

* 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 it 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 achieved by excluding spring-boot-starter-tomcat from starter-web dependency and adding another server dependency e.g. Starter-jetty.

Topic: Spring Boot

* 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(Datasource)~~

~~@EnableAutoConfiguration(exclude={Datasource})~~

* @SpringBootApplication

→ It is equivalent to @Configuration @EnableAutoConfiguration and @ComponentScan with their default attributes.

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

→ Using @Value annotation or of different types.

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 maps 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 environments such as Dev., QA and Prod. The configuration for these environments may be different.
- To make this easy and clean, spring has provided profiles, to help separate the configuration for each environment.
- So, instead of maintaining this programmatically, the properties can be kept in separate 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 helps us to monitor and manage our spring boot application on production.
- It includes Auditing, health & metrics gathering.
- Actuator can be enabled by adding dependency: `<dependency>

 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId> </dependency>`
- Using Spring Actuator, you can access flows like bean created, CPU usage etc via `http://localhost:8080/actuator`.

* How to use Actuator.

① URL:- `http://localhost:8080/actuator/health`.

② URL:- `http://localhost:8080/actuator/info`

→ 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 explicitly include/expose all endpoints use `management.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:

→ - `HTTPTrace`: - All HTTP request and response captured by actuator

- Enabling `HttpTraceRepository`

① Before 2.2.x It can be achieved by enabling 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 its in memory repository.

`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`

- * Custom Endpoints for Actuator
 - This can be achieved by adding following annotations:
 - ① `@Endpoint` and `@Component` at class level
 - ② `@ReadEndpoint`, `@WriteOperation` for GET
 - ③ `@WriteOperation` for POST
 - ④ `@DeleteOperation` for DELETE || on method level.

custom Endpoints:- `http://localhost:8080/actuator/{id}`

prefix is `actuator` and `{id}` is mentioned in `@Endpoint(id = "id")`

- * How to use YML file instead of properties
 - ① Create a YML file in resources folder
 - ② Use `@ConfigurationProperties(prefix = "yml")`
 - ③ `@PropertySource(value = "classpath:application.yml", factory = YMLPropertySourceFactory.class)`

Project configuration file (yml)

Create PropertySource in class

Project configuration file (yml) using annotations

- * Project Lombok
 - It is used to reduce boilerplate code in Java.
 - We can achieve 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 Transaction management implicitly.
- ④ Exception Handling
- ⑤ support caching.

* Interfaces used in Hibernate :-

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

② Session (`org.hibernate.Session`)

- Its factory for transaction used to connect application with persistent store hibernate framework/bb.
- used to get physical connection with DB
- provides methods for CRUD operations

③ Transaction (`org.hibernate.Transaction`)

This specifies a single unit of work to execute

e.g. if code of bb - connection is from

Session factor f = metadata.getSessionFactoryBuilder().build();

Session s = f.openSession();

Transaction t = s.beginTransaction();

t.begin();

s.save(persistentobj);

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 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 :- class1.class1 @OneToOne(targetEntity = class2.class2)
`private class1 class1;`

② ManyToOne Mapping :-

Syntax :- class2.class1 @ManyToOne(cascade = CascadeType.ALL)
`private class2 class2;`

③ ManyToMany mapping

Syntax - annotations bottom to top

```
class class1 {
```

Annotations of beans

```
@ManyToMany(targetEntity = Degree.class, cascade =
```

• cascade type will follow this rule, p(CascadeType ALL)

```
@JoinTable(name = "JoinTableName", class1 P.K.
```

• join column = {@JoinColumn(name = "JoinId")},

inverseJoinColumns = {@JoinColumn(name = "

class2 P.K. → inverseJoinId")})

```
private List<class2> c2;
```

Annotations of beans if top to bottom class1 P.K.

* What are hibernation configuration file?

→ hibernate.cfg.xml file is located at C:\boot\src\main\resources

- Configuration file contains database specific configurations and used to initialize session factory.

- 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's mapping.xml file.

Named as class-name.hbm.xml file.

* Steps to create sample App of Hibernate.

① Create the persistent POJO (student)

② Create mapping file ③ Create configuration file.

④ class for retrieving or storing the persistence POJO (DAO)

⑤ Run the application.

Annotations depend on beans

* Difference between `openSession` and `getCurrentSession`.

→ ① `getCurrentSession()` method returns the session bound to context.

→ Even if we don't close the session, when we close

(E.g.) A `SessionFactory`, it automatically gets closed.

② `openSession()` method helps to open new session.

In 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 ensure that data exists.

in database most data not found proxy

* hibernate caching

→ cache

hibernate first level, with second level / query cache

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

* First-level cache: get objects stored at appt.

→ - Hibernate first level cache is associated with Session

. Object mapping store (1) Object store (2)

→ - Hibernate first level cache is by default enabled and there is no way to disable it via configuration.

- 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. (Session factory)

- Hibernate second level cache is disabled by default but we can enable it through configuration.
- Currently EHCache & Infinispan provides implementation for Hibernate second level cache.

* EHCache Implementation in storage of both.

- Add hibernate-ehcache dependency in your maven project or add corresponding jars.

`<dependency>`

`<groupId> org.hibernate </groupId>`

`<artifactId> hibernate-ehcache </artifactId>`

`</dependency>`

- ② Add below properties:

`hibernate.cache.region.factory-class = org.hibernate.cache.EhcacheRegionFactory`

`hibernate.cache.use.second-level-cache = true`

`hibernate.cache.use.query-level-cache = true`

`net.sf.ehcache.configurationResourceName = xml file`

This file contains caching configuration for caching particular table(s).

- 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 :- a) at build config via

(1) Add required dependencies in pom.xml like

(2) Create entity, controller, service & DAO layers.

Required Dependencies

- ① spring-boot-starter-data-jpa
- ② hibernate-core
- ③ hibernate-entitymanager
- ④ To connect to database like connectors dependency.

* How to generate Hibernate SQL on console

→ Spring boot → Spring JPA → Show-SQL = true in application.properties

Spring → Show-SQL inside properties tag of config file

* Entity Life cycle and states

→ EntityState → transient (Transient Entity)

new entity → **NEW** → Persist

managed

Session

* 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 ~~Eager~~ 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 work fast as SQL.

* Inversion of Control (IoC) is found with

- It is also known as dependency injection container
- IoC is principle most often used in object-oriented programming, which transfers the control of objects to container (framework).
- IoC container : - (YSAJ. eptDtsf = dtsf) -
- 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) uses IoC principle

- It is a pattern uses IoC principle, which transfers the control of objects to container (framework).
- DI allows loose coupling of components & moves the responsibility of managing components on the side of container (Spring framework).
- # **@Component** → allows Spring to automatically detect custom beans.

* **@Autowire** is part of Spring framework

→ helps to inject object dependency implicitly.

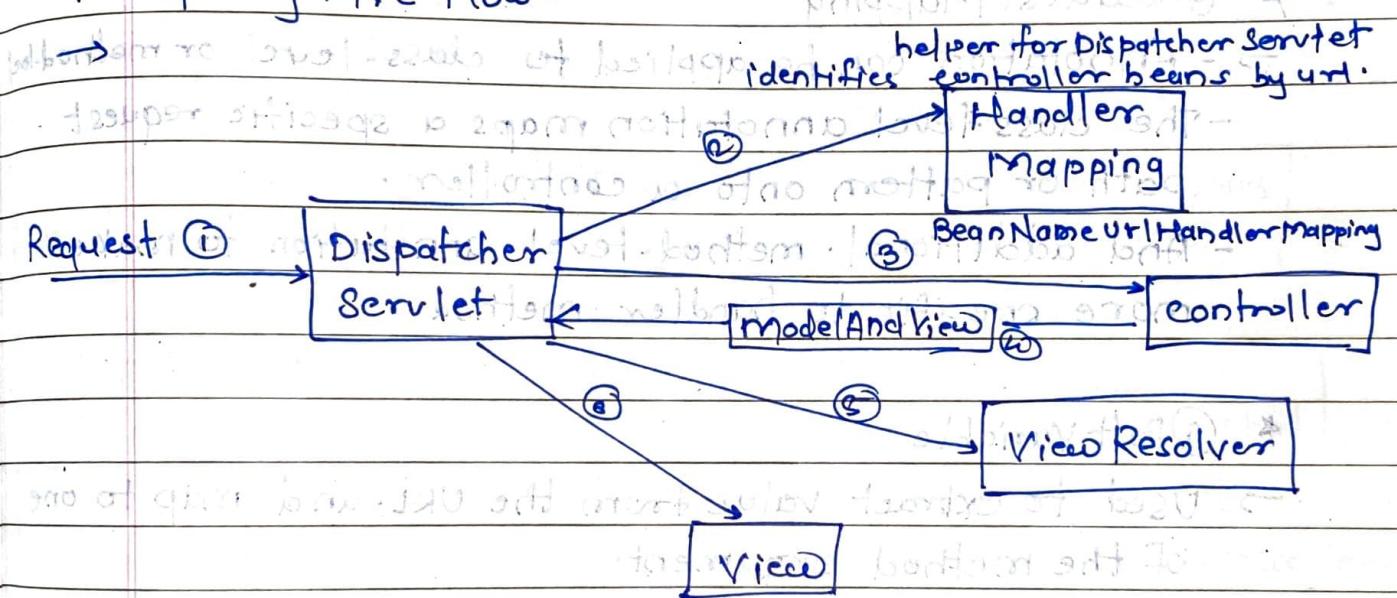
- It can't be used to inject primitive & string objects.

* **@Qualifier** associated with @Autowire

→ 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 instances 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 is intercepted by the DispatcherServlet that works as front-end controller.
- ② DispatcherServlet gets 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.
- * front controller → DispatcherServlet (This class required to specify in web.xml file).

* View Resolver → provides mapping between view names

↳ and actual views in real place → default is (D) →
the file pd-21-71-8 and

* InternalResourceViewResolver → default (②)

→ It is implementation of ViewResolver interface

↳ in Spring MVC framework and it is responsible for (③)

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 it more specific to handler method.

* @PathVariable

→ Used to extract value from the URL and map to one of the method arguments.

* @RequestParam, before a parameter name (④)

→ Used to extract query parameters, form parameters, and even files from the request.

e.g. @RequestParam(name = "id") String id

- It is mapped to one of the method arguments.

Given to values of id to model object (⑤)

* @ModelAttribute (in JMX of developer)

→ Used as a part of Spring MVC

① can be used to inject data objects in the model

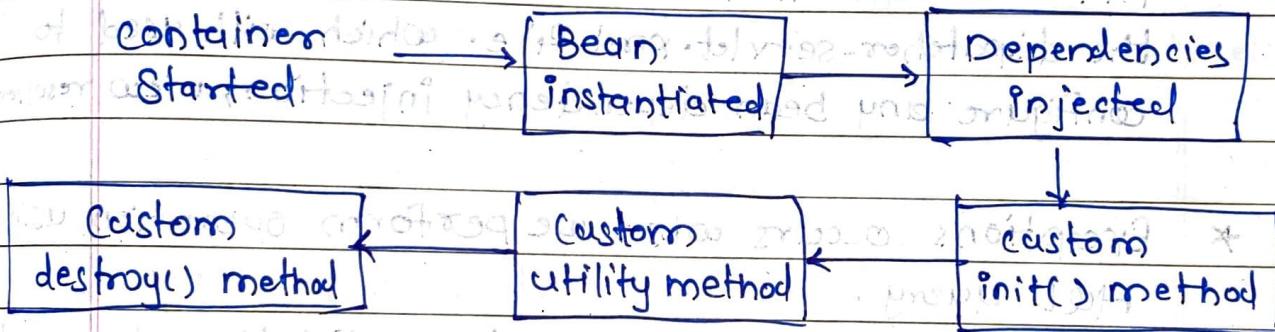
before JSP loads (model) → null or not found

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. (Initialize)
- After that, the container creates an instance of a bean as per the request, and dependencies are injected. (Spring bean creation)
- And finally the bean is destroyed when spring container is closed. (Spring bean destruction)
- 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.

Flow of bean life cycle does not happen as follows -



- We need to annotate an 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 non-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 properties *

→ - To configure dispatcher servlet, we can use web.xml file.

- We can configure dispatcher servlet as below in web.xml.

< servlet >

~~• dispatcher & by name~~ : class

→ Go to < servlet-name > dispatcher < /servlet-name >

→ < servlet-class > package.classname < /servlet-class >

→ < /servlet > is ported to add < /servlet > in < /servlet >

< servlet-mapping >

→ autowired < servlet-name > dispatcher < /servlet-name >

→ < url-pattern > do < /url-pattern > H

→ If in < servlet-mapping > do don't throw any error

→ ported to < portlet > portlet & portlet

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

* Exceptions occurs when we perform operation using

→ SQL query .

→ ① SQLGrammar :- Query syntax or if table not present

→ ② IllegalArgumentException :- Wrong argument or fail

→ ③ IOException :- failed to recognize or/incorrect format

→ ④ EntityNotFoundException :- Entity didn't match

→ ⑤ NonUniqueResultException :- result.unique() but

→ No unique data present in account

→ ⑥ LazyInitializationException :- tried to access elements

→ From child which are lazily loaded without an active

→ session. the best solution is to use session.getAttribute("ID")

→ ⑦ SQLException :- to insert null values in standard

→ ⑧ Import JDBC packages

→ ⑨ Load and Register JDBC driver

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

(C) JDBC: OracleDriver.registerDriver();

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

.OracleDriver registerDriver always goes after class.forName();

③ Open connection to the database/driver below pls

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

"jdbc:oracle:thin:@192.168.1.10:1521:orcl" "username", "password");
(102) 192.168.1.10 = db host 1521 = port 102 = port

④ Create a statement object to perform sql query.

Statement sql = conn.createStatement();

driver word of connection "oracle" 2326 return 111 ->

⑤ Execute the statement object & return a resultset

& string sqlStr = "select * from table";

ResultSet rset = sql.execute(sqlStr);

⑥ Process the resultset

while(rset.next()) {

String name = rset.getString("name");

int mobile = rset.getInt("mobile");

System.out.println(name + " " + mobile);

⑦ Close resultset and statement object.

sql.close(); if not close, memory leaks can occur

rset.close(); if not closed, occurs

buffer not freed, memory leak

⑧ close connection & release all resources

conn.close();

return and the program will proceed

return lines pd execution of short programs - fast to write

but need structure of code - good for reuse

- * Prepared statement
 - PreparedStatement stmt = con.prepareStatement(sqlQuery);
 - Resultset r = stmt.executeQuery();

("select * from Employee where id = ?")

- * How can we execute stored procedure in JDBC.

- By using CallableStatement of ResultSet

(e.g.)

(Procedure) String sql = "{call storedProcedureName(?,?)}";

CallableStatement cs = con.prepareCall(sql);

(map type conversion of test1 to test2 or others)

- * Hibernate Dialects

- - Hibernate uses "dialect" configuration to know which database we are using so that it can switch to the database specific SQL generator. (Code in hibernate) 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 -

- 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.

* Save()

state to be stored in DB

- - Stores an object into the database for account
- 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.
- e.g. can save object ~~within~~ within/outside Transactional boundaries
- e.g. long id = (Long) session.save(entity);

AS of JPA 2.6

* Persist()

AS of JPA 2.6

- - This is similar to save() method and adds entity object to persistent context and isn't bound.
- It does not return anything? Return type is void.
- supported by JPA 1.0, 1.1, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6
- e.g. session.persist(entity);
- can only save object within Transactional boundaries.

AS of JPA 2.6

* SaveOrUpdate()

AS of JPA 2.6

- - save() generates new identifier & execute insert query.
- save() method inserts an entry if identifier doesn't exist, else it will throw error.
- SaveOrUpdate() does an insert or update but doesn't throw any error.
- remove obj from db.

* Merge()

AS of JPA 2.6

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

(User, "test-user", "123")
Session.merge(user)

Session.flush()

* Hibernate object state

→ Transient → Persistent → Detached

new created Object & not associated with Hibernate Session

after calling save() & associate with session

after calling close() method.

detached from session

* 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 achieved by cascading in JPA.

- ① ALL
- ② PERSIST
- ③ MERGE
- ④ REMOVE
- ⑤ REFRESH
- ⑥ DETACH

* Hibernate cascade type

→ ① REPLICATE

② SAVE_UPDATE

③ Lock

* What is AJAX call?

Aynchronous JavaScript And XML is used for

- Updating webpage without reloading webpage.

→ 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.onreadystatechange = function(){}
```

```
// response
```

```
xhttp.open("GET", "ajax.txt", true);
```

```
xhttp.send();
```

```
}
```

* Difference between REST and SOAP.

→ SOAP

- Simple Object Access Protocol
- Representational State Transfer
- SOAP is protocol of REST → REST is architectural style.
- Web service is protocol or pattern
- SOAP cannot make use of REST can make use of SOAP as the underlying protocol

→ HTML for web services.

- SOAP requires more bandwidth.
- REST doesn't require much bandwidth.
- SOAP can only work with XML format.
- REST can work with plaintext, HTML, XML, JSON.

* Difference between REST and AJAX.

→ REST

AJAX

- Software architecture pattern.
- It is a method to dynamically load or update UI for users to request information from servers without reloading the page.

* Difference between Criteria and HAL.

→ Criteria

HAL

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

- Criteria is safe for SQL injection.

* Difference between Statement, PreparedStatement and CallableStatement.

→ ① Statement → Used to execute SQL queries.

→ ② PreparedStatement - Used to execute dynamic or crafting 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 vs DELETE

→ ① GET

→ used to request data from a specific resource.

→ GET carries request parameters appended in URL.

→ ② POST

→ used by WWW to request data and accepts the data enclosed in the body of request message.

→ POST creates new resource.

→ ③ PUT

→ used to update resource available on the server.

→ replaces existing data at target URL.

→ ④ DELETE

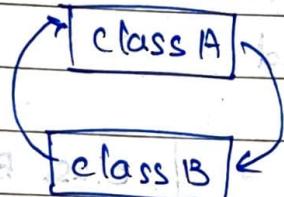
→ requests the origin server to delete the resource identified by the URL.

⑤ PATCH

→ used to modify resources 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 it is called circular dependency.



e.g. class A in Spring

```

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

class B in Spring

```

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

- Exception occurred while circular dependency injection is BeanCurrentlyInCreationException.

* How to resolve circular dependency?

→ ① Redesign or 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.