S.D.S
Page:
Date: / /

* Spring Boot.

- Spring boot is a framework for Rapid Application development build using spring framework with extra support of auto-configuration and embedded application server like tomcat, jetty.
- It helps us in creating efficient fast stand-alone applications which we can run. It basically removes a lot of configurations & dependencies.

* RAD (Rapid Application Development)

→ - RAD is modified waterfall model which focuses on developing software in a short span of time.
- Phases of RAD are as follows:
① Business modeling.
② Data modeling.
③ Process modeling.
④ Application Generation.
⑤ Testing and Turnover.

* Is this possible to change the port of Embedded Tomcat server in spring boot?

→ Yes, put server.properties in application.properties.

* Can we override or replace the Embedded Tomcat server in spring boot?

→ Yes, we can replace Embedded Tomcat with any other servers by using starter dependencies.
- tomcat server is embedded because of starter-web dependency. i.e. spring-boot-starter-web.
- This can be achieve by excluding spring-boot-starter-~~web~~ tomcat from starter-web dependency. and adding another server dependency. e.g. starter-jetty.

* Can we disable the default web server in the spring Boot Application.

→ Yes, we can use the application.properties to configure the web application type i-e. spring.main.web-application-type = none.

* How to disable a specific auto-configuration class?

→ ~~@EnableAutoConfiguration(DataSo~~
~~@Ea~~ @EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})

* @SpringBootApplication:

→ It is equivalent to @Configuration @EnableAuto-Configuration and @componentscan with their default attributes.

* How to use property defined in application.properties file into your java class.

→ Using @Value annotation.

e.g @Value("${property.name}")
Private String value;

* @Restcontroller.

→ -It is convenience annotation for creating Restful controllers.
-It is specialization of @component and is auto detected through class path scanning.
-It adds @Controller & @ResponseBody annotations.
-It converts the response to JSON or XML.

* Difference b/w @RestController & @Controller.

→ - @Controller Map the model object to view or template and makes it human readable.
- @Restcontroller simply returns the object and object data is directly written into HTTP response as JSON or XML.

* Web Application vs REST API.

→ - Response from web application is generally view i.e. HTML + CSS + Javascript because they are intended for human viewers.
- REST API just returns data in the form of JSON or XML because most of the REST clients are programs.

* RequestMapping vs GetMapping.

→ - RequestMapping can be used with GET, POST, PUT and other request methods using attribute on the annotation.
- GetMapping is only an extension of RequestMapping which helps you to improve clarity on request. i.e. only for GET Request.

* Profiles in Spring Boot

→ When developing application, we typically deal with multiple enviroments such as Dev., QA and Prod. The configuration for these enviroments may be different
- To make this easy and clean, spring has provided profiles, to help seperate the configuration for each environment.
- So, instead of maintaining this programmatically, the properties can be kept in seperate files such as application-dev.properties & application-prod.properties.

→ The default properties points to the currently active profile using spring.profiles.active = prod/dev.

* Spring Actuator.

→ It is an additional feature that help us to monitor and manage our spring boot application on production.

- It includes Auditing, health & metrics gathering
- Actuator can be enable by adding dependedency: spring-boot-starter-actuator in pom.xml.
- Using spring Actuator, you can access flows like bean created, cpu usage, http hits to server.

* How to use Actuator.

① URL :- http://localhost:8080/actuator/health.
{ "status" : "UP" }

② URL :- http://localhost:8080/actuator
→ help for actuator urls/Endpoints.

③ URL :- http://localhost:8080/actuator/beans
→ Beans informations.

* Actuator Endpoints

→ ① By default Exposed endpoints can be seen at http://localhost:8080/actuator

② To explicity include/expose all endpoints use managment.endpoints.web.exposure.include = *

③ To expose only selected endpoints. (mentioned only).
management.endpoints.jmx.exposure.include = health, info, env, beans

④ To get environmental configuration about the server
- http://localhost:8080/actuator/env

⑤ To get all the spring beans loaded in the context.
- http://localhost:8080/actuator/beans.

* Enabling HTTP Trace.

→ - HTTP Trace:- All HTTP request and response captured by actuator
- Enabling HTTP Trace
① Before 2.2.x it can be achived by ~~enable~~ adding management.endpoints.web.exposure.include = *
And removed from versions later 2.2.x due to memory consumption
② To enable HTTP trace, just create the bean of "HttpTraceRepository. which is in memory repository.

e.g.

```
class Configuration {
    @Bean
    Public HttpTraceRepository httpTraceRepository() {
        return new InMemoryHttpTraceRepository();
    }
```

* How can we change Actuator Endpoints?

→ By Default:- url/actuator
- Add management.endpoints.web.base.path=/manage.

* Change management server port (Actuator)

→ management.~~endpoint~~ server.port = 8090

Page :
Date : / /

* Custom Endpoints for Actuator.

→ This can be achieved by adding following annotations:

① @Endpoint and @Component at class level

~~② @ReadEndpoint~~

② @ReadOperation for GET
@WriteOperation for POST
@DeleteOperation for DELETE on method level.

Custom Endpoints:- http://localhost:8080/actuator/{id}
↓ mentioned in
@Endpoint(id = "id")

* How to use YML file instead of properties.

→ ① Create YML file in resources folder
② Use @ConfigurationProperties(prefix = "yml")
@PropertySource(value = "class:application.yml", factory=
YMLPropertySourceFactory.class)
↓
Create PropertySource in class

* Project Lombok.

→ - It is used to reduce boilerplate code in Java.
- We can achived this by using some annotations provided by lombok.

* Hibernate

→ Hibernate is object-relational mapping tool (ORM) used to map java objects and database tables.
- It provides JPA implementation hence we can use JPA annotations & xml configurations to achieve this mapping.

* Why Hibernate?

→ ① Avoids JDBC coding.
② Supports HQL with Object-oriented.
③ Provides Transcation management implicity.
④ Exception Handling
⑤ support caching.

* Interfaces used in Hibernate :-

→ ① SessionFactory (org.hibernate.SessionFactory) :-
Instance of this used to retrive session object for database operation and it is one per database.

② Session (org.hibernate.session) :-
- Its factory for transcation, used to connect application with persistant store hibernate framework/DB.
- used to get physical connection with DB
- provides methods for CRUD operations.

③ Transcation (org.hibernate.Transcation) :-
This specifies single unit of work.

e.g

```
Sessionfactor f = metadata.getSessionFactoryBuilder().build();
Session s = f.openSession();
Transcation t = s.beginTranscation();

s.save(persistantobj);
t.commit();
f.close(); s.close();
```

* Annotations in Hibernate.

→ ① Entity :- used with model classes to specify that they are entity beans.
- javax.persistence.Entity.

② Table :- used to define corresponding table in database
- javax.persistence.Table.

③ Access :- Used to defined the access Type, either field or property. By default value is field.
- javax.persistence.Access.
- e.g. @Access (value = AccessType.PROPERTY)

④ Id :- used to defined the primary key in the entity.

⑤ @EmbeddedId :- used to defined composite primary key in entity bean.

⑥ Column :- used to defined column ~~na~~name in database table.

⑦ GeneratedValue :- used to defined strategy to be used for generation of primary key.
e.g. GenerationType.IDENTITY.

⑧ OneToOne :- used to define one-to-one mapping between two entity beans.

⑨ Cascade :- used to defined the cascading between two entity beans & is used with mappings.

⑩ PrimaryKeyJoinColumn :- used to define the property for foreign key.

* Mappings in Hibernate.

→ ① One-to-one Mapping :-

```
Syntax :-  class class1 {
@OneToOne(targetEntity = class2.class)
  private class1 class1; }
```

② ManyToOne Mapping :-

```
Syntax :- class class1{
  @ManyToOne(Cascade = CascadeType.ALL)
  private class2 c2; }
```

③ ManyToMany mapping

syntax:-

```
class class1 {
@ManyToMany (targetEntity = Degree.class, cascade =
                                    {CascadeType: ALL})
@JoinTable (name = "JoinTableName",   class1 p.k.
        joinColumn = {@JoinColumn (name = "JoinId")},
        inverseJoinColumns = {@JoinColumn (name = "
        class2 P.K. ⟶ inverseJoinId")})
private List<class2> c2;
```

* What are hibernation configuration file?

→ hibernate.cfg.xml

- configuration file contains database specific configurations and used to initialize sessionfactory.
- Database configuration are written inside session-factory xml tag. and to connect more than two database we need to create another session-factory tag.

* Hibernate Mapping file.

→ To map tables to java Entities, hibernate mapping file is used and it is ~~mapping~~ xml file.
- named as classname.hbm.xml

* steps to create sample App of Hibernate.

→ ① Create the persistent POJO ② Create mapping file ③ Create configuration file.
④ class for retrieving or storing the persistence POJO (DAO)
⑤ Run the application.

* Difference opensession and getcurrentsession.

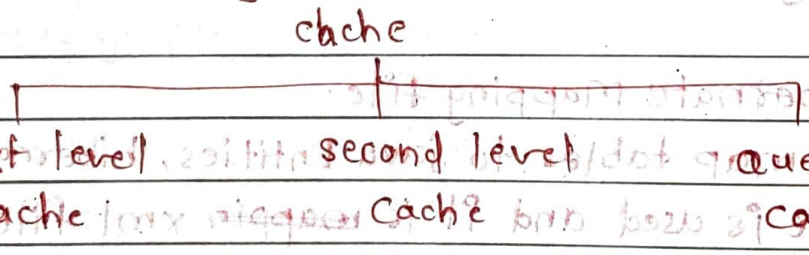
→ ① getCurrentSession() method returns the session bound to context.
- Even if we don't close the session when we close SessionFactory, it automatically get closed.
② openSession() method helps to open new session.
This session we have to close manually.

* Difference between session get() & load() method?

→ ① get() loads the data as soon as it is called
② load() returns proxy object & load data only when it is actually required
So load() is better as it supports lazy loading.
③ load() throws Exception when data is not found so we should use when data exists
④ get() when we want to sure that data is exists in database

* hibernate caching

→



- To reduce number of database queries & to make application faster.

* First-level cache:

→ - Hibernate first level cache is associated with Session object
- Hibernate first level cache is by default enabled and there no way to disable it.
- Still hibernate provides methods through which we can

delete selected objects from the cache or clear the cache completely.
- Any object cached in a session will not be visible to other session & when session is closed, all the objects will also be lost.

* Second level cache. (SessionFactory)
→ - Hibernate second level cache is disabled by default but we can enable it through configuration.
- Currently EHCache & Infispan provides implementation for Hibernate second level cache.

* EHCache Implementation
→ ① Add hibernate-ehcache dependency in your maven project or add corresponding jars.

```
<dependency>
    <groupId> org.hibernate </groupId>
    <artifactId> hibernate-ehcache </artifactId>
</dependency>
```

② Add ~~beto~~ below properties:

```
hibernate.cache.region.factory_class = org.hibernate.cache.
                ehcache.EhcacheRegionFactory
hibernate.cache.use_second_level_cache = true.
hibernate.cach.use.query_level-cache = true.
net.sf.ehcache.configurationResourceName = xml file
```

This file contains caching configuration for caching particular tables).

③ Add annotation @Cache to entity & add caching strategy to use.

```
@Cache (usage=CacheConcurrencyStrategy.READ_ONLY
        region="table").
```

* How to integrate Hibernate and Spring ?

→ Steps to integrate :-
① Add require dependencies in pom.xml
② Create entity, controller, service & DAO layers.

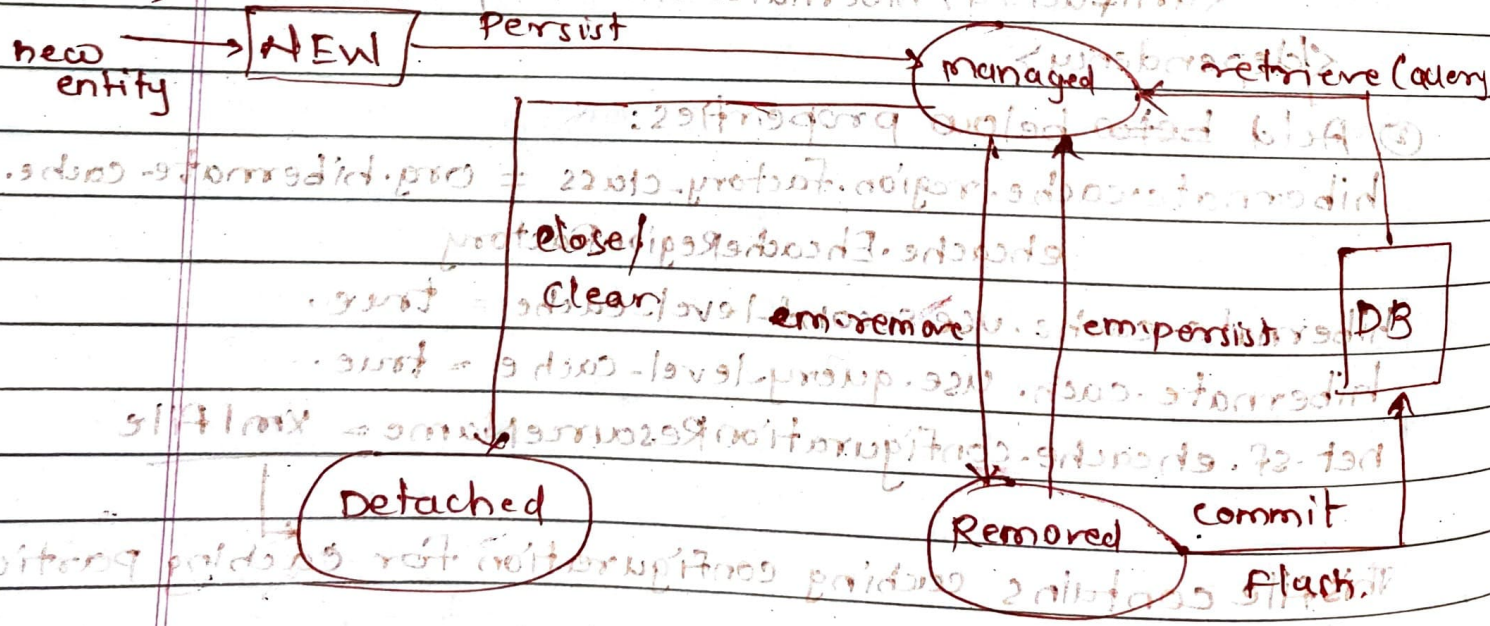
Required Dependencies :
① spring-boot-starter-data-jpa
② hibernate-core
③ hibernate-entitymanagers
④ To connect to database sql connector dependency.

* How to generate Hibernate SQL on console ?

→ Spring boot → spring.jpa.show-sql = true.
Spring → show-sql inside property tag of config file

* Entity Life cycle and states ?

→



* What Lazy and Eager loading in hibernate?

→ ① Lazy :- loads the data only when we explicitly call getter or size method.
- ManyToMany & OneToMany associates used lazy loading strategy by default.
- fetch = fetchType.LAZY;
- Initial load time is much smaller than Eager loading.

② Eager :- loads the data ~~happens when~~ at the time of their parents fetched.
- ManyToOne & OneToOne associates used ~~lazy~~ Eager loading by default.
- fetch = fetchType.EAGER;
- Loading takes too much time than lazy loading.

* HQL (Hibernate Query Language)

→ It is object-oriented query language, similar to SQL but instead of operating on tables & columns. it works with persist objects and their properties.

* SQL vs HQL.

→ - SQL is traditional query language that directly interacts with RDBMS & HQL is Java-based OOP language that uses the hibernate interface to convert the OOP code into query statement and then interacts with database.
- SQL deals with relationship between two tables or columns & HQL deals with relationship between two objects.
- SQL is faster than non-native HQL. But by setting correct cache size of query plan, you can make HQL works fast as SQL.

* Inversion of Control (IoC)

→ - It is also known as dependency injection container
- IoC is principle most often used in object-oriented programming, which tranfers the control of objects to container (framework).
- IoC container :
① Creates object of specified class (beans).
② injects all dependency objects through a constructor, a property or method at run time.
③ Disposes it at the appropriate time.

* Dependency Injection (DI)

→ - It is a pattern uses IoC principle, which transfers the control of objects to a container or framework.
- DI allows loose coupling of components & moves the the responsibility of managing components on the container.

* @Component

→ allows spring to automatically detect custom beans.

* @Autowired

→ - Helps to inject object dependency implicitly.
- It can't be used to inject primitive & string objects.

* @Qualifier

→ It is used to resolve autowiring conflicts, when there are multiple beans of same type in spring container.

* Bean Scopes :-

→ ① Singleton :- container creates a single instance of bean & it is by default

② Prototype :- Different instance every time when it is requested from container.

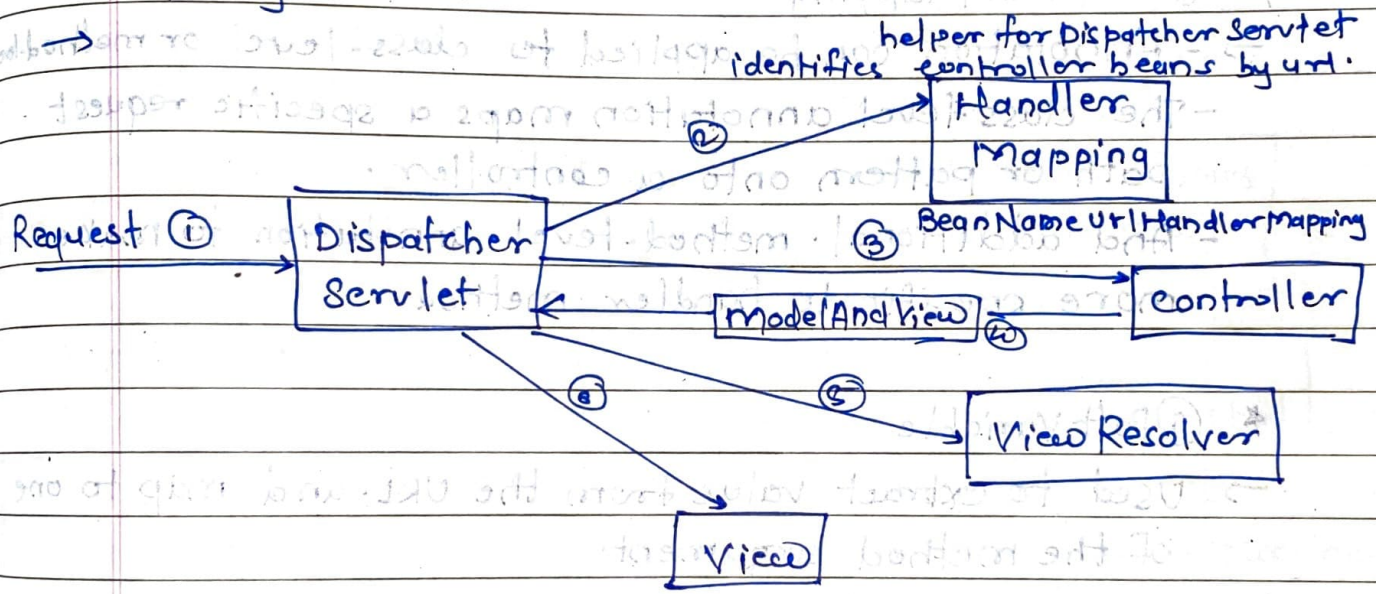
③ Request :- A Bean instance for single HTTP request

④ Session :- A Bean instance for HTTP session.

⑤ Global Session :- A Bean instance for an global Session

* Spring MVC flow.

→



① Once the request generated, it intercepted by the DispatcherServlet that works as front-end controller.

② DispatcherServlet get an entry of handler mapping from XML file & forwards the request to the controller.

③ Then controller returns the ModelAndView

④ The DispatcherServlet checks the entry of view resolver in XML file & invokes the specified view component.

* FrontController → DispatcherServlet (This class required to specify in web.xml file).

* Viewo Resolver → provides mapping b/w view names and actual views.

* InternalResourceViewoResolver
→ It is implementation of ViewoResolver interface in Spring MVC framework.
It resolves logical view names (like "hello") to internal physical resources/views like JSP files.

* @RequestMapping
→ - Annotation can be applied to class-level or method-level
- The class-level annotation maps a specific request path or pattern onto a controller.
- And additional method-level annotation to make more specific to handler method.

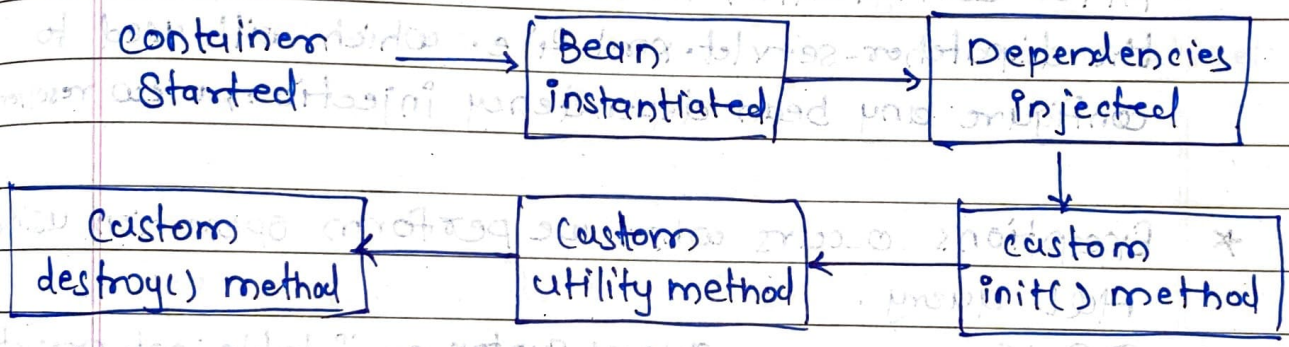
* @PathVariable
→ Used to extract value from the URL and map to one of the method argument.

* @RequestParam
→ Used to extract query parameters, form parameters and even files from the request.
e.g. @RequestParam(name="id") String id
- It is map to one of the method argument.

* @ModelAttribute
→ - Used as a part of spring MVC
① can be used to inject data objects in the model before JSP loads
② can be used to read data from existing model

* Spring Bean Life Cycle:-

→ - Bean life cycle is managed by spring container.
- When we run a program, first of all the spring container gets started.
- After that the container creates a instance of a bean as per the request and dependencies are injected.
- And finally the bean is destroyed when spring container is closed.
- So if we want to execute some code on bean instantiation and just after closing the spring container, then we can write that code inside the custom init() method & destroy() method.



- We need to annotate init() method by @PostConstruct annotation, init method gets called once container is started and we need to annotate destroy() method by @PreDestroy annotation.
- To invoke the destroy() method we have to call the close() method of configurableApplicationContext.

* Difference between ApplicationContext & BeanFactory.

→ - BeanFactory is basic version of IoC container and ApplicationContext extends the features of BeanFactory.
- BeanFactory loads beans on-demand by lazy-loading strategy while ApplicationContext loads all beans at startup by eager loading strategy.
- BeanFactory is light-weight as compared to ApplicationContext

* How to configure dispatcher servlet

→ - To configure dispatcher servlet, we can use web.xml file.
- We can configure dispatcher servlet as below in web.xml.

```
<servlet>                                   ← any name.
    <servlet-name> dispatcher </servlet-name>
    <servlet-class> package.className </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name> dispatcher </servlet-name>
    <url-pattern> *.do </url-pattern>
</servlet-mapping>
```

- After configuration in web.xml we need to create the dispatcher-servlet.xml file. which will used to configure any bean dependency injection, view resolver, etc.

* Exceptions occurs when we perform operation using HQL query.

→ ① SqlGrammer :- query syntax or if table not present
② IllegalArgumentException :- Wrong argument or fail to ~~recogize~~ recognize or incorrect format
③ EntityNotFoundException :- Entity didn't match
④ NonUniqueResultException :- result.unique() but no unique data present.
⑤ LazyInitialicationException :- tried to access elements from child which are lazily loaded without an active session.

* JDBC steps :-

→ ① Import JDBC packages
② Load and Register JDBC driver

e.g. DriverManager.registerDriver(new oracle.jdbc.driver.
oracleDriver());

OR

Class.forName("oracle.jdbc.driver.OracleDriver");

③ Open connection to the database

Connection conn = DriverManager.getConnection(URL, username, password);

④ Create a statement object to perform sql query.

Statement sql = conn.createStatement();

⑤ Execute the statement object & return a resultset

String sql_str = "select * from table";
ResultSet rSet = sql.execute(sql_str);

⑥ Process the resultset

```
while(rset.next()) {
    rSet.getString("name");
    rSet.getInt("mobile");
}
```

⑦ Close resultset and statement object.

sql.close(); } If not close, memory leaks can occur.
rSet.close();

⑧ Close connection.

conn.close();

* Prepared statement

→ PreparedStatement stmt = con.prepareStatement(SQLQuery);
ResultSet r = stmt.executeQuery();

* How can we execute Stored procedure in JDBC.

→ By using CallableStatement

e.g.
String sql = "{ call storedProcedureName(?,?)}";
CallableStatement cs = con.prepareCall(sql);

* Hibernate Dialect

→ - Hibernate uses "dialect" configuration to know which database we are using so that it can switch to the database specific SQL generator code whenever/ wherever necessary.
- i.e. it generates sql queries specific to our database.

* Autowiring modes

→ ① no:- default autowiring. It means no autowiring.
② byName:- injects object dependency according to name of bean. propertyName & beanName should be same. internally calls setter method.
③ byType:- injects object dependency by type. propertyName & beanName can be different. Internally calls setter method.
④ Constructor:- Injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
⑤ autodetect:- Spring tries to autowire by constructor. If it fails, tries to autowire by type.

S.D.S.
Page:
Date: / /

* Save()
→ - Stores an object into the database
- It will persist the given transient instance and returns generated id and return type is serializable.
- Calls either save() or update() on the basis of identifier exists or not. Supported by Hibernate.
- ~~eg~~. Can save object ~~within~~ within/outside Transcational boundries
e.g. Long id = (Long) session.save(entity);

* persist()
→ - It is similar to save() method and adds entity object to persistent context.
- It does not return anything: Return Type is void.
- supported by JPA.
e.g. session.persist(entity);
- can only save object within Transcational boundries.

* SaveORUpdate()
→ - save() generates new identifier & execute insert query.
- save() method insert an entry if identifier doesn't exist, else it will throw error
- SaveOrUpdate() does an insert or Update but doesn't throw any error.
-

* Merge()
→ - used when we want to change a detached entity into the persistent state again & it will automatically update the database.

* Hibernate object state

→ Transient → Persistent → Detached.

- Transient: new created object & not associated with Hibernate Session
- Persistent: after calling save() & associated with Hibernate session.
- Detached: after calling session close() method.

* Cascade Types in JPA

→ - When we perform some action on target entity, the same action will be applied to associated entity. And this can be achieve by cascading in JPA

- Cascade Types in JPA :-
  ① ALL ② PERSIS ③ MERGE ④ REMOVE
  ⑤ REFRESH ⑥ DETACH.

* Hibernate Cascade Type.

→ ① REPLICATE ② SAVE_UPDATE ③ LOCK.

* What is AJAX call?

→ Asynchronous Javascript And XML is used for
- Updating web page without reloading web page.
- Request data from a server after page load.
- Receive data from a server after page load.
- Send data to the server.

```
e.g. function loadDOC(){
 var xhttp = new XMLHttpRequest();
 xhttp.onChangemethod = function(){
      // response}
xhttp.open("GET", "ajax.txt", true);
 xhttp.send();
}
```

* Difference between REST and SOAP.

→

SOAP	REST
- Simple Object Access Protocol	- Representational State Transfer
- SOAP is protocol.	- REST is architectural style or pattern
- SOAP cannot make use of REST.	- REST can make use of SOAP as the underlying protocol for web services.
- SOAP requires more bandwidth.	- REST don't require much bandwidth.
- SOAP can only work with XML format.	- REST can work with plain text, HTML, XML, JSON.

* Difference between REST and AJAX.

→

REST	AJAX.
- Software architecture pattern. and it is a method for users to request information from servers.	- It is a method to dynamically load or update UI w.o reloading the page.

* Difference between Criteria and HQL.

→

Criteria	HQL.
- Criteria can only select data, we can't perform non-select operations using criteria queries.	- HQL can perform both select and non-select operations.
- Supports pagination	- ~~Doens~~ Doesn't support pagination.
- Criteria is safe for SQL injection.	

* Difference between Statment, PreparedStatement and CallableStatement.

→ ① Statement - Used to execute SQL queries.
② PreparedStatement - Used to execute dynamic or parameterized SQL queries.
③ CallableStatement - Used to execute stored procedures.

* Life Cycle of a Servlet

→ ① Servlet class is loaded.
② Servlet instance is created.
③ init method is invoked.
④ Service method is invoked.
⑤ destroy method is invoked

* GET vs POST vs PUT

→ ① GET
→ used to request data from a specific resource
→ GET carries request parameter appended in URL.

② POST
→ used by www to request data and accepts the data enclosed in the body of request message.
→ POST create new resource.

③ PUT
→ used to update resource arable on the server.
→ replaces existing data at target URL.

④ DELETE
→ requests the origin server to delete the resource identified by the URL.

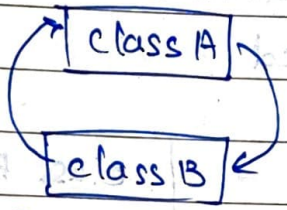
⑤ PATCH

→ used to modify resource where client sends partial data that is to be updated without modifying the entire Data.

* Circular dependency in Spring ?

→ - When two or more beans require instance of each other through constructor dependency injections, then is called circular dependency.

-



e.g.

```
class A {
    private B b;
    @Autowired
    public A (B b) {
        this.b = b;
    }
}
```

```
class B {
    private A a;
    @Autowired
    public B (A a){
        this.A = a;
    }
}
```

- Exception occured while circular dependency injection is BeanCurrentlyInCreationException.

* How to resolve circular dependency ?

→ ① Redesign :- change hierarchy.

② Use @Lazy Annotation.

→ Instead of fully initializing the both the beans, initialize one of the bean lazily.

→ It will create a proxy to inject into other bean. and it will be fully created when it is needed.

e.g. public class A {
private B b;
@Autowired
public A (@Lazy B b) {
this.B b = b;
}

(3) Use setter / field injection.
→ Using setter injection also beans will get created but ~~injection~~ dependencies are injected when they are actually needed.

e.g.

```
class A {
private B b;

@Autowired
public void setB (B b) {
  this.B = B;
}

public B getB() {
  return b;
}
```

```
class B {
  private A a;

  @Autowired
  public void setA (A a) {
    this.a = a;
  }

  Public A getA () {
    return a;
  }
```

(4) Use @PostConstruct
→ injecting a dependency using @Autowired on one of the beans, and then use a method annotated with @postconstruct to set the dependency.

e.g.

```
class A {
@Autowired
private B b;

@postConstruct
public void init() {
  b.setA(this);
}
```

```
public B getB() {
  return B;
}
```

class B {
private A a;

classB

public void setA (A a) {
this.a = a;
}.

⑤ Implement ApplicationContextAware and InitizingBean
→ If one of the beans implement ApplicationContextAware the bean has a access to spring context & can extract the other bean from there.
→ Implementing InitializingBean indicates that this bean has to some actions after all its properties have been set.

e.g.
class A implements ApplicationContextAware, InitializingBean{
private B b;
private ApplicationContext context;

public B getB() {
return b;
}.

@Override
Public void afterPropertiesSet() throws Exception {
b = context.getBean(B.class);
}

@Override
public void setApplicationContext(final ApplicationContext ctx){
context = ctx;
}

* @Immutable in Hibernate.

→ This annotation tells Hibernate that any updates to an ~~and~~ immutable entity should not be passed on to the database without giving any error.
- @Immutable can be placed on a collection too

* Hibernate N+1 problem.

→ - N+1 problem is performance issue in ORM that fires multiple select queries in a database for a single select query at application layer.
- i.e. N = number of records in table and for each record one selection query for mapped table.

eg.

```
* class A {
    private Long id;
    private string name;

    @ManyToMany(fetch = FetchType.LAZY)
    private List<B> ListB;

* Class B {
    private Long id;
    private String name;
}.
```

- From Above, A class is mapped to class B
- To fetch all data from A table, we can use below query:-

```
List<A> findAllBy();
```

- Equivalent Queries that ORM will execute are;
  (i) Select * from A;

ii) select * from B where id = <A_id>;

- So we need one select for "A" and N additional selects for fetching B for each A_id.
- This is N+1 problem in Hibernate and can be identify by enabling SQL logging.

* How to resolve N+1 problem in Hibernate.

→ - N+1 problem can be ~~achie~~ resolve by creating join tables and get the combined result in single query.

① Spring Data JPA approach.

→ using @EntityGraph or using select query with fetch join.

i.e.

(i) @Query("select a from A a left join fetch a.b")
List<A> findAll();

(ii) @EntityGraph(attributePaths = {"b"})
List<A> findAll();

② Hibernate Approach.

→ (i) "select *from A a fetch a.b b"; ← HQL.

(ii) Criteria criteria = session.createCriteria(A.class);
criteria.setFetchMode("b", fetchMode.EAGER);

* How to exclude any package without using the basepackage filter?

→ @SpringBootApplication(exclude = {A.class}).

* How to disable a specific auto-configuration class?

→ @EnableAutoConfiguration((exclude = {DataSourceAuto-configuration.class})

* Spring ~~Init~~ Initializer.

→ The Spring initializer is a web application that generates a spring boot project with everything we need to start it quickly.

* What is filter in spring boot?

→ - A filter is an object used to intercept the HTTP requests and responses of application.
- Purpose of filter is performing filtering on either the request to resource or on the response from a resource or both.

* How to add filter to an application?

→ ① By implementating filter interface.
② Using FilterRegistrationBean.
③ Using MVC controller. (@WebFilter)

* @PathVariable.

→ Used to extract the value from URL.
e.g. http://localhost:8080/url/pankaj
this name can be extracted using @PathVariable.

```
@GetMapping(path="/url/{name}")
public String getName(@PathVariable String name){
    return name;
}
```
→ Pankaj

* What is Swagger?

→ - Swagger is used to describe structure of the APIs.
- It is an open-source service provided in spring boot which make easier for the machines to find out API structure for web services.

S.D.S.
Page:
Date: / /

* What is Hot-swapping in spring-boot?
→ - It is a way to reload the changes without restarting Tomcat or Jetty server. IDE supports byte code hot swapping.
- If we make any changes that don't affect the method signature, it should load without any side effect.

* What is AOP?
→ - Aspect-oriented programming.
- AOP supplements object-oriented programming that aims to increase modularity.
- AOP breaks program logic into various parts, which are called concerns.

* Shutdown in actuator?
→ It is a management endpoint that allows for a smooth and proper shutdown of application.
- It is not authorized by default, but it can be enabled by using management.endpoint.shutdown.enabled = true.

* Can spring boot also be used to create non-web applications?
→ - Yes, spring boot supports both web & non-web application development.
- We need to remove web dependencies from classpath & the application context to create a non-web application.

* What is starter dependency in spring boot module?
→ ① Data JPA starter ② Test starter ③ Security starter
④ Web starter ⑤ Mail starter ⑥ Thymeleaf starter

* @Bean annotation:

→ - Bean annotation is applied to a method to specify that it return a bean to be managed by spring context/spring container.

- They usually used in configuration classes methods so that bean methods may reference other bean method by calling them directly.

* Spring Validation using spring boot annotation:

→ - Annotations that can be used for spring validation defined by Java Bean validation API.

① @NotEmpty :→ not null or not empty

② @NotBlank :→ not null and must contain atleast one non-whitespace character.

③ @NotNull → not null only.

④ @Email → valid email-Id.

⑤ @Min → field must be a number whose value must be higher than or equal to specified value.
e.g: @Min(value = 10) → value >= 10.

⑥ @Max → field must be a number whose value must be less than or equal to specified value
e.g @Max(value = 10) → value <= 10.

⑦ Size → field size falls within the specified range. Can be applied to strings, collections and Arrays.

* @Primary annotation.

→ - @Primary annotation is used to give higher preference to a bean, when there are multiple beans of same type.

- @Primary can be used on any class directly or on methods with @Bean or with @Component

Repository in JPA is central repository marker interface.
→ Repository<T, ID>.

* @Conditional annotation
→ - It is used to indicate whether a given component is eligible for registration based on a defined component
- @Conditional annotation can be used for following:
(i) whether property is available or not using enviromental variables irrespective of its value.
(ii) Like Profiles, condition whether a property value is available or not.
(iii) Conditions based on a Bean definition / Bean objects are present in spring Application Context.
(iv) Conditions based on some Resources are present in spring Application Context or not.

* CRUDRepository
→ - It is a base interface & extends Repository interface
- Provides CRUD operation.
- Return Type of saveAll() method is Iterable.

* JPARepository
→ - extends PagingAndSortingRepository that extends CrudRepository.
- provide CRUD and paging operations, and also additional methods like flush(), saveAndFlush() and deleteBatch().
- Return Type of saveAll() method is List.

* PagingAndSortingRepository
→ - Extension of crudRepository
- Provides addition methods to retrieve entities using paging and sorting abstractions.
→ methods: ① Page<T> p = findAll(Pagable page);
② Iterable<T> i = findAll(Sort sort);