mention extensions.

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.

argument:

```
public String addAlien(@ModelAttribute Alien alien) {
    return "result";
}
```

(or) @ModelAttribute("alien1") Alien alien

optional.

→ result.jsp ${alien}

Top of method:

```
@ModelAttribute("course")
public String courseName() {
    return "Java";
}
```

→ result.jsp ${course}

⇒ Same

Create a Spring MVC Project (We have to Avoid boilerplate 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

→ Stone = 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/1 id

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

@PostMapping("jobPost") use annotation sending json format in 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";

JPA? Connection to postgres

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 will doesn'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=?

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

Repository → No, Controllers & Service layer in Data Rest. Only model, Repository.

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

(Aspect Oriented Programming)