Institute for Advanced Computing And Software Development (IACSD)
Akurdi, Pune

Advanced Java

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,

Nigdi Pradhikaran, Akurdi, Pune - 411044.

IACSD Advance Java

Introduction to Enterprise Java

Web application

The application which can be developed using web components like **HTML, JSP, JS, XML, Servlet** etc. is called web application. A web application is running on particular server.

Client-Server Architecture in java

Client

The media which is use to send request on server is called Client and receive request from server is called client.Our browser will act as client.

Server

The media which is use to receive request from client and give response to client is called Server.

The main purpose of the server is to pick up the request from client identify the requested and dispatch dynamic response to client. **Protocol**

The main job of protocol in client-server architecture is to carry the request data from client to server and carry response from server to client. e.g. **TCP/IP,HTTP,FTP,SMTP etc..**

What is HTTP

HTTP stands for hyper text transfer protocol which is use to exchange data over the web.

HTTP is stateless protocol means every request treated as new request on server. HTTP unable to manage clients previous request data.

Examples of Servers

1. Apache Tomcat
2. Weblogic
3. Websphere
4. Glashfish

What is Web Server?

Web server is a server which will provide an environment to execute web applications only like Servlet, JSP ,Html ... etc.

What is Application Server

Application server is a server that will provide an environment to execute both web applications and distributed applications like Servlet, JSP and EJB etc...

What is a Web Container?

It provides runtime environment for JEE applications.

Web Container will perform following action

1. Life cycle management
2. Multi threaded support
3. Security

4. It support Connection pooling, Transaction Management, messaging, clustering, load balancing and persistence. etc...

Servlet

A servlet is a Java™ technology-based Web component, managed by a container that generates dynamic content. Like other Java technology-based components, servlets are platform-independent Java classes that are compiled to platform-neutral byte code that can be loaded dynamically into and run by a Java technology-enabled Web server. Containers, sometimes called servlet engines, are Web server extensions that provide servlet functionality. Servlets interact with Web clients via a request/response paradigm implemented by the servlet container. The servlet API is a standard Java extension and is therefore portable and platform-neutral. Servlets are secure, since they operate within the context of a security manager.

What is a Servlet Container?

The servlet container is a part of a Web server or application server that provides the network services over which requests and responses are sent. A servlet container also contains and manages servlets through their lifecycle. All servlet containers must support HTTP as a protocol for requests and responses, with additional protocols such as HTTPS (HTTP over SSL) etc... A servlet container may place security restrictions on the environment in which a servlet executes.

HTTP Basics

Servlets use the HTTP protocol, so a basic understanding of the protocol is assumed when using servlets.

- Requests, Responses, and Headers
- HTTP is a simple, stateless protocol. A client, such as a web browser, makes a request, the web server responds, and the transaction is done.
- When the client sends a request, the first thing it specifies is an HTTP command, called a method, that tells the server the type of action it wants performed.
- This first line of the request also specifies the address of a document (a URL) and the version of the HTTP protocol it is using.

  For example: GET /intro.html HTTP/1.0
- This request uses the GET method to ask for the document named intro.html, using HTTP Version 1.0.
- After sending the request, the client can send optional header information to tell the server extra information about the request, such as what software the client is running and what content types it understands.
- After the client sends the request, the server processes it and sends back a response. The first line of the response is a status line that specifies the version of the HTTP protocol the server is using, a status code, and a description of the status code.For example:HTTP/1.0 200 OK. This status line includes a status code of 200, which indicates that the request was successful, hence the description "OK". Another common status code is 404, with the description "Not Found"—as you can guess, this means that the requested document was not found.

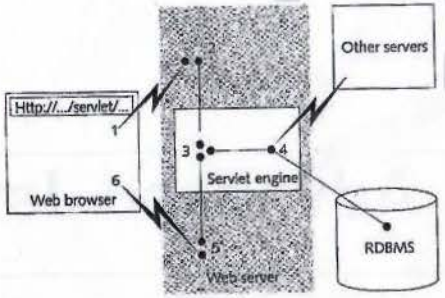
GET and POST methods

The GET and POST methods are used to get information and upload information to the webserver respectively. GET actually has the ability to pass information to the server and because of this is often used incorrectly. Many servers will limit the size of the url for a GET request to only 240 characters.POST is used for sending (posting) information to the server. It generally has an unlimited length.GET and POST are generally the only methods we will be dealing with in this unit, and generally speaking, are the only two methods most developers need.

Other methods

- In addition to GET and POST, there are several other lesser-used HTTP methods.
- There's the HEAD method, which is sent by a client when it wants to see only the headers of the response, to determine the document's size, modification time, or general availability.
- There's also PUT, to place documents directly on the server, and
- DELETE, to do just the opposite. These last two aren't widely supported due to complicated policy issues.
- The TRACE method is used as a debugging aid—it returns to the client the exact contents of its request.
- Finally, the OPTIONS method can be used to ask the server which methods it supports or what options are available for a particular resource on the server.

How a request is handled ?



1. A client (e.g., a Web browser) accesses a Web server and makes an HTTP request.
2. The request is received by the Web server and handed off to the servlet container.
3. Note : The servlet container can be running in the same process as the host Web server, in a different process on the same host, or on a different host from the Web server for which it processes requests.
4. The servlet container determines which servlet to invoke based on the configuration of its servlets, and calls it with objects representing the request and response.

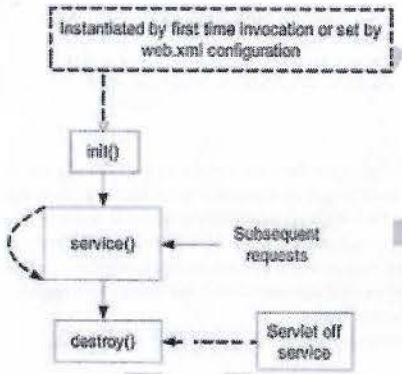
5. The servlet uses the request object to find out who the remote user is, what HTTP POST parameters may have been sent as part of this request, and other relevant data. The servlet performs whatever logic it was programmed with, and generates data to send back to the client. It sends this data back to the client via the response object.
6. Once the servlet has finished processing the request, the servlet container ensures that the response is properly flushed, and returns control back to the host Web server.

The Servlet Interface

Servlet interface is the central abstraction of the Java Servlet API. All servlets implement this interface either directly, or more commonly, by extending a class that implements the interface. The two classes in the Java Servlet API that implement the Servlet interface are GenericServlet and HttpServlet. For most purposes, Developers will extend HttpServlet to implement their servlets.

Servlet Life Cycle

A servlet is managed through a well defined life cycle that defines how it is loaded and instantiated, is initialized, handles requests from clients, and is taken out of service. This life cycle is expressed in the API by the init, service, and destroy methods of the javax.servlet.Servlet interface that all servlets must implement directly or indirectly through the GenericServlet or HttpServlet abstract classes.



- **Loading and Instantiation**

The servlet container is responsible for loading and instantiating servlets. The loading and instantiation can occur when the container is started, or delayed until the container determines the servlet is needed to service a request.

When the servlet engine is started, needed servlet classes must be located by the servlet container. The servlet container loads the servlet class. After loading the Servlet class, the container instantiates it for use.

- **Initialization**

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data,

initialize resources and perform other one-time activities. The container initialize the servlet instance by calling the init method of the Servlet interface with a unique (per servlet declaration) object implementing the ServletConfig interface.

- **Request Handling**

After a servlet is properly initialized, the servlet container may use it to handle client requests. Requests are represented by request objects of type ServletRequest. The servlet fills out response to requests by calling methods of a provided object of type ServletResponse. These objects are passed as parameters to the service method of the Servlet interface.

In the case of an HTTP request, the objects provided by the container are of types HttpServletRequest and HttpServletResponse.

- **End of Service**

The servlet container is not required to keep a servlet loaded for any particular period of time. When the servlet container determines that a servlet should be removed from service, it calls the destroy method of the Servlet interface to allow the servlet to release any resources it is using and save any persistent state. After the destroy method completes, the servlet container must release the servlet instance so that it is eligible for garbage collection.

What is Request?

The request object encapsulates all information from the client request. In the HTTP protocol, this information is transmitted from the client to the server in the HTTP headers and the message body of the request.

Request parameters for the servlet are the strings sent by the client to a servletcontainer as part of its request. The container populates the parameters from the URI query string and POST-ed data. The parameters are stored as a set of name-value pairs. The following methods of theServletRequest interface are available to access parameters:

- getParameter
- getParameterNames
- getParameterValues
- getParameterMap

- **Headers**

A servlet can access the headers of an HTTP request through the following methods of the HttpServletRequest interface:

- getHeader
- getHeaders
- getHeaderNames

- **Attributes**

Attributes are objects associated with a request. Attributes may be set by the container to express information that otherwise could not be expressed via the API, or may be set by a servlet to communicate information to another servlet (via theRequestDispatcher). Attributes are accessed with the following methods of theServletRequest interface:

- getAttribute

- getAttributeNames
- setAttribute

- **Request Path Elements**

The request path that leads to a servlet servicing a request is composed of many important sections.
The following elements are obtained from the request URI path:

- **Context Path:** The path prefix associated with the ServletContext that this servlet is a part of.
- **Servlet Path:** The path section that directly corresponds to the mapping which activated this request.
- **PathInfo:** The part of the request path that is not part of the Context Path or the Servlet Path.

The following methods exist in the HttpServletRequest interface to access this information:

- getContextPath
- getServletPath
- getPathInfo

requestURI = contextPath + servletPath + pathInfo

- **Lifetime of the Request Object**

Each request object is valid only within the scope of a servlet's service method, or within the scope of a filter's doFilter method, unless the asynchronous processing is enabled for the component. Containers commonly recycle request objects in order to avoid the performance overhead of request object creation.

HTTP request types in servlets

The GET and POST requests are to be handled by the doGet and doPost methods of your servlet class that extends the abstract HttpServlet class. These methods are automatically called by the HttpServlet class's service method, which is called when a request arrives at the server. The method service performs the following steps in order:

- determines the request type
- calls the appropriate method (doGet or doPost)
  There is no need for you to call the service method in your servlets.Every call to doGet and doPost for an HttpServlet receives an object that implements the interface HttpServletRequest. This object contains the request from the client. A variety of methods are available in this interface and the ServletRequest interface (super-interface of HttpServletRequest) to enable the servlet to process the client's request, getContextPath()
- Cookie[] getCookie()
- HTTPSessiongetSession()
- String getParameter(String name)
  The web server that executes the servlet creates an HttpServletResponse object. The web server passes the HttpServletResponse object to the servlet's service method (which, in turn, passes it to doGet or doPost). This object contains the response to the client. A variety of methods are available in this interface and the ServletResponse (super-interface of HttpServletResponse) to formulate the response to the client, egs
- ServletOutputStreamgetOutputStream()
- PrintWriter getWriter()

- void setContentType(String type)
- Apart from the doGet and doPost methods, the abstract class HttpServlet also provides methods such as doDelete (for client to remove documents from the server) and doPut (for client to place documents onto the server).

ServletContext

The ServletContext interface defines a servlet's view of the Web application within which the servlet is running. The Container Provider is responsible for providing an implementation of the ServletContext interface in the servlet container. Using theServletContext object, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can access. A ServletContext is rooted at a known path within a Web server. ServletContext object is global to entire web application. Object of ServletContext will be created at the time of web application deployment. It's *Scope is as* long as web application is executing, ServletContext object will be available, and it will be destroyed once the application is removed from the server.

Context Attributes

A servlet can bind an object attribute into the context by name. Any attribute bound into a context is available to any other servlet that is part of the same Web application. The following methods of ServletContext interface allow access to this functionality:

- setAttribute
- getAttribute
- getAttributeNames
- removeAttribute

ServletConfig

Servlet Container creates ServletConfig object for each Servlet during initialization, to pass information to the Servlet. This object can be used to get configuration information such as parameter name and values from deployment descriptor file(web.xml).ServletConfig object is one per servlet class.Object of ServletConfig will be created during initialization process of the servlet. This Config object is public to a particular servlet only It's Scope is as long as a servlet is executing, ServletConfig object will be available, it will be destroyed once the servlet execution is completed. We should give request explicitly, in order to create ServletConfig object for the first time.
The following methods of ServletConfiginterface allow access to the functionality:

- **getInitParameter**
- **getServletName**
- **getServletContext**

What is Response?

The response object encapsulates all information to be returned from the server to the client. In the HTTP protocol, this information is transmitted from the server to the client either by HTTP headers or the message body of the request.

Lifetime of the Response Object

Each response object is valid only within the scope of a servlet's service method, or within the scope of a filter's doFilter method. Containers commonly recycle response objects in order to avoid the performance overhead of response object creation.

Sessions

The Hypertext Transfer Protocol (HTTP) is by design a stateless protocol. To build effective Web applications, it is imperative that requests from a particular client be associated with each other. Many strategies for session tracking have evolved overtime. The following are some approaches for tracking a user's sessions. The standard name of the session tracking cookie must be JSESSIONID. Containers may allow the name of the session tracking cookie to be customized through container specific configuration.

What is a Session?

Session is a conversational state between client and server and it can consists of multiple request and response between client and server. Since HTTP and Web Server both are stateless, the only way to maintain a session is when some unique information about the session is passed between server and client in every request and response.

HTTP protocol and Web Servers are stateless, what it means is that for web server every request is a new request to process and they cant identify if its coming from client that has been sending request previously.

But sometimes in web applications, we should know who the client is and process the request accordingly. For example, a shopping cart application should know who is sending the request to add an item and in which cart the item has to be added or who is sending checkout request so that it can charge the amount to correct client.

What is the need of session tracking?

1. To identify the client among multiple clients
2. To remember the conversational state of the clnt(eg : list of the purchased books/ shopping cart/bank acct details/stocks) throughout current session

session = Represents duration or time interval
default session timeout for Tomcat =30 minutes

Session Consists of all requests/resps coming from/ sent to SAME client from login to logout or till session expiration timeout.

There are several techniques for session tracking.
J2EE specific techniques :

- **Using Cookies**

Cookie,is a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management. Cookie is a key value pair of information.
A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number. The servlet sends cookies to the browser by using the

addCookie() method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each. The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the getCookies() method. Several cookies might have the same name but different path attributes.
This should be saved by the browser in its space in the client computer. Whenever the browser sends a request to that server it sends the cookie along with it. Then the server can identify the client using the cookie.
In java, following is the source code snippet to create a cookie:
Cookie cookie = new Cookie("userID", "7456");
res.addCookie(cookie);
Session tracking is easy to implement and maintain using the cookies. Disadvantage is that, the users can opt to disable cookies using their browser preferences. In such case, the browser will not save the cookie at client computer and session tracking fails.

What is a cookie?

Cookie is small amount of text data.

Created by -- server (servlet or JSP prog or WC) & downloaded (sent) to clnt browser---within response header

Cookie represents data shared across multiple dyn pages from the SAME web appln.(meant for the same client)

1. Create cookie/s instance/s
javax.servlet.http.Cookie(String cName,String cVal)

2.Add the cookie/s to the resp hdr.
HttpServletResponse API :
void addCookie(Cookie c)

3. To retrieve the cookies :
HttpServletRequest :
Cookie[] getCookies()

4.Cookie class methods :
String getName()
String getValue()
void setMaxAge(int ageInSeconds)
def age =-1 ---> browser stores cookie in cache
=0 ---> clnt browser should delete cookie
>0 --- persistent cookie --to be stored on clnt's hard disk.

int getMaxAge()

Disadvantages of pure cookie based scenario

0. Web developer (servlet prog) has to manage cookies.

1. Cookies can handle only text data : storing Java obj or bin data difficult.

2. As no of cookies inc., it will result into increased net traffic.

3. In cookie based approach : entire state of the clnt is saved on the clnt side. If the clnt browser rejects the cookies: state will be lost : session tracking fails.

- **Hidden Fields**

<INPUT TYPE="hidden" NAME="technology" VALUE="servlet">

Hidden fields like the above can be inserted in the webpages and information can be sent to the server for session tracking. These fields are not visible directly to the user, but can be viewed using view source option from the browsers. This type doesn't need any special configuration from the browser of server and by default available to use for session tracking. This cannot be used for session tracking when the conversation included static resources like html pages.

- **URL Rewriting**

Original URL: http://server:port/servlet/ServletName

Rewritten URL: http://server:port/servlet/ServletName?sessionid=1234

When a request is made, an additional parameter is appended with the URL. In general, additional parameters will be session id or sometimes the user id. It will suffice to track the session. This type of session tracking doesn't need any special support from the browser. Disadvantage is, implementing this type of session tracking is tedious. We need to keep track of the parameter as a chain link until the conversation completes and also should make sure that, the parameter doesn't clash with other application parameters.

Session tracking API

Session tracking API is built on top of the first three methods. This is inorder to help the developer to minimize the overhead of session tracking. This type of session tracking is provided by the underlying technology. Lets take the java servlet example. Then, the servlet container manages the session tracking task and the user need not do it explicitly using the java servlets. This is the best of all methods, because all the management and errors related to session tracking will be taken care of by the container itself. Every client of the server will be mapped with a javax.servlet.http.HttpSession object. Java servlets can use the session object to store and retrieve java objects across the session. Session tracking is at the best when it is implemented using session tracking api.

How to access HttpSession object?

Every request is associated with an HttpSession object. It can be retrieved using getSession(booleancreateFlag) available in HttpServletRequest. It returns the current HttpSession associated with this request or, if there is no current session and create is true, and then returns a new session. A session can be uniquely identified using a unique identifier assigned to this session, which is called session id. getId() gives you the session id as String.

Method isNew()returns true if the client does not know about the session or if the client chooses not to join the session. getCreationTime() returns the time when this session was created. getLastAccessedTime() returns the last time the client sent a request associated with this session.

How to store data in session?

Once you have got access to a session object, it can be used as a HashTable to store and retrieve values. It can be used to transport data between requests for the same user and session. setAttribute(String name, Object value) adds an object to the session, using the name specified. Primitive data types cannot be bound to the session.

Note that your object needs to implement Serializable interface if you are going to store it in session and carry it over across different web servers.

How to retrieve data from session?

getAttribute(String name) returns the object bound with the specified name in this session.

How to invalidate a session object?

By default every web server will have a configuration set for expiry of session objects. Generally it will be some xxxx seconds of inactivity. That is when the user has not sent any request to the server for the past xxxx seconds then the session will expire. When a session expires, the HttpSession object and all the data it contains will be removed from the system. When the user sends a request after the session has expired, server will treat it as a new user and create a new session.

Apart from that automatic expiry, it can also be invalidated by the user explicitly. HttpSession provides a method invalidate() this unbinds the object that is bound to it. Mostly this is used at logout. Or the application can have an absurd logic like after the user logins he can use the application for only 30 minutes. Then he will be forced out. In such scenario you can use getCreationTime().

1. Get Http Session object from WC

API of HttpServletRequest ---

HttpSession getSession()

Meaning --- Servlet requests WC to either create and return a NEW HttpSession object(for new clnt) or ret the existing one from WC's heap for existing client.

HttpSession --- i/f from javax.servlet.http

In case of new client :

HttpSession<String,Object> --empty map

String,Object ---- (entry)= attribute

OR

HttpSession getSession(boolean create)

2. : How to save data in HttpSession?(scope=entire session)

API of HttpSession i/f

public void setAttribute(String attrName,Object attrVal)

eg : hs.setAttribute("clnt_info",validatedCustomer);//no javac err

attribute : server side object ---server side entry (key n value pair) --map

equivalent to map.put(k,v)
eg : hs.setAttribute("cart",l1);

3. For retrieving session data(getting attributes)
public Object getAttribute(String attrName) //key
eg : Customer cust=(Customer) hs.getAttribute("clnt_info");

4. To get session ID (value of the cookie whose name is jsessionid -- unique per client by WC)
String getId()

4.5 How to remove attribute from the session scope?
public void removeAttribute(String attrName)
eg : hs.removeAttribute("clnt_info");

5. How to invalidate session?
HttpSession API
public void invalidate()
(WC marks HS object on the server side for GC ---BUT cookie is NOT deleted from client browser)

6. HttpSession API
public boolean isNew()
Rets true for new client & false for existing client.

7.How to find all attr names from the session ?
public Enumeration<String> getAttributeNames()
--rets java.util.Enumeration of attr names.

8. Default session timeout value for Tomcat = 30 mins
How to change session timeout ?
HttpSession i/f method
public void setMaxInactiveInterval(int secs)
eg : hs.setMaxInactiveInterval(300); --for 5 mins .

OR via xml tags in web.xml
<session-config>
<session-timeout>5</session-timeout> : unit : min
</session-config>

What is an attribute ?
attribute = server side object(entry/mapping=key value pair)

Who creates server side attributes ? -- web developer (servlet or JSP prog)

Each attribute has --- attr name(String) & attr value (java.lang.Object)
Attributes can exist in one of 3 scopes --- req. scope,session scope or application scope
1. Meaning of req scoped attr = attribute is visible for current req.
2. Meaning of session scoped attr = attribute is visible for current session.(shared across multiple reqs coming from SAME client)
3. Meaning of application scoped attr = attribute is visible for current web appln.(shared across multiple reqs from ANY client BUT for the SAME web application)

Page Navigation Techniques
Page Navigation=Taking the user from 1 page to another page.

2 Ways
1. Client Pull
Taking the client to the next page in the NEXT request (coming all the way from client)
1.1 User takes some action --eg : clicking on a button or link & then client browser generates a new URL to take the user to the next page.

1.2 Redirect Scenario
User doesn't take any action. Client browser automatically generates new URL to take user to the next page.(next page can be from same web application , or diff web application on same server or any web page on any server)

API of HttpServletResponse i/f
public void sendRedirect(String redirectURL) throws IOException
eg : For redirecting client from Servlet1 (/s1) to Servlet2 (/s2) , use
response.sendRedirect("s2");

If the response already has been committed(pw flushed or closed) , this method throws(WC) an IllegalStateException.(since WC can't redirect the client after response is already committed)

2. Server Pull.
Taking the client to the next page in the SAME request.
Also known as resource chaining or request dispatching technique.
Client sends the request to the servlet / JSP. Same request can be chained to the next page for further servicing of the request.

Steps-
1. Create RequestDispatcher object for wrapping the next page(resource --can be static or dynamic)
API of ServletRequest
javax.servlet.RequestDispatcher getRequestDispatcher(String path)

2.Forward scenario
API of RequestDispatcher
public void forward(ServletRequest req,ServletResponse res)

This method allows one servlet to do initial processing of a request and another resource to generate the response. (i.e division of responsibility)

Uncommitted output in the response buffer is automatically cleared before the forward.

If the response already has been committed(pw flushed or closed) , this method throws an IllegalStateException.

Limitation --only last page in the chain can generate dynamic response.

3. Include scenario
API of RequestDispatcher
public void include(ServletRequest req,ServletResponse res)

Includes the content of a resource @run time (servlet, JSP page, HTML file) in the response. -- server-side includes.

Limitation -- The included servlet/JSP cannot change the response status code or set headers; any attempt to make a change is ignored.

JSP

Java Server Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content from servlets. You simply write the regular HTML in the normal manner. Separating the static HTML from the dynamic content provides a number of benefits over servlets alone, and the approach used in Java Server Pages offers several advantages over competing technologies such as ASP, PHP.
JSP is widely supported and thus doesn't lock you into a particular operating system or Web server and that JSP gives you full access to servlet and Java technology for the dynamic part, rather than requiring you to use an unfamiliar and weaker special-purpose language.
Although what you write often looks more like a regular HTML file than a servlet, behind the scenes, the JSP page is automatically converted to a normal servlet, with the static HTML simply being printed to the output stream associated with the servlet's service method. This translation is normally done the first time the page is requested.
Aside from the regular HTML, there are three main types of JSP constructs that you embed in a page: scripting elements, directives, and actions.

Scripting Elements [Scriptlets]

JSP scripting elements let you insert code into the servlet that will be generated from the JSP page. There are three forms:

1. Expressions of the form <%= expression %>, which are evaluated and inserted into the servlet's output
2. Scriptlets of the form <% code %>, which are inserted into the servlet's _jspService method (called by service)
3. Declarations of the form <%! code %>, which are inserted into the body of the servlet class, outside of any existing methods.
4. Scriptlets have access to the same automatically defined variables as expressions (request, response, session, out, etc). So, for example, if you want output to appear in the resultant page, you would use the out variable, as in the following example.

```
<%
String queryData = request.getQueryString(); out.println("Attached GET data: " + queryData);
%>
```

JSP Expressions

A JSP expression is used to insert values directly into the output. It has the following form: <%= Java Expression %>
The expression is evaluated, converted to a string, and inserted in the page. For example, the following shows the date/time that the page was requested:
Current time: <%= new java.util.Date() %>.

JSP Declarations

A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class (outside of the _jspService method that is called by service to process the request). A declaration has the following form:
<%! Java Code %>
For example, a JSPfragment that prints the number of times the current page has been requested since the server was,
<%! privateintaccessCount = 0; %>

Accesses to page since server reboot:
<%= ++accessCount %>

Predefined Variables /implicit objects in jsp -

To simplify code in JSP expressions and scriptlets, you are supplied with some automatically defined variables, sometimes called implicit objects.

1. **request**

This variable is the HttpServletRequest associated with the request;it gives you access to the request parameters, the request type (e.g., GET or POST), and the incoming HTTP headers (e.g., cookies).

2. **response**

This variable is the HttpServletResponse associated with the response to the client.

3. **out**

This is the PrintWriter used to send output to the client. However, to make the response object useful, this is a buffered version of Print- Writer called JspWriter.

4. **session**

This variable is the HttpSession object associated with the request. Recall that sessions are created automatically, so this variable is bound even if there is no incoming session reference.

5. **application**

This variable is the ServletContext as obtained via getServletConfig().getContext().

6. **config**

This variable is the ServletConfig object for this page.

7. **pageContext**

JSP introduced a new class called PageContext to give a single point of access to many of the page attributes and to provide a convenient place to store shared data.

8. **page**

This variable is simply a synonym for this

JSP directives

JSP directive affects the overall structure of the servlet that results from the JSP page. In JSP, there are three types of directives: page, include, and taglib. The page directive lets you control the structure of the servlet by importing classes, customizing the servlet superclass, setting the content type, and the like.

Jsp Page Directive

The page directive lets you define one or more of the following case-sensitive attributes: import, contentType, isThreadSafe, session, buffer, autoflush, extends, info, errorPage, isErrorPage, and language. These attributes are explained below,

1. **The import Attribute**

The import attribute of the page directive lets you specify the packages that should be imported by the servlet into which the JSP page gets translated.

<%@ page import="java.util.*" %>

2. **The contentType Attribute**

The contentType attribute sets the Content-Type response header, indicating the MIME type of the document being sent to the client.

<%@ page contentType="text/plain" %>

3. **The isThreadSafe Attribute**

The isThreadSafe attribute controls whether or not the servlet that results from the JSP page will implement the SingleThreadModel interface. The default value is true, which means that the system assumes you made your code thread safe.

<%@ page isThreadSafe="true" %>

4. **The session Attribute**

The session attribute controls whether or not the page participates in HTTP sessions. Use of this attribute takes one of the following two forms:
<%@ page session="true" %>
A value of true (the default) indicates that the predefined variable session (of type HttpSession) should be bound to the existing session if one exists; otherwise, a new session should be created and bound to the session.

5. **The buffer Attribute**

The buffer attribute specifies the size of the buffer used by the out variable, which is of type JspWriter (a subclass of PrintWriter).
e.g.<%@ page buffer="sizekb" %>

6. **The autoflush Attribute**

The autoflush attribute controls whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows e.g.<%@ page autoflush="true" %>

7. **The extends Attribute**

The extends attribute indicates the superclass of the servlet that will be generated for the JSP page e.g.
<%@ page extends="package.class" %>

8. **The info Attribute**

The info attribute defines a string that can be retrieved from the servlet by means of the getServletInfo method.
e.g.<%@ page info="Some Message" %>

9. **The errorPage Attribute**

The errorPage attribute specifies a JSP page that should process any exceptions (i.e., something of type Throwable) thrown but not caught in the current page. It is used as follows:
<%@ page errorPage="Relative URL" %>
The exception thrown will be automatically available to the designated error page by means of the exception variable.

10. **The isErrorPage Attribute**

The isErrorPage attribute indicates whether or not the current page can act as the error page for another JSP page.
e.g.<%@ page isErrorPage="true" %>

JSP Taglib Directive

- JSP taglib directive is used to define the tag library with "taglib" as the prefix, which we can use in JSP.
- More detail will be covered in JSP Custom Tags section

- JSP taglib directive is used in the JSP pages using the JSP standard tag libraries
- It uses a set of custom tags, identifies the location of the library and provides means of identifying custom tags in JSP page.

Syntax of taglib directive:

<%@ taglib uri="uri" prefix="value"%>
Here "uri" attribute is a unique identifier in tag library descriptor and "prefix" attribute is a tag name.

Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
  <%@ taglib prefix="gurutag" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Directive JSP</title>
<gurutag:hello/>
</head>
<body>
</body>
</html>
```

JSP Include directive

- JSP "include directive"(codeline 8) is used to include one file to the another file
- This included file can be HTML, JSP, text files, etc.
- It is also useful in creating templates with the user views and break the pages into header footer and sidebar actions.
- It includes file during translation phase

Syntax of include directive:

<%@ include....%>

Example:

Directive_jsp2.jsp (Main file)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
  <%@ include file="directive_header_jsp3.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```



```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Directive JSP2</title>
</head>
<body>
<a>This is the main file</a>
</body>
</html>
```

Directive_header_jsp3.jsp (which is included in the main file)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

</head>
<body>
<a>Header file : </a>
<%int count =1; count++;
out.println(count);%> :
</body>
</html>
```

Using JavaBeans with JSP

JavaBean

JavaBeans are reusable software components for Java that can be manipulated visually in a builder tool. Practically, they are classes written in the Java programming language conforming to a particular convention. They are used to encapsulate many objects into a single object (the bean), so that they can be passed around as a single bean object instead of as multiple individual objects. A JavaBean is Defined with following standards,

- A bean class must have a zero-argument (empty) constructor.
- A bean class should have no public instance variables (fields).
- Persistent values should be accessed through methods called getXxx and setXxx.

Basic Bean Use

The jsp:useBean action lets you load a bean to be used in the JSP page.Beans provide a very useful capability because they let you exploit the reusability of Java classes.
The simplest syntax using bean is:
<jsp:useBean id="name" class="package.Class" />

This usually means "instantiate an object of the class specified by Class, and bind it to a variable with the name specified by id." Even you can specify a scope attribute that makes the bean associated with more than just the current page.

Accessing Bean Properties

Once you have a bean, you can access its properties with jsp:getProperty, which takes a name attribute that should match the id given in jsp:useBean and a property attribute that names the property of interest.

e.g. <jsp:useBean id="book1" class="com.MyBook" />
<jsp:getProperty name="book1" property="title" />
<%= book1.getTitle() %>

Setting Bean Properties

To modify bean properties, you normally use jsp:setProperty.

- *To set individual specific property*

<jsp:setProperty name="book1"
property="title"
value='<%= request.getParameter("title") %>' />

- *To set all properties or Associating All Properties with Input Parameters* <jsp:setProperty name="book1" property="*" />

Sharing Beans

Although the beans are indeed bound to local variables, that is not the only behavior. They are also stored in one of four different locations, depending on the value of the optional scope attribute of jsp:useBean.

The scope attribute has the following possible values:

- **Page**

This is the default value. It indicates that, in addition to being bound to a local variable, the bean object should be placed in the PageContext object for the duration of the current request.Beans created with page scope are almost always accessed by jsp:getProperty, jsp:setProperty, scriptlets, or expressions later in the same page.

- **Application**

This very useful value means that, in addition to being bound to a local variable, the bean will be stored in the shared ServletContext available through the predefined application variable or by a call to getServletContext().Values in the ServletContext can be retrieved by the getAttribute method.

- **session**

This value means that, in addition to being bound to a local variable, the bean will be stored in the HttpSession object associated with the current request, where it can be retrieved with getValue.

- **request**

This value signifies that, in addition to being bound to a local variable, the bean object should be placed in theServletRequest object for the duration of the current request.

JSTL tutorial examples

JSTL stands for JSP Standard Tag Library and JSTL represents a set of predefined tags.The main purpose of JSTL is to simplify the JSP development. JSTL is not a language like JSP, it's a tag library by which we can write java programs in JSP, like for loop , if else statement etc.

Advantages of JSTL

1. Fast development.
2. Easy understandable for Humans.
3. No need to use the scriptlet tag in the JSP page.

How to use JSTL in JSP?

To use JSTL in our JSP pages, we need to download the JSTL jars and we need to add jar into WEBINF/lib directory. JSTL has some JSTL Core Tags and JSTL Functions. Before writing any JSTL Core tag or JSTL function into JSP page we need to add tag library into JSP page Header section.

JSTL Core Tags library
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
JSTL Functions Tag library
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

JSP Actions

JSP actions use the construct in XML syntax to control the behavior of the servlet engine. We can dynamically insert a file, reuse the beans components, forward user to another page, etc. through JSP Actions like include and forward. Unlike directives, actions are re-evaluated each time the page is accessed.

Syntax:

<jsp:action_name attribute="value" />

There are 12 types of Standard Action Tags in JSP. Here is the list of Standard Action tags in JSP:

- jsp:useBean
- jsp:include
- jsp:setProperty
- jsp:getProperty
- jsp:forward
- jsp:plugin
- jsp:attribute
- jsp:body
- jsp:text
- jsp:param
- jsp:attribute
- jsp:output

jsp:useBean

- This action name is used when we want to use beans in the JSP page.
- With this tag, we can easily invoke a bean.

Syntax of jsp: UseBean:

<jsp:useBean id="" class="" />

Here it specifies the identifier for this bean and class is full path of the bean class

Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Action JSP1</title>
</head>
<body>
<jsp:useBean id="name" class="demotest.DemoClass">
</body>
</html>
```

jsp:include

- It also used to insert a jsp file into another file, just like including Directives.
- It is added during request processing phase

Syntax of jsp:include

<jsp:include page="page URL" flush="true/false">

Example:

Action_jsp2 (Code Line 10) we are including a date.jsp file

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```



```
<title>Date Guru JSP</title>
</head>
<body>
<jsp:include page="date.jsp" flush="true" />
</body>
</html>
```

Date.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<p>
Today's date: <%= {new java.util.Date()).toLocaleString()%>
</p>
</body>
</html>
```

jsp:setProperty

- This property of standard actions in JSP is used to set the property of the bean.
- We need to define a bean before setting the property

Syntax:

<jsp:setproperty name="" property="" >

Here, the name defines the bean whose property is set and property which we want to set.

Also, we can set value and param attribute.

Here value is not mandatory, and it defines the value which is assigned to the property.

Here param is the name of the request parameter using which value can be fetched.

The example of setproperty will be demonstrated below with getproperty

jsp:getProperty

- This property is used to get the property of the bean.
- It converts into a string and finally inserts into the output.

Syntax:

```
<jsp:getAttribute name="" property="" >
```

Here, the name of the bean from which the property has to be retrieved and bean should be defined. The property attribute is the name of the bean property to be retrieved.

Example of setProperty and getProperty:

TestBean.java:

```
package demotest;
import java.iO.Serializable;
public class TestBean implements Serializable{
 private String msg = "null";
 public String getMsg() {
        return msg;
 }
 public void setMsg(String msg) {
        this.msg = msg;
 }
}
```

Action_jsp3.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Action 3</title>
</head>
<body>
<jsp:useBean id="GuruTest" class="demotest.TestBean" />
<jsp:setProperty name="GuruTest" property="msg" value="GuruTutorial" />
<jsp:getProperty name="GuruTest" property="msg" />
</body>
</html>
```

jsp:forward

It is used to forward the request to another jsp or any static page.

Here the request can be forwarded with no parameters or with parameters.

Syntax:

IACSD Advance Java

```
<jsp:forward page="value">
```

Here value represents where the request has to be forwarded.

Example:

Action_jsp41.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Action JSP1</title>
</head>
<body>
<jsp:forward page="jsp_action_42.jsp" />
</body>
</html>
```

```
Jsp_action_42.jsp
Guru Action JSP2</title>
</head>
<body>
<a>This is after forward page</a>
</body>
</html>
```

jsp:plugin

- It is used to introduce Java components into jsp, i.e., the java components can be either an applet or bean.
- It detects the browser and adds <object> or <embed> JSP tags into the file

Syntax:

```
<jsp:plugin type="applet/bean" code="objectcode" codebase="objectcodebase">
```

- Here the type specifies either an object or a bean
- Code specifies class name of applet or bean
- Code base contains the base URL that contains files of classes

jsp:param

- This is child object of the plugin object described above

- It must contain one or more actions to provide additional parameters.

Syntax:

```
<jsp:params>
<jsp:param name="val" value="val"/ >
</jsp:params>
```

Example of plugin and param

Action_jsp5.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Action guru jsp5</title>
</head>
<body>
<jsp:plugin type="bean" code="Student.class" codebase="demotest.Student">
 <jsp:params>
   <jsp:param name="id" value="5" />
   <jsp:param name="name" value="guru" />
 </jsp:params>
</jsp:plugin>
</body>
</html>
```

Student.java

```
package demotest;

import java.io.Serializable;

public class Student implements Serializable {

 public String getName () {
        return name;
 }
 public void setName (String name) {
        this.name = name;
 }
 public int getId() {
        return id;
 }
 public void setId (int id) {
        this.id = id;
 }
 private String name = "null";
 private int id = 0;

}
```

jsp:body

- This tag is used to define the XML dynamically i.e., the Elements can generate during request time than compilation time.
- It actually defines the XML, which is generate dynamically element body.

Syntax:

```
<jsp:body></jsp:body>
```

jsp:attribute

- This tag is used to define the XML dynamically i.e. the elements can be generated during request time than compilation time
- It actually defines the attribute of XML which will be generated dynamically.

Syntax:

```
<jsp:attribute></jsp:attribute>
```

Example of body and attribute:

Action_jsp6.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Action Guru JSP6</title>
</head>
<body>
<jsp:element name="GuruXMLElement">
<jsp:attribute name="GuruXMLattribute">
```

```
Value
</jsp:attribute>
<jsp:body>Guru XML</jsp:body>
</jsp:element>
</body>
</html>
```

jsp:text

- It is used to template text in JSP pages.
- Its body does not contain any other elements, and it contains only text and EL expressions.

Syntax:

```
<jsp:text>template text</jsp:text>
```

Here template text refers to only template text (which can be any generic text which needs to be printed on jsp) or any EL expression.

Example:

Action_jsp7.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Action JSP7</title>
</head>
<body>
<jsp:text>Guru Template Text</jsp:text>
</body>
</html>
```

jsp:output

- It specifies the XML declaration or the DOCTYPE declaration of jsp
- The XML declaration and DOCTYPE are declared by the output

Syntax:

```
<jsp:output doctype-root-element="" doctype-system="">
```


Here, doctype-root-element indicates the root element of XML document in DOCTYPE.

Doctype-system indicates doctype which is generated in output and gives system literal

Example:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Action Guru JSP8</title>
</head>
<body>
<jsp:output doctype-root-element="html PUBLIC"
doctype-system="http://www.w3.org/TR/html4/loose.dtd"/>
</body>
</html>
```

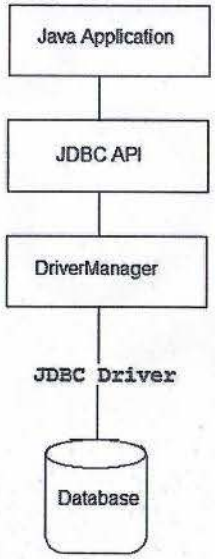
JDBC & Transaction Management

Introduction to JDBC API

Java Database Connectivity(JDBC) is an Application Programming Interface(API) used to connect Java application with Database. and execute the query with the database. It is a part of JavaSE (Java Standard Edition).

JDBC is used to interact with various types of Database such as Oracle, MS Access, My SQL and SQL Server. JDBC can also be defined as the platform-independent interface between a relational database and Java programming. It allows java program to execute SQL statement and retrieve result from database.

The JDBC API consists of classes and methods that are used to perform various operations like: connect, read, write and store data in the database.



Java introduced JDBC 4.0, a new version which is an advanced specification of JDBC. It provides the following advance features

- Connection Management
- Auto loading of Driver Interface.
- Better exception handling

- Support for large object
- Annotation in SQL query.

JDBC 4.1

The JDBC 4.1 version was introduced with Java SE 7 and includes the following features:

- Allows to use a try-with-resources statement to automatically close resources of type Connection, ResultSet etc.
- Introduced the RowSetFactory interface and the RowSetProvider class to create all types of row sets supported by your JDBC driver.

JDBC 4.2

version is introduced with Java SE 8 and includes the following features:

- The REF_CURSOR support.
- Added an interface java.sql.DriverAction
- Added an interface java.sql.SQLType
- Added an Enum java.sql.JDBCType
- Add Support for large update counts
- Improved the existing interfaces: Driver, DriverManage, DatabaseMetaData.
- Interfaces and classes enhanced for RowSet1.2:

JDBC 4.3

- Added support to Statement for enquoting literals and simple identifiers
- Added Sharding support
- Enhanced Connection to be able to provide hints to the driver that a request, an independent unit of work, is beginning or ending
- Enhanced DatabaseMetaData to determine if Sharding is supported
- Added the method drivers to DriverManager to return a Stream of the currently loaded and available JDBC drivers

JDBC Architecture -

JDBC Drivers

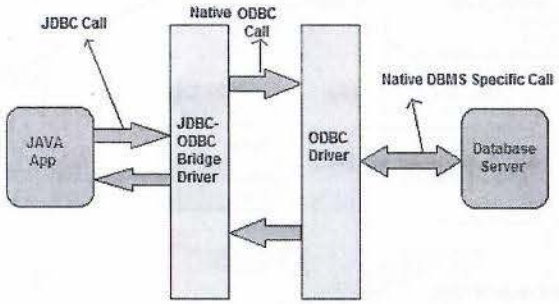
JDBC Driver is required to establish connection between application and database. It also helps to process SQL requests and generating result. The following are the different types of driver available in JDBC which are used by the application based on the scenario and type of application.

- Type-1 Driver or JDBC-ODBC bridge
- Type-2 Driver or Native API Partly Java Driver
- Type-3 Driver or Network Protocol Driver
- Type-4 Driver or Thin Driver

1. JDBC-ODBC bridge

Type-1 Driver act as a bridge between JDBC and other database connectivity mechanism(ODBC). This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver.

Note: In Java 8, the JDBC-ODBC Bridge has been removed.



Advantage

- Easy to use
- Allow easy connectivity to all database supported by the ODBC Driver.

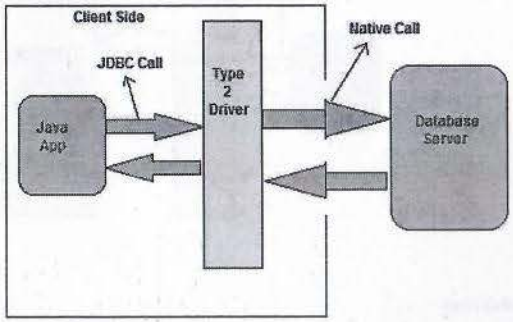
Disadvantage

- Slow execution time
- Dependent on ODBC Driver.
- Uses Java Native Interface(JNI) to make ODBC call.

2. Native API Driver

IACSD Advance Java

This type of driver make use of Java Native Interface(JNI) call on database specific native client API. These native client API are usually written in C and C++.



Advantage

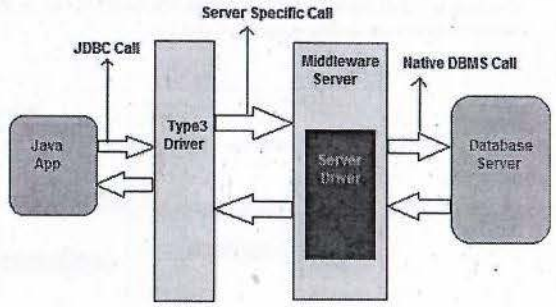
- faster as compared to Type-1 Driver
- Contains additional features.

Disadvantage

- Requires native library
- Increased cost of Application

3. Network Protocol Driver

This driver translate the JDBC calls into a database server independent and Middleware server-specific calls. Middleware server further translate JDBC calls into database specific calls.



Advantage

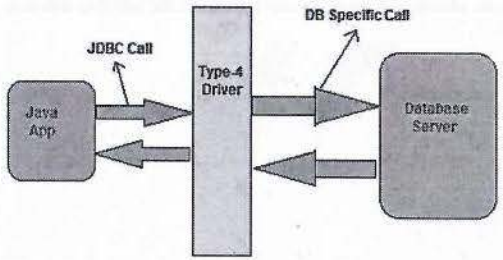
- Does not require any native library to be installed.
- Database Independency.
- Provide facility to switch over from one database to another database.

Disadvantage

- Slow due to increase number of network call.

4. Thin Driver

This is Driver called Pure Java Driver because. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

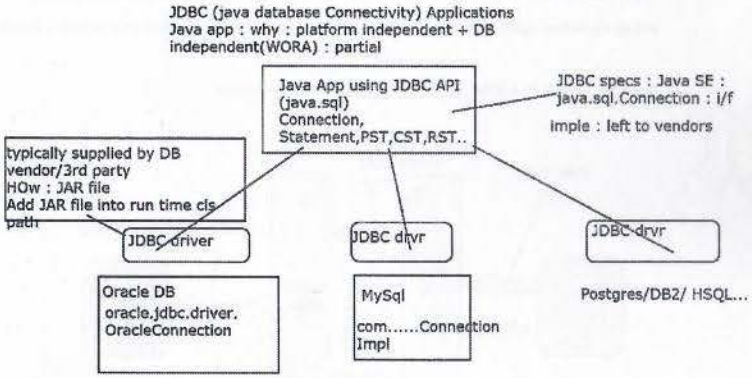


Advantage

- Does not require any native library.
- Does not require any Middleware server.
- Better Performance than other driver.

Disadvantage

- Slow due to an increased number of network call.



Generalized steps in JDBC

1. Add JDBC jar in runtime class path
2. get fixed db conn. (D.M.getConn(url,name,pwd))
3. create statement
Statement <---- PST <---- CST
4. exec methods
execQuery , execUpdate,exec
5. close all db resources.

JDBC API is mainly divided into two packages. When we are using JDBC, we have to import these packages to use classes and interfaces in our application.

1. java.sql
2. javax.sql

java.sql package

This package includes classes and interfaces to perform almost all JDBC operations such as creating and executing SQL Queries.

Important classes and interface of java.sql package

classes/interface	Description
java.sql.BLOB	Provide support for BLOB(Binary Large Object) SQL type.
java.sql.Connection	creates a connection with specific database
java.sql.CallableStatement	Execute stored procedures
java.sql.CLOB	Provide support for CLOB(Character Large Object) SQL type.
java.sql.Date	Provide support for Date SQL type.
java.sql.Driver	create an instance of a driver with the DriverManager.
java.sql.DriverManager	This class manages database drivers.
java.sql.PreparedStatement	Used to create and execute parameterized query.
java.sql.ResultSet	It is an interface that provide methods to access the result row-by-row.
java.sql.SQLException	Encapsulate all JDBC related exception.
java.sql.Statement	This interface is used to execute SQL statements.
DatabaseMetaData	Comprehensive information about the database as a whole.
DriverAction	An interface that must be implemented when a Driver wants to be notified by DriverManager.
ResultSetMetaData	An object that can be used to get information about the types and properties of the columns in a ResultSet object.
Statement	The object used for executing a static SQL statement and returning the results it pro duces.
Wrapper	Interface for JDBC classes which provide the ability to retrieve the delegate instance when the instance in question is in fact a proxy class.

IACSD Advance Java

The javax.sql package

This package is also known as JDBC extension API. It provides classes and interface to access server-side data.

Important classes and interface of javax.sql package

classes/interface	Description
javax.sql.DataSource	Represent the DataSource interface used in an application.
javax.sql.PooledConnection	provide objects to manage connection pools.
RowSet	The interface that adds support to the JDBC API for the JavaBeans™ component model.

JDBC Classes& Interfaces:

A list of popular *interfaces* of JDBC API is given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

Classes of JDBC API

A list of popular *classes* of JDBC API is given below:

- DriverManager class
- Blob class
- Clob class
- Types class

DriverManager class

In Java, the DriverManager class it an interface between the User and the Driver. This class is used to have a watch on the driver which is been used for establishing the connection between a database and a driver. The DriverManager class have a list of Driver class which are registered and are called as DriverManager.registerDriver().

Connection interface

In Java, The Connection interface is used for creating the session between the application and the database. This interface contains Statement, PreparedStatement and DatabaseMetaData. The connection objects are used in Statement and the DatabaseMetaData. commit(), rollback() etc.. are some of the methods of Connection Interface.

Sr. No.	Method	Description
1	public Statement createStatement()	It is used for creating an object of statement for executing the SQL queries.
2	public Statement createStatement(intresultSetType,intresultSetConcurrency)	It is used for creating objects for the ResultSet from the given type and concurrency.
3	public void setAutoCommit(boolean status)	It is used for setting the commit status. By default, it is always true.
4	public void commit()	It is used to save the changes which have been commit or rollback permanent
5	public void rollback()	It is used to delete the changes which have been commit or rollback permanent
6	public void close()	It is used to delete the changes which have been commit or rollback permanent

Statement interface

In Java, The Statement interface is used for executing queries using the database. This interface is a factory of ResultSet. It is used to get the Object of ResultSet. Methods of this interface is given below.

S.No.	Method	Description
1	public ResultSetexecuteQuery(String sql)	It is used for executing the SELECT query
2	public intexecuteUpdate(String sql)	It is used for executing any specified query
3	public boolean execute(String sql)	It is used when multiple results are required.
4	public int[] executeBatch()	It is used for executing the batch of commands.

ResultSet interface

In Java, the ResultSet Interface is used for maintaining the pointer to a row of a table. In starting the pointer is before the first row. The object can be moved forward as well as backward direction using TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int). Methods of this interface is given below.

Sr. No.	Method	Description
1	public boolean next()	It is used for moving the cursor to the next position from the current position.
2	public boolean previous()	It is used for moving the cursor to the previous position from the current position.
3	public boolean first()	It is used for moving the cursor to the first position from the current position.
4	public booleanlast()	It is used for moving the cursor to the Last position from the current position.
5	public booleanabsolute(int row)	It is used for moving the cursor to the specified position from the current position.
6	public booleanrelative(int row)	It is used for moving the cursor to the relative row number from the current position.
7	public intgetInt(intcolumnIndex)	It is used to get the data from the specified position.
8	public intgetInt(String columnName)	It is used to get the data from the specified column name of the current row.
9	public StringgetString(intcolumnIndex)	It is used to get the data from the specified column name of the current row in form of an integer.

10	public StringgetString(StringcolumnIndex)	It is used to get the data from the specified column name of the current row in form of string.

PreparedStatement interface

In Java, The PreparedStatement interface is a subinterface of Statement. It is mainly used for the parameterized queries.

A question mark (?) is passed for the values.

The values to this question marks will be set by the PreparedStatement. Methods of this interface is given below.

Sr.No.	Method	Description
1	public void setInt(intparamIndex, int value)	It is used for setting the integer value for the given parameter index.
2	public void setString(intparamIndex, String value)	It is used for setting the String value for the given parameter index.
3	public void setFloat(intparamIndex, float value)	It is used for setting the Float value for the given parameter index.
4	public void setDouble(intparamIndex, double value)	It is used for setting the Double value for the given parameter index.
5	public intexecuteUpdate()	It is used for executing a query.
6	public ResultSetexecuteQuery()	It is used for executing the select query.

CallableStatement Interface

CallableStatement interface is used to call the **stored procedures and functions.**

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

public CallableStatement prepareCall("{ call procedurename(?,?...?)}");

The example to get the instance of CallableStatement is given below:

CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");

It calls the procedure myprocedure that receives 2 arguments.

Full example to call the stored procedure using JDBC

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

```
create or replace procedure "INSERTR"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

The table structure is given below:

1. create table user(id number(10), name varchar2(200));

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user.

Note that you need to create the user table as well to run this application.

```
import java.sql.*;
public class Proc {
public static void main(String[] args) throws Exception{
```

```
//Class.forName("oracle.jdbc.driver.OracleDriver");  //optional
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
stmt.setInt(1,1011);
stmt.setString(2,"Amit");
stmt.execute();

System.out.println("success");
}
}
```

Now check the table in the database, value is inserted in the user table.

Example to call the function using JDBC

In this example, we are calling the sum4 function that receives two input and returns the sum of the given number. Here, we have used the **registerOutParameter** method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed.

The **Types** class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Let's create a simple function in the database first.

```
create or replace function sum4
(n1 in number,n2 in number)
return number
is
temp number(8);
begin
```



```
temp :=n1+n2;
return temp;
end;
/
```

Now, let's write the simple program to call the function.

```
import java.sql.*;

public class FuncSum {
public static void main(String[] args) throws Exception{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
stmt.setInt(2,10);
stmt.setInt(3,43);
stmt.registerOutParameter(1,Types.INTEGER);
stmt.execute();
System.out.println(stmt.getInt(1));
}
}
```

Output: 53

Example - Callable Statement
Database - MYSQL

1. **create table person(id number(10), gender char);**
2. **Procedures-**

p1 →

```
delimiter //
create procedure p1( i int, g char)
```

```
 begin
 insert into person values(i,g);
 end
 //

p2 →
delimiter //
create procedure p2( i int, out g char)
 begin
 select gender into g from person where id=i;
 end
 //
```

3. Program with MYSQL

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class CollableEx {
  static final String DB_URL = "jdbc:mysql://localhost/test";
  static final String USER = "root";
  static final String PASS = "root";
  static final String QUERY1 = "{call p1 (?, ?)}"; //Insert Query
  static final String QUERY2 = "{call p2 (?, ?)}"; //Select Query with out parameter

  public static void main(String[] args) {
    // Open a connection
    try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
      CallableStatement stmt1 = conn.prepareCall(QUERY1);
                CallableStatement stmt2 = conn.prepareCall(QUERY2);
    ){
      stmt1.setInt(1, 102);
      stmt2.setInt(1, 102);  // This would set ID

      // Because second parameter is OUT so register it
      stmt2.registerOutParameter(2, java.sql.Types.CHAR);
      stmt1.setString(2, 'M'+"");

      //Use execute method to run stored procedure.
      System.out.println("Executing stored procedure..." );
      stmt1.execute(); //check in DB record inserted

      stmt2.execute();

      //Retrieve employee name with getXXX method
      String empName = stmt2.getString(2);
```



```
      System.out.println("Emp Name with ID: 102 is " + empName);
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

ResultSetMetaData Interface

The ResultSetMetaData interface is used to get metadata from the ResultSet object. Metadata are the data about data. Methods of this interface is given below.

Sr. No.	Method	Description
1	public intgetColumnCount()throws SQLException	It is used to get the total number of columns.
2	public String getColumnName(int index)throws SQLException	It is used to get the name of the column of a specified column index.
3	public StringgetColumnTypeName(int index)throws SQLException	It is used to get the name of the column of a specified index.
4	public StringgetTableName(int index)throws SQLException	It is used to get the name of a table from the specified column index

Steps to connect a Java Application to Database

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

1. **Register the Driver**
2. **Create a Connection**
3. **Create SQL Statement**
4. **Execute SQL Statement**
5. **Closing the connection**

Register the Driver

It is first an essential part to create JDBC connection. JDBC API provides a method Class.forName() which is used to load the driver class explicitly. For example, if we want to load a jdbc-odbc driver then the we call it like following.

Example to register with JDBC-ODBC Driver till java 1.5

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Create a Connection

After registering and loading the driver in step1, now we will create a connection using getConnection() method of DriverManager class. This method has several overloaded methods that can be used based on the requirement. Basically it require the database name, username and password to establish connection. Syntax of this method is given below.

Syntax

getConnection(String url)

getConnection(String url, String username, String password)

getConnection(String url, Properties info)

This is a sample example to establish connection with Oracle Driver

```
Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","username","password");
```

```
import java.sql.*;
class Test {
 public static void main(String[] args) {
  try {
   //Loading driver
   Class.forName("oracle.jdbc.driver.OracleDriver");

   //creating connection
    Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "username",
"password");

   Statement s = con.createStatement(); //creating statement

   ResultSet rs = s.executeQuery("select * from Student"); //executing statement

   while (rs.next()) {
    System.out.println(rs.getInt(1) + " " + rs.getString(2));
   }

   con.close(); //closing connection
  } catch (Exception e) {
   e.printStacktrace();
  }
 }
}
```

Create SQL Statement

In this step we will create statement object using createStatement() method. It is used to execute the sql queries and defined in Connection class. Syntax of the method is given below.

Syntax

public Statement createStatement() throws SQLException

Example to create a SQL statement

Statement s=con.createStatement();

Execute SQL Statement

After creating statement, now execute using executeQuery() method of Statement interface. This method is used to execute SQL statements. Syntax of the method is given below.

Syntax

public ResultSet executeQuery(String query) throws SQLException

Example to execute a SQL statement

In this example, we are executing a sql query to select all the records from the user table and stored into resultset that further is used to display the records.

```
ResultSet rs=s.executeQuery("select * from user");
 while(rs.next())
 {
  System.out.println(rs.getString(1)+" "+rs.getString(2));
}
```

Closing the connection

This is final step which includes closing all the connection that we opened in our previous steps. After executing SQL statement you need to close the connection and release the session. The close() method of Connection interface is used to close the connection.

Syntax

public void close() throws SQLException

Example of closing a connection

con.close();

Now lets combine all these steps into a single example and create a complete example of JDBC connectivity.

Example: All Steps into one place - Oracle DB

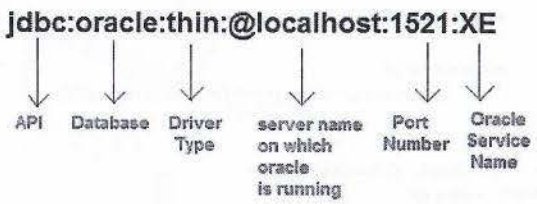
```
import java.sql.*;
class Test {
 public static void main(String[] args) {
  try {
   //Loading driver
   Class.forName("oracle.jdbc.driver.OracleDriver"); //optional

//creating connection Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "username", "password");
   Statement s = con.createStatement(); //creating statement
   ResultSet rs = s.executeQuery("select * from Student"); //executing statement
```

```
while (rs.next()) {
 System.out.println(rs.getInt(1) + " " + rs.getString(2));
}
```

Java: Connecting to Oracle Database using Thin Driver

To connect a Java application with Oracle database using Thin Driver. You need to follow the following steps

Load Driver Class: The Driver Class for oracle database is oracle.jdbc.driver.OracleDriver and Class.forName("oracle.jdbc.driver.OracleDriver") method is used to load the driver class for Oracle database.
Create Connection: For creating a connection you will need a Connection URL. The Connection URL for Oracle is



You will also require Username and Password of your Oracle Database Server for creating connection.
Loading jar file: To connect your java application with Oracle, you will also need to load ojdbc14.jar file. This file can be loaded into 2 ways.

Copy the jar file into C:\Program Files\Java\jre7\lib\ext folder.
or,
Set it into classpath.
NOTE: Oracle 10g as database. For other version of Oracle you will be require to do some small changes in the Connection URL.

programs for performing simple operations like create, insert and select on database tables.

Create a table in Oracle Database
Use the below SQL command to create a table Student into oracle database. This table has two columns: sid and sname.

```
create table Student(sid number(10),sname varchar2(20));
```
Insert some record into the table
```
insert into Student values(101,'adam');
```



```
insert into Student values(102,'abhi');
```

JDBC Example : Accessing record

```
import java.sql.*;
class Test
{
 public static void main(String []args)
 {
        try{
                //Loading driver
                Class.forName("oracle.jdbc.driver.OracleDriver");

                //creating connection
                Connection con = DriverManager.getConnection
                                ("jdbc:oracle:thin:@localhost:1521:XE","username","password");

                Statement s=con.createStatement();     //creating statement

                ResultSet rs=s.executeQuery("select * from Student");  //executing statement

                while(rs.next()){
                        System.out.println(rs.getInt(1)+" "+rs.getString(2));
                }

                con.close();  //closing connection
        }catch(Exception e){
                e.printStackTrace();
        }
 }
}
```

```
101 adam
102 abhi
```

Example: Inserting record

```
import java.sql.*;
class Test
{
 public static void main(String []args)
 {
        try{
                //Loading driver...
                Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
            //creating connection...
            Connection con = DriverManager.getConnection
                        ("jdbc:oracle:thin:@localhost:1521:XE","username","password");

            PreparedStatement pst=con.prepareStatement("insert into Student values(?,?)");

            pst.setInt(1,104);
            pst.setString(2,"Alex");
            pst.executeUpdate();
        }catch(Exception e){
            e.printStackTace()
        }
 }
}
```

Connecting to MySQL Database using Thin Driver

To connect a Java application with MySQL database using Thin Driver. You need to follow the following steps

Load Driver Class: The Driver Class for MySQL database is com.mysql.jdbc.Driver and Class.forName("com.mysql.jdbc.Driver") method is used to load the driver class for MySQL database.
Create Connection: For creating a connection you will need a Connection URL. The Connection URL for MySQL is
connecting to mysql database

You will also require Username and Password of your MySQL Database Server for creating connection.
Loading jar file: To connect your java application with MySQL, you will also need to load mysql-connector.jar file. This file can be loaded into 2 ways.
Copy the jar file into C:\Program Files\Java\jre7\lib\ext folder.
or,

Set it into classpath. For more detail see how to set classpath

Program -
Create a table in MySQL Database
Use the below SQL command to create a table Student into mysql database. This table has two columns: sid and name.

```
create table Student(sid int(10),name varchar(20));
Insert some record into the table
insert into Student values(102,'adam');
insert into Student values(103,'abhi');
```


Example: Accessing record using JDBC Java application.

```
import java.sql.*;
class Test
{
 public static void main(String []args)
 {
        try{
            //Loading driver
            Class.forName("com.mysql.jdbc.Driver");

            //creating connection
            Connection con = DriverManager.getConnection
                        ("jdbc:mysql:/ /localhost:3306/test","username","password");

            Statement s = con.createStatement();   //creating statement

            ResultSet rs = s.executeQuery("select * from Student"); //executing statement

            while(rs.next()){
                System.out.println(rs.getInt(1)+" "+rs.getString(2));
            }

            con.close();   //closing connection
        }catch(Exception e){
            e.printStackTrace();
        }
 }
}
102 adam
103 abhi
```

Example: Inserting Record
We can insert data into table using Java application. using Prepared statement and parameterized query to execute SQL query.
The executeUpdate() method is used to execute the insert query.

```
import java.sql.*;
class Test
{
 public static void main(String []args)
 {
        try{
            //Loading driver
            Class.forName("com.mysql.cj.jdbc.Driver");
```

```
            //creating connection
            String dbURL =
"jdbc:mysql://localhost:3306/advjava?createDatabaseIfNotExist=true&useSSL=false&allowPublicKeyRetr
ieval=true";
        cn = DriverManager.getConnection(dbURL, "root", "root");

            PreparedStatement pst=con.prepareStatement("insert into Student values(?,?)");

            pst.setInt(1,104);
            pst.setString(2,"Alex");
            pst.executeUpdate();

}catch (Exception e) {
            e.printStackTrace();
        }
}
```

JDBC MongoDB Connectivity

In this tutorial, we are going to discuss database connectivity between Java application and MongoDB. MongoDB is a document based database that is used to store data into JSON format rather than table or relation. It is completely different from the relational database like: mysql. You can get idea of MondoDB from our latest tutorial here MongoDB Tutorial.

Now lets start to learn the connectivity process. Before Starting make sure you have mongoDB installed in your computer system. These are some prerequisites that should be followed to get connected.

Prerequisites

A running MongoDB on localhost using the default port for MongoDB 27017. See Installation Mongodb installation steps.

MongoDB Driver.

After installing MongoDB, it is important to have mongodb driver as well. So to get these, you can either visit the official site MongoDB Java Driver or download the Jar file from our site. Keep this JAR file into your java application directory.

Note: We don't need to use JDBC API for mongodb connectivity, It's API is completely independent and does not require any other external library.

Following are the essential connectivity steps for Java application to mongodb database.

Create MongoDB Instance

You can instantiate a MongoClient object without any parameters to connect to a MongoDB instance running on localhost on port:27017. Use the following code.

```
MongoClient mongoClient = MongoClients.create();
```

You can specify the mongodb host and running port, in case mongodb is running on different port.

```
MongoClient mongoClient = MongoClients.create("mongodb://hostOne:27017");
```


Access a Database

After creating mongoDB instance, use its getDatabase() method to access a database by specifying the name of the database to the getDatabase() method. If a database does not exist, MongoDB creates the database when you first store data for that database. For example, we are connecting to test database.

```
MongoDatabase database = mongoClient.getDatabase("test");
```

Access a Collection

After creating mongoDB database, use its getCollection() method to access a collection by specifying name. If a collection does not exist, MongoDB creates the collection when you first store data for that collection.

Use the following code to create collection, for example : student.

```
MongoCollection<Document> collection = database.getCollection("student");
```

Create a Document

To create the document using the Java driver, use the Document class, and use its append() method to include additional fields and values to the document object. For example, see the following code.

```
Document doc = new Document("name", "Ramesh");
doc.append("id",12);
```

Insert a Document

After creating document, you can insert documents into the collection. We can insert single and multiple documents too. To insert a single document into the collection, you can use the collection's insertOne() method.

```
 collection.insertOne(doc);
```

To add multiple documents, you can use the collection's insertMany() method which takes a list of documents to insert.

Fetch A Document

You can use the collection's find() method to fetch a record from the document. The find() method returns an iterable so you loop through to get individual data.

Now Lets put all these steps into one Java application and connect to the MongoDB database.

Java MongoDB Example:

Here we are connecting to mongodb by specifying the host and port number and then create a database and a collection to store the data. The insertOne() method is used to insert a single record to the document. See the below example.

```
import com.mongodb.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
public class Demo {
 public static void main(String[] args){
        try{
                // Connecting To MongoDB
```

```
            MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
            // Creating DataBase
            MongoDatabase db = mongoClient.getDatabase("test");
            // Creating Collection/Table
            MongoCollection<Document> table = db.getCollection("student");
            // Creating Document/Record
            Document doc = new Document("name", "Ramesh");
            doc.append("id",12);
            // Inserting Data
            table.insertOne(doc);
        }catch(Exception e){
            System.out.println(e);
        }
 }
}
```

Accessing Data from MongoDB

After creating database and inserting data, now lets fetch that record. We are using find() method to get records from the document and then iterating the record, and displaying by using their names. See the below example.

```
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
public class Demo {
 public static void main(String[] args){
        try{
            // Connecting To MongoDB
            MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
            // Creating DataBase
            MongoDatabase db = mongoClient.getDatabase("test");
            // Creating Collection/Table
            MongoCollection<Document> table = db.getCollection("student");
            // Accessing Data
            FindIterable<Document> data =  table.find();
            // Traversing Data
            for(Document record : data) {
                System.out.println(record.getInteger("id")+" : "+record.getString("name"));
            }
            mongoClient.close();

        }catch(Exception e){
            System.out.println(e);
```



```
        }
 }
}
```

12 : Ramesh

Access in JSON Format

MongoDB is a document based database and uses JSON like format to store the data into documents. It provides a method toJSON() to access data into JSON format that we used in this example.

```
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
public class Demo {
 public static void main(String[] args){
        try{
            // Connecting To MongoDB
            MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
            // Creating DataBase
            MongoDatabase db = mongoClient.getDatabase("test");
            // Creating Collection/Table
            MongoCollection<Document> table = db.getCollection("student");
            // Accessing Data
            FindIterable<Document> data =  table.find();
            // Traversing Data
            for(Document record : data) {
                System.out.println(record.toJson());
            }
            mongoClient.close();

        }catch(Exception e){
            System.out.println(e);
        }
 }
}
```

Copy

```
{"_id": {"$oid": "5ece65f5e8aa304abbf99953"}, "name": "Ramesh", "id": 12}
```

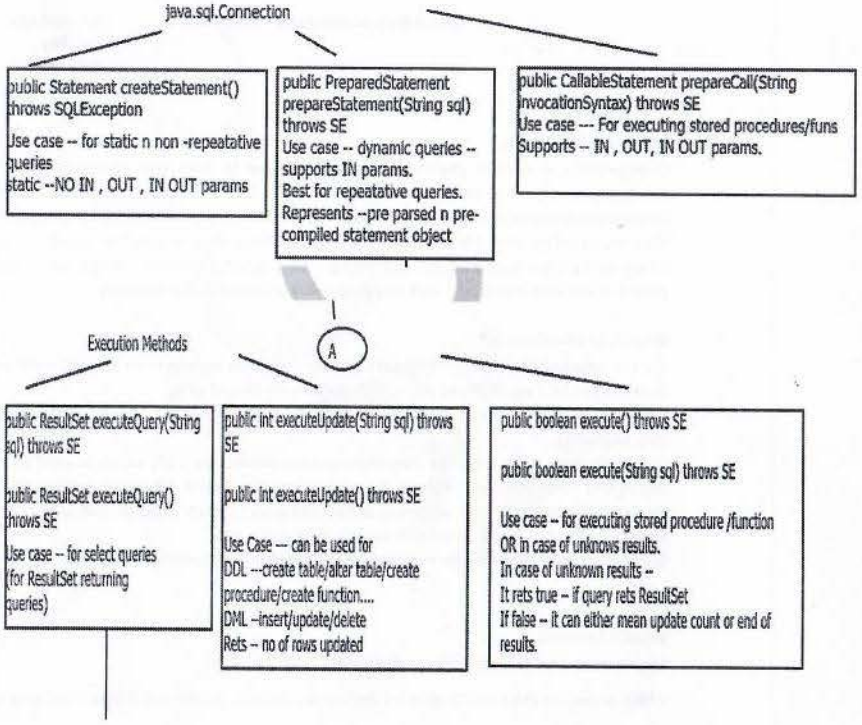
Stored procedures and functions Invocation

SQL Injection overview and prevention

JDBC Steps

Add JDBC driver in run time class path.(In IDE add ext jar)

1. Load Type IV JDBC driver in JVM's method area.(meta space)
API of java.lang.Class<T>
public static <T> T forName(String clsName) throws ClassNotFoundException
2. Get a FIXED connection from underlying DB.
API of java.sql.DriverManager class
public static Connection getConnection(String dbURL,String userName,String password)
throws SQLException
dbURL -- DB sepecific for Type IV JDBC driver
syntax : jdbc:mysql://localhost:3306/test



ResultSet processing
---java.sql.ResultSet -- i/f
Object representation of selceted rows n columns.
API of ResultSet
1. public boolean next() throws SE

2. How to read row data ?
public Type getType(int colPos) throws SE

public Type getType(String colName) throws SE

Type --- JDBC data type / generic SQL type

Native DB types	JDBC Data types
char/varchar/varchar2	--- string
number(n)	---- int/long
number(m,n)	--- float/double
date	--- java.sql.Date
time	--- java.sql.Time
timestamp	-- java.sql.TimeStamp
clob	-- java.sql.Clob
blob	-- java.sql.Blob

Before terminating -- close RST, ST/PST/CST
Close DB connection (or use try-with-resources)

A

Set IN Param/s
API of PreparedStatement
public void setType(int paramPosition,Type val) throws SE

Type -- JDBC data type
eg :

Hibernate

Hibernate is an object relational mapper (**ORM**) Framework which is used to turn our java objects into a way to persist them into the database and vice versa.

What is ORM

Object-Relational Mapping (**ORM**) is a technique which provides us facility to query and manipulate data from a database using an object-oriented programming paradigm.

ORM binds our tables or stored procedures in java classes, so that instead of writing SQL statements to interact with your database, you use methods and properties of objects.

Advantage of Hibernate Framework

1. Hibernate is non-invasive framework means it does not force use to extend or implement API classes or interfaces.
2. Simple to learn and work means no need to learn complicated SQL queries.
3. Hibernate is a bit faster than JDBC queries.
4. By default transaction support is given.
5. Database independent query
6. Light-weight framework.

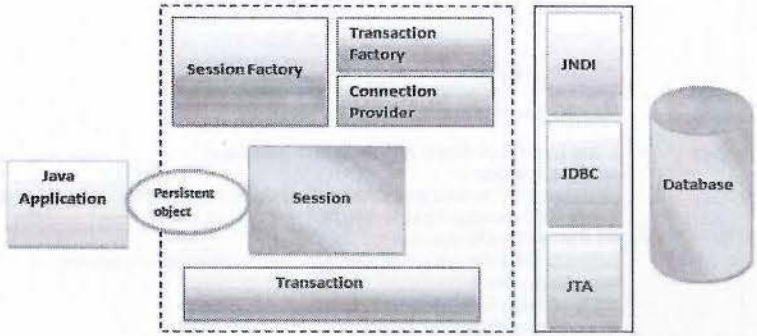
Limitation of Hibernate

1. Hibernate is slower than pure JDBC driver API.
2. Hibernate is not recommended for small project.
3. Hibernate is not suitable for Batch processing
4. So Many configurations needed for simple query also.

Hibernate Architecture

Elements of Hibernate Architecture

1. Configuration
2. SessionFactory
3. Session
4. Transaction
5. Query



Core objects of Hibernate Framework

Internal API Used by Hibernate

What is Configuration?

Configuration is a class given by hibernate people to load the hibernate XML configuration file.
Configuration cfg = new Configuration();
Configuration object is used to create the SessionFactory object. It is Object Oriented Representation of Hibernate configuration file and mapping file is nothing but Configuration object.
When we call configure() method on configuration object, hibernate configuration file(hibernate.cfg.xml placed in run time classpath) and mapping files are loaded in the memory.

What is SessionFactory?

We use sessionfactory object to create session object(via openSession or getCurrentSession) singleton(1 instance per DB / application) ,immutable,inherently thread safe.
It is a heavy weight object, therefore it has to be created only once for an application & that too at the very beginning.
SessionFactory is an interface (org.hibernate.SessionFactory : i/f) which is used to create the session object and SessionFactory contains the connection information, hibernate configuration information and mapping files,location path.Instances of SessionFactory are *thread-safe* and typically shared throughout an application.SessionFactory holds second level cache also.
SessionFactorysessionFactory = configurationObject.buildSessionFactory();

What is Session?

Session is an interface (org.hibernate.Session : i/f)

which is used to execute SQL queries like insert , update , delete and it holds first level cache.

Session object is persistence manager for the hibernate application

Session object is the abstraction of hibernate engine for the Hibernate application

Session object provides methods to perform CRUD operations

Session just represents a thin wrapper around pooled out DB connection.

Session is associated implicitly with L1 cache (having the same scope as the session lifetime) , referred to as Persistence context.

Example of CRUD

save() - Inserting the record

get() / load() - Retrieving the record

update() - Updating the record

delete() - Deleting the record

public void delete(Object ref) throws HibernateExc

ref -- either persistent or detached pojo ref.

What is Transaction?

Transaction is an interface using which we maintain transactions in hibernate applications.

What is Query?

Query is an interface which is used to execute DML statements in hibernate applications like select , delete etc.

Hibernate Mapping and Configuration file

Hibernate Mapping and Configuration file are the heart of any **ORM** based framework application, here in hibernate application mapping is possible in two ways either using XML file or using Hibernate given annotation.

What is Hibernate Mapping file?

Hibernate mapping file is used to configure java persistent class details like class name, fields name etc. In hibernate application our java class name will be treated as table name for any database and fields name will be treated as column name of table.

We can construct one or more hibernate mapping files in a hibernate application. The naming convention of the hibernate mapping file is **javaClassName.hbm.xml.**

Syntax

```
<hibernate-mapping>
<class name="java _persistent_class" table="table_name">
<!– Here id should be primary key of database –>
<id name="field_name" column="database_column_name" type="Java_data_type" />
<property name="field_name" column="database_column_name"type="Java_data_type"/>
</class>
</hibernate-mapping>
```


What is Hibernate Configuration file?

Hibernate Configuration file is used to pass java JDBC driver name, driver class name, username and password and configuration file will be loaded file **Configuration** hibernate class in hibernate application. Note that we have to pass hibernate mapping file into Hibernate Configuration file.

Syntax

```
<hibernate-configuration>
<session-factory>
<!– JDBC Connection info –>
<property name="connection_driver_class">Driver Class Name</property>
<property name="connection_url">URL </property>
<property name="connection_user">user </property>
<property name="connection_password">password</property>
<!– Hibernate properties –>
<property name="show_sql">true/false</property>
<property name="Hibernate_Dialect">Hibernate Given dialect</property>
<property name="hbm2ddl.auto">create/update/delete/select</property>
<!– Hibernate mapping file –>
<mapping resource="mappingfilename.xml" / > </session-factory>
</hibernate-configuration>
```

Steps to create hibernate application in eclipse

To use hibernate framework in java application we need to follow some steps.We can use hibernate framework in both application in standalone application and in web application also, So steps is similar for both of the application but to add jar some differences are there like.

1. For java application we need to create **lib** folder and we need to place the all the hibernate jar files and then we have to configure build path.
2. And for web application we need to add hibernate jar files inside WEB-INF/lib folder and we no need to configure build path for web application.

Steps to create hibernate application

1. Create the java project
2. Add Hibernate jar files.
3. Create Persistent class inside any package.
4. Create the hibernate mapping file for Persistent class.
5. Create the Hibernate Configuration file for database information like driver name , URL username and password.
6. Create class to retrieves or stores java persistent object.

Step 1: Create java project using below steps
New -> Java Project -> Give name of project and press next and next.

Step 2: Create lib folder in created java project.
After creating lib folder add all hibernate application related jar files and configure build path.

Step 3: Create Persistent class
Step 4: Create hibernate mapping file
Step 5: Create hibernate configuration file
Step 6: Create Test Class having main method.

Hibernate SQL Dialects List

Hibernate Dialects is used to connect with specific database and it convert hibernate query language(HQL)to database specific query language. In simple word we can say Hibernate uses "Dialects" configuration to know which database you are using so that it can switch to the database specific SQL generator code wherever/whenever necessary.

Hibernate Query Language

Hibernate Query Language **(HQL)** is a new language given by Hibernate people and it is powerful query language. **HQL** is fully object-oriented and understands notions like inheritance, polymorphism.

Why HQL came ?

As we know for every relational database will have their own query language which differs database to database so hibernate came up with new HQL language which is nothing but query language but it is not database dependent.We can say **HQL** is database independent query language.
HQL uses java class name as database table name and class fields name as table column name. e.g. To fetch roll ,name from student table.
SQL in Oracle
>select roll , name from student;
HQL
selects.roll , s.name from Student s;

Here Student is class name and roll and name is fields name of Student class. HQL query will be executed with Query interface given by hibernate.

Hibernate Criteria API

Hibernate Criteria API is used to retrieve data from database based on some criteria like greater than , less than where clause.We can use Hibernate Criteria API for data retrieval purpose only.

Advantages of Hibernate Criteria API

By the help of **Criteria API** we can impose a lot of condition on SQL queries in easy way because Hibernate has given some predefined methods like gt() for greater than etc.

Disadvantages of Criteria API

Criteria API lags a lot in performance when we have huge amount of data to retrieve.
How to use Criteria API
We can get Criteria object by calling **createCriteria()** of session interface.
Criteria criteria = session.createCriteria(Entity class);
To impose any condition like greater than or less than we have to use **Restriction** interface with **Criterion.**
Syntax
Criterion c = Restrictions.gt("roll", new Integer(101));//greater than

Hibernate Caching – First Level Cache

Hibernate supports three level of cache to optimize the performance of hibernate application.

1. First level cache
2. Second level cache
3. Query level cache

Why Hibernate support Cache ?

There are so many problems we have to face if we will not use any cache concept in our application like.

1. Performance issue we will get widely.
2. Network bandwidth issue we will.

When to use Hibernate cache

If same data is available around the application and if data will change very rarely then we can implement the cache concept in our application. But if data is changing frequently then we should not use cache.

What is Hibernate Caching – First Level Cache?

Hibernate supports First Level Cache in the form of Session object and it is configured by default we cannot disable first level cache but we can remove some of the object from Session object using some predefined method.
The scope of Hibernate First level cache is of per session object. Once session is closed cached objects are gone forever.

Hibernate second level cache

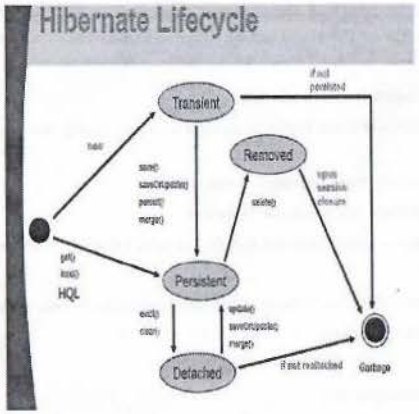
Hibernate Second Level cache will be created and associated with Hibernate SessionFactory and It will be available to all sessions.
We can use Second level cache in all sessions within the application.
Once the Hibernate SessionFactory object will be closed then all cached values will be gone.
Working of Hibernate second level cache

1. Hibernate application will try to load Entity from Session cache first if found that entity then it won't go in Second level cache.
2. If Hibernate does not find any entity in session object cache then it will search in Second level cache, if that entity found in second level cache then it will process.
3. If that entity does not find then Hibernate query will hit the database.

How to implement Second level cache in hibernate?

Hibernate does not support Second level cache by default so we need to use some vendor provided second level cache like EHCache is one of the most popular second level cache vendor using this we can implement.

POJO/Entity Life cycle



Transient State

An object is said to be in a transient state if it is not associated with the session and has no matching record in the database table.

For example

Account account=new Account();

account.setAccno(101);

account.setName("Amol");

account.setBalance(12000);

Persistent State

An object is said to be in a persistent state if it is associated with session object (L1 cache) and will result into a matching record in the database table.(i.e upon commit) session.save(account); tx.commit();

or

Account account=session.get(Account.class,102);

OR via HQL/JPQL

Note

When the POJO is in persistent state it will be in synchronization with the matching record in DB i.e if we make any changes to the state of persistent POJO it will be reflected in the Database.

(after commiting tx) -- i.e automatic dirty checking will be performed(resulting in insert/update/delete)

IACSD Advance Java

Detached state

Object is not associated with the session but has a matching record in the database table. If we make any changes to the state of a detached object it will NOT be reflected in the database.

session.clear();

session.evict(Object);

session.close();

Note: By calling update method on session object it will go from detached state to persistent state.

By calling delete method on session object it will go from persistent state to transient state.

Hibernate's Automatic Dirty checking

The process of automatically updating the database with the changes to the persistent object when the session is flushed (@ commit) is known as automatic dirty checking.

An object (POJO) enters persistent state when any one of the following happens:

When the code invokes session.save, session.persist or session.saveorUpdate or session.merge

OR

When the code invokes session.load or session.get OR

Result of JPQL

Any changes to a persistent object are automatically saved to the database when the session is flushed.

Flushing is the process of synchronizing the underlying database with the objects in the session's L1 cache.

Even though there is a session. flush method available but you generally don't need to invoke it explicitly.

A session gets flushed when the transaction is committed.

Methods of Session API

public void persist(Object ref)

Persists specified transient POJO on underlying DB , upon committing the transaction.

void clear()

When clear() is called on session object all the objects associated

with the session object become detached. But Database Connection is not closed.

(Completely clears the session. Evicts all loaded instances and cancel all pending saves, updates and deletions)

void close()

When close() is called on session object all the objects associated with the session object become detached and also closes the Database Connection.

public void evict(Object ref)

It detaches a particular persistent object, detached or disassociates from the session.

(Remove this instance from the session cache. Changes to the instance will not be synchronized with the database.)

void flush()

When the object is in a persistent state, whatever changes we made to the object state will be reflected in the database only at the end of transaction.

If we want to reflect the changes before the end of the transaction (i.e before committing the transaction) call the flush method.

(Flushing is the process of synchronizing the underlying DB state with persistable state of session cache)

boolean contains(Object ref)

The method indicates whether the object is associated with the session or not.

Void refresh(Object ref) -- ref --persistent or detached

This method is used to get the latest data from database and make corresponding modifications to the persistent object state.

(Re-read the state of the given instance from the underlying database)

public void update(Object ref)

Note :-

If an object is in a persistent state there is no need to call the update method.

As the object is in sync with the database whatever changes made to the object will be reflected to the database at the end of transaction.

eg --- updateAccount(Account a, double amt)

```
{
  Session ses;
  Transaction tx;
  sop(a);
  set amt
  ses.update(a);
  sop(a);
}
```


When the object is in detached state, record is present in the table but object is not in sync with database, therefore update() method can be called to update the record in the table.

Which exceptions update method can raise?

1. StaleStateException -- If u are trying to update a record (using session.update(ref)), whose id doesn't exist.

i.e update can't transition from transient --->persistent

It can only transition from detached --->persistent.

eg -- update_book.jsp -- supply updated details + id which doesn't exist on db.

2. NonUniqueObjectException -- If there is already a persistence instance with the same id in session.

eg -- UpdateContactAddress.java

public Object merge(Object ref)

Can Transition from transient -->persistent & detached --->persistent.

Regarding Hibernate merge

1. The state of a transient or detached instance may also be made persistent as a new persistent instance by calling merge(). API of Session – Object merge(Object object)

2. Copies the state of the given object(can be passed as transient or detached) onto the persistent object with the same identifier.

3. If there is no persistent instance currently associated with the session, it will be loaded.

4. Return the persistent instance. If the given instance is unsaved, save a copy of and return it as a newly persistent instance. The given instance does not become associated with the session.

5. will not throw NonUniqueObjectException --Even If there is already a persistence instance with the same id in session.

public void saveOrUpdate(Object ref)

The method persists the object (insert) if matching record is not found (& id inited to default value) or fires update query If u supply Object , with non-existing ID -- Fires StaleStateException.

lock()

When the lock() method is called on the session object for a persistent object, until the transaction is committed in the hibernate application , externally the matching record in the table cannot be modified.

session.lock(object,LockMode);

eg - session.lock(account,LockMode.UPGRADE);

When will persistent POJO/s become detached in the following cases ?

openSession : when prog explicitly calls : session.close (typically in a finally block) OR try-with-resources

getCurrentSession : session auto closes upon tx boundary(commit or rollback)

Hibernate performs auto dirty checking upon commit(session.flush) : i.e it will check the state of L1 cache against DB & auto fires DML (eg : insert/update/delete)

If you modify the state(via setters) of PERSISTENT entity/pojo : Hibernate performs auto dirty checking @ commit (resulting in insert/update/delete) i.e it will auto sync up the state of DB with that of L1 cache.

If you modify the state(via setters of pojo) of DETACHED entity/pojo : Hibernate CAN NOT perform auto dirty checking , since it's already de coupled from L1 cache

i.e it will NOT auto sync up the state of DB with that of L1 cache.

POJO Annotations

Package : javax.persistence

@Entity : Mandatory : cls level

@Id : Mandatory : field level or property (getter) : PK

Optional annotation for further customization :

@Table(name="tbl_name) : to specify table name n more

@GeneratedValue : to tell hibernate to auto generate ids

auto / identity(auto incr : Mysql) / table / sequence(oracle)

eg : @Id => PK

@GeneratedValue(strategy=GenarationType.IDENTITY) => auto increment

@Column(name,unique,nullable,insertable,updatable,length,columnDefinition="double(8,2)") : for specifying col details

@Transient : Skipped from persistence(no col will be generated in DB table)

@Temporal : java.util.Date , Calendar , GregorianCalendar LocalDate(date) ,LocalTime(time) , LocalDateTime (timestamp/datetime) : no temporal annotation.

@Lob : BLOB(byte[]) n CLOB(char[]) : saving / restoring large bin /char data to/from DB

@Enumerated (EnumType.STRING): enum (def : ordinal : int)

Steps of creating hibernate full fledge application - Maven project

1. **org.hibernate.Session :** interface (implementation classes : hibernate core jar)

Represents : Main runtime i/f for prog interaction with hibernate

Supplies CRUD APIs(eg : save, persist, get,load, createQuery,update,delete....)

Represents : wrapper around pooled out db connection.

It has INHERENTLY L1 cache(persistence ctx) associated with it.

DAO layer creates session instance as per demand(one per request for CRUD operation)

light weight , thread unsafe object

NO NEED for accessing the session in synchronized manner : since different thread representing different client requests , will have their own session object

2. org.hibernate.SessionFactory : interface (imple classes : hibernate core jar)

JOB : to provide session objects(openSession / getCurrentSession)

singleton instance per DB / application

immutable , inherently thread safe

Represents : DB specific config , including connection pool.

It has L2 cache associated with it : MUST be configured explicitly

3. Configuration : org.hibernate.cfg.Configuration class.

Provider of SF

4. Additional APIs- Transaction, Query, CriteriaQuery ...

5. hibernate.cfg.xml : centralized configuration file , to create SessionFactory (i.e bootstrapping hibernate framework)

Important property : hibernate.hbm2ddl.auto=update

Checks if table is not yet created for a POJO : create a new table. BUT if table already exists : continues with the existing table.

5.1 HibernateUtils --- create new java file to create singleton immutable SF instance

6. Using of POJO Annotations - and Add <mapping class="F.Q POJO class name"/> in hibernate.cfg.xml

7. Create DAO i/f & write its implementation class - Hibernate based DAO impl class

7.1 No data members ,constructor , cleanup

7.2 Directly add CRUD methods.

Steps in CRUD methods

1. Get hibernate session from SF

API of org.hibernate.SessionFactory

public Session openSession() throws HibernateException

OR

public Session getCurrentSession() throws HibernateException

2. Begin a Transaction
API of Session
public Transaction beginTransaction()throws HibernateException

3. try {
perform CRUD using Session API (eg : save/get/persist/update/delete/JPQL...)
commit the tx.
} catch(RuntimeException e)
{
roll back tx.
re throw the exc to caller
} finally {
close session --destroys L1 cache , pooled out db cn rets to the pool.
}

4 Refer to Hibernate Session API
(hibernate api-docs & readme : hibernate session api)

5. Create main(..) based Tester & test the application.

Which of the following layers are currently hibernate specific(native hibernate) ?
DAO : org.hibernate.SF , Session, Transaction,Query... : hibernate specific
POJO : javax.persistence : annotations => hibernate inde. (JPA compliant)
Utils : Configuration , org.hibernate.SF => hibernate specific
6. Add a breakpoint before commit, observe and conclude.

7. Replace openSession by getCurrentSession

---

8. Objective : Get user details
I/P : user id
O/P : User details or error
API : session.get

9. Confirm L1 cache
by invoking session.get(...) multiple times.

10. Hibernate POJO states :
transient , persistent , detached.

11. Objective : Display all user details
Can you solve it using session.get ? : NO
sql : "select * from users_tbl"
hql : "from User"
jpql : "select u123 from User u123"
u : POJO alias

11.1 Solve it using HQL(Hibernate query language)/JPQL (Java Persistence Query Language)

Object oriented query language, where table names are replaced by POJO class names & column names are replaced by POJO property names, in case sensitive manner.

11.2. Create Query Object --- from Session i/f
<T> org.hibernate.query.Query<T> createQuery(String jpql/hql,Class<T> resultType)
T --result type.

11.3. To execute query
Query i/f method
public List<T> getResultList() throws HibernateException
--Rets list of PERSISTENT entities.

12. Objective : Display all users registered between start date and end date & under a specific role
I/P : begin dt , end date , role
eg : sql = select * from users where reg_date between ? and ? and user_role=?
jpql ="select u from User u where u.regDate between :strt and :end and u.userRole=:role";

Passing named IN params to the query
Query i/f method
public Query<T> setParameter(String paramName,Object value) throws HibernateException.

Steps in Eclipse IDE for creating a Maven based dynamic web app with Hibernate and MySql

1. Choose File --New (or from drop down list) --Maven project --Create a simple project (no archetype)--finish
Choose packaging as "war"

2. Copy entries from pom.xml (from earlier hibernate project)
Copy this additional entry for web app (WAR file creation)

```
<plugin>
 <artifactId>maven-war-plugin</artifactId>
 <version>3.0.0</version>
 <configuration>
  <warName>artifact-id</warName>
 </configuration>
</plugin>
```

3. R click on project --maven --update project (Force update!)

4. R click on project --Java EE Tools -- Generate Deployment Descriptor stub
(It will add web.xml under WEB-INF)
Edit web app version from 2.5 to 3.1

5. R click on project --properties --targeted runtime -- Choose Apache Tomcat 9.0--finish

6. In case of any errors, choose Project tab at the top ---clean project

7. Confirm if JDK version updated to JDK 11 & dependencies are added.

8. Write index.jsp, with changing timestamp.

R Click on project --Run as --run on server --choose Tomcat server --finish

Relationship between Entity and Entity -(Inheritance , Association : HAS-A--aggregation and later composition)

Types of associations
one-to-one
one-to-many
many-to-one
many-to-many

1. @MappedSuperlass : => inheritance in hibernate.
JPA compliant --javax.persistence
Details : appears in a super class , has no corresponding table. All other entities can extend from same.

2. Many to Many association between Course and Student , with additional columns (eg : admission date, status)
Steps
1. Create entities -- Course , Student
Do u need a bi-dir asso or uni dir ? **uni dir**
Course 1<------* Admission
Student 1<----* Admission

eg : In Admission entity
@ManyToOne //mandatory , o.w hib throws MappingExc
@JoinColumn(name="s_id")
private Student student;

@ManyToOne //mandatory , o.w hib throws MappingExc
@JoinColumn(name="c_id")
private Course course;

Which tables ? courses ,students , admissions
admissions : col names of FKs --will be generated by hibernate
Can it be replaced by prog supplied names ? YES -- @JoinColumn

3. One to one (This can be created in uni directional as well as bi directional manner)
eg of bi-dir :
Student 1<---->1 Address

eg : In Student :
....
+Address

In Address class
....
+Student
Annotation : @OneToOne

In any bi dir , one to one (or one to many or many to many) relationship -- hibernate --CAN NOT figure out automatically : which is the owning side n which is the inverse side.

Owning side --is the side where FK appears
Inverse side(non owning) of the asso---- opposite side , where FK does not appear

In one-to-one , as well as many-to many --any side can be the owning side . NO particular rules. You can decide based upon data navigation requirements.

In a bi-dir one-to-one what will happen if you don't specify mappedBy ?
Ans : Hibernate will create FKs in both sides

What is mappedBy
mappedBy --attr / elem of @OneToOne / @OneToMany / @ManyToMany
where does it appear : non owning side /inverse side/one side
what should be its value -- name of the association property as it appears in the owning side.

eg : Suppose Student --owning side
Address --inverse side
eg : In Student POJO : owning side

```
@OneToOne
@JoinColumn(name="adr_id")
private Address address;
```

In Address POJO :

```
@OneToOne(mappedBy ="address")
private Student student;
```

Objective
Link address details
User i/p : student id (PK) , adr details

Objective
Get Student details
i/p : email

If you want address details ALWAYS lifted along with student details : above soln works!
Why ?

JPA/ Hibernate has default fetching policies for all the associations -
one -to -one : EAGER
one-to-many : LAZY
many-to-one : EAGER
many-to-many : LAZY

Can you change the many-to-one : EAGER to LAZY ? YES n it will be applied -Now let's try adding fetch type to one-to-one asso.
Observations : Hibernate DOES NOT change the fetch type to LAZY.(i.e it will always work in EAGER manner)
CORRECTION : If you change the fetch=FetchType.LAZY , in the owning side of the association , hibernate DOES behave in the lazy manner(i.e if Student is the owning side having FK n if you use
session.get(Student.class, studentId) OR JPQL --it will fire the select query ONLY on the students table)
BUT if you add fetch=FetchType.LAZY , in the non-owning(i.e inverse) side of the association , hibernate DOES NOT behave in the lazy manner , and continues to behave in the eager manner.

BUT if you don't want it -- i.e. want to fetch only student details --solution exists

Since default fetching policy of one to one is eager, while accessing Student details , it will automatically load address details always.

Test it
Get Student Details
User i/p : student id
Check the o/p
Even after , changing the fetch type to LAZY , it still acts in EAGER manner
So a better suggestion : create it in a unidirectional manner (typically!)

As uni dir association
Options
1. Student ---> Address
OR
2. Address ---> Student

Which one is suitable to avoid additional address details lifted every time along with the student ?
Ans : 2

Answer this
If you are configuring one-to-one association between entities , which is a better approach ?

1. Both entities having their separate PKs

OR

2. Sharing the same PK

Answer : 2

How to map a one-to-one association in which the primary key value of one entity(Studen't PK) is also used as the primary key value of the other entity(eg : Address) ?

JPA Annotation : @MapsId

Solution:

You can use JPA's @MapsId annotation to tell Hibernate that it will use the foreign key of an associated entity as the primary key.

eg : In address table : PK will also work as FK referencing the PK of students tables

How to fetch the complete details , in a single join query ?

Using "join fetch" keyword in JPQL

eg : String projectJPQL = "select p from Project p join fetch p.students where p.title=:title";

Entity Types :

1. If an object has its own database identity (primary key value) then it's type is Entity Type.
2. An entity has its own lifecycle. It may exist independently of any other entity.
3. An object reference to an entity instance is persisted as a reference in the database (a foreign key value).

eg : College is an Entity Type. It has it's own database identity (It has primary key).

4. Mandatory annotations -- @Entity + @Id

Value Types :

1. If an object does not have its own database identity (no primary key value) then it's type is Value Type.
2. Value Type object belongs to an Entity Type Object.
3. It's embedded in the owning entity and it represents the table column in the database.
4. The lifespan of a value type instance is bounded by the lifespan of the owning entity instance.
5. Annotation -- @Embeddable

Different type in Hibernate --- Entity Type or a Value Type

Entity Type	Value Type
1. Has its own DB identity -- Primary key (mapped by @Id)	1. Doesn't have any DB identity (no @Id annotation)
2.An entity has its own lifecycle -- it may exist independently of any other entity.	2. Owned by Entity, no independent life-cycle. Life cycle bounded by the life cycle of owning entity.
3. Supports shared references. eg Item & category. 2 Items can share the same category.	3. Doesn't support shared references.
4. eg Item,Category,User,Order.	4. eg all Java types are stored as value type. UDTs also can be mapped as value types(a.k.a components in hibernate jargon) eg Person has hobbies . Or Person has Address.

Different types of Value Types

Basic, Composite, Collection Value Types :

1. Basic Value Types :

Basic value types are : they map a single database value (column) to a single, non-aggregated Java type.

Hibernate provides a number of built-in basic types.

all primitive types ,String, Character, Boolean, Integer, Long, Byte, ... etc.

Optional : @Basic

2. Composite Value Types :

In JPA composite types also called Embedded Types. Hibernate traditionally called them Components.

2.1 Composite Value type looks like exactly an Entity, but does not have it's own lifecycle and identifier.

Will never have : @Entity & @Id

Annotations Used

1. @Embeddable : Mandatory

Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity. It doesn't have own identifier.

eg : Address is eg of Embeddable

Student HAS-A Address(eg of Composition --i.e Address can't exist w/o its owning Entity i.e Student)

College HAS-A Address (eg of Composition --i.e Address can't exist w/o its owning Entity i.e College)

BUT Student will have its own copy of Address & so will College(i.e Value Types don't support shared reference)

2. @Embedded : optional

Specifies a persistent field or property of an entity whose value is an instance of an embeddable class. The embeddable class must be annotated as Embeddable.

eg : Address is embedded in College and User Objects.

3. @AttributesOverride :

Used to override the mapping of a Basic (whether explicit or default) property or field or Id property or field.

In Database tables observe the column names. Student table having STREET_ADDRESS column and College table having STREET column. These two columns should map with the same Address field streetAddress. @AttributeOverride gives a solution for this.

To override multiple column names for the same field use @AtributeOverrides annotation.

eg : In Student class :

```
@Embedded
@AttributeOverride(name="streetAddress", column=@Column(name="STREET_ADDRESS"))
private Address address;
```

where , name --POJO property name in Address class

3. Collection Value Types :

Hibernate allows to persist collections.

But Collection value Types can be either collection of Basic value types, Composite types and custom types.

eg :

Collection mapping means mapping a group of values to a single field or property. But we can't store list of values in a single table column in the database. It has to be done in a separate table.

eg : Collection of embeddables

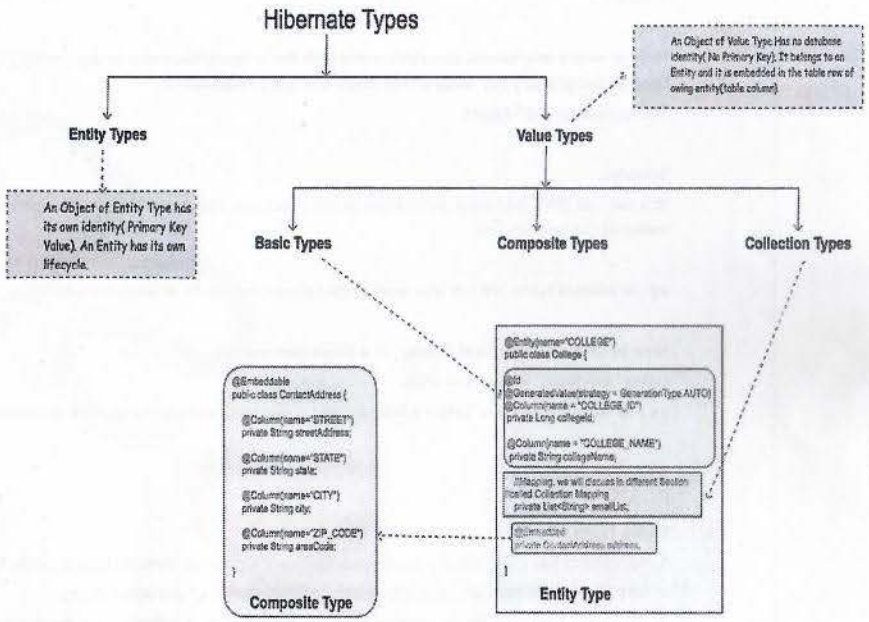
```
@ElementCollection
  @CollectionTable(name="CONTACT_ADDRESS", joinColumns=@JoinColumn(name="USER_ID"))
  @AttributeOverride(name="streetAddress", column=@Column(name="STREET_ADDRESS"))
  private List<ContactAddress> address;
```

eg : collection of basic type

```
  @ElementCollection
  @CollectionTable(name="Contacts", joinColumns=@JoinColumn(name="ID"))
```

IACSD Advance Java

```
@Column(name="CONTACT_NO")
private Collection<String> contacts;
```



Hibernate Project Example- Maven project

Steps for Hibernate + Java SE

1. Change perspective to Java

2. Create Maven Project ---skip archetype selection --
grp id,artifact id , packaging option : jar --finish

3. Creates default pom.xml . Add <build> n <dependencies> tags
4. Update the project .
R click on the project --> Maven --> Update Project -->select Force update checkbox -->Finish

4. Copy hibernate.cfg.xml under : src/main/resources .
Edit hibernate.cfg.xml , as per your DB settings.

6. Create HibernateUtils class to create singleton , immutable , inherently thrd safe SessionFactory instance.

7. Create a class TestHibernate under <tester> package.
Add following code.

```
import static utils.HibernateUtils.*;
import org.hibernate.*;

public class TestHibernate {

	public static void main(String[] args) {
		try(SessionFactory sf=getSf())
		{
			System.out.println("Hibernate booted.....");
		}catch (Exception e) {
			e.printStackTrace();
		}

	}

}
```

8. Run this as "java application" , check console to see ,
sf created & hib booted .

Above confirms bootstrapping of hibernate framework.

9. Create a POJO n test auto table creation
9.1 Create a User POJO
Add these Data members
userId (PK) ,name,email,password,confirmPassword,role(enum), regAmount;
LocalDate/Date regDate;
byte[] image;
Add JPA annotations (javax.persistence)
Confirm auto table creation.

9.2 Add <mapping> entry per POJO in hibernate.cfg.xml
Run TestHibernate to confirm auto table creation.

10. Create Hibernate based DAO layer , to insert a record.
10.1 DAO layer i/f
String registerUser(User user);
10.2 Hibenrate based DAO implementation class
no data mebers, no constr,no clean up
CRUD method

11. Create a main(...) based tester to test entire application , for user registration.

Example Hibernate - Maven Project

DBConnectivity - HibernateUtils .java

```
package utils;

//hibernate native API
import org.hibernate.*;
import org.hibernate.cfg.Configuration;

public class HibernateUtils {

	private static SessionFactory factory;
	static {
		System.out.println("in static init block");
		factory = new Configuration().configure().buildSessionFactory();
	}
	public static SessionFactory getFactory() {
		return factory;
	}
}
```

Pojo - User.java

```
package pojos;
/*
 * userId (PK) ,name,email,password,confirmPassword,role(enum), regAmount;
     LocalDate/Date regDate;
     byte[] image;
 */


import java.time.LocalDate;
import javax.persistence.*;


@Entity //MANDATRY class level anno meant for hibernate => following is entity
class
@Table(name = "users_tbl")
public class User {
    @Id //mandatory field or property level(getter) annotation => PK
    @GeneratedValue(strategy = GenerationType.IDENTITY) //=> constraint : auto
increment : OPTIONAL
    @Column(name = "user_id") //col name
    private Integer userId;
    @Column(length = 30)//=>varchar(30)
    private String name;
    @Column(length = 30,unique = true)//=>varchar(30) , unique constraint
```

```
    private String email;
    @Column(length = 20)
    private String password;
    @Transient //=> skip from persistence( no col.)
    private String confirmPassword;
    @Enumerated(EnumType.STRING) //=> col type : varchar storing enum constant
name
    @Column(length = 20,name="user_role")
    private Role userRole;
    @Column(name="reg_amount")
    private double regAmount;
    @Column(name="reg_date")
    private LocalDate regDate;//col type : date
    @Lob
    private byte[] image;
    public User() {
        System.out.println("in def ctor of "+getClass());
    }

    public User(String name, String email, String password, String
confirmPassword, Role userRole, double regAmount,
            LocalDate regDate) {
        super();
        this.name = name;
        this.email = email;
        this.password = password;
        this.confirmPassword = confirmPassword;
        this.userRole = userRole;
        this.regAmount = regAmount;
        this.regDate = regDate;
    }
    //parameterized ctor
    public User(String name, String email, double regAmount, LocalDate regDate) {
        super();
        this.name = name;
        this.email = email;
        this.regAmount = regAmount;
        this.regDate = regDate;
    }


    //all setters n getters
    public Integer getUserId() {
        return userId;
    }



    public void setUserId(Integer userId) {
        this.userId = userId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
```



```
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getConfirmPassword() {
        return confirmPassword;
    }
    public void setConfirmPassword(String confirmPassword) {
        this.confirmPassword = confirmPassword;
    }
    public Role getUserRole() {
        return userRole;
    }
    public void setUserRole(Role userRole) {
        this.userRole = userRole;
    }
    public double getRegAmount() {
        return regAmount;
    }
    public void setRegAmount(double regAmount) {
        this.regAmount = regAmount;
    }
    public LocalDate getRegDate() {
        return regDate;
    }
    public void setRegDate(LocalDate regDate) {
        this.regDate = regDate;
    }
    public byte[] getImage() {
        return image;
    }
    public void setImage(byte[] image) {
        this.image = image;
    }
    @Override
    public String toString() {
        return "User [userId=" + userId + ", name=" + name + ", email=" + email +
", userRole=" + userRole
                + ", regAmount=" + regAmount + ", regDate=" + regDate;
    }


}
```

Dao-
interface - IUserDao .java

```
package dao;


import java.io.IOError;
import java.io.IOException;
import java.time.LocalDate;
import java.util.List;
```

```
import pojos.Role;
import pojos.User;

public interface IUserDao {
   String saveUser(User user);

   String saveUserWithGetCurntSession(User user);

   User getUserDetails(int userId);

   List<User> getAllUserDetails();

   List<User> getSelectedUsers(LocalDate start, LocalDate end, Role role);

   List<String> getSelectedUserNames(LocalDate start);

   List<User> getSelectedUserDetails(LocalDate start);

   String changePassword(String email, String oldPwd, String newPwd);

   String applyBulkDiscount(LocalDate date, double discount);

   String deleteUserDetails(int userId);

   String bulkDeleteUsers(LocalDate date, double amount);

   String saveImage(int userId, String fileName) throws IOException;

   String restoreImage(int userId, String newFileName)throws IOException;
}
```

Implementation Class - `UserDaoImpl.java`

```
package dao;

import pojos.Role;
import pojos.User;
import static utils.HibernateUtils.getFactory;

import java.io.File;
```



```
import java.io.IOException;
import java.time.LocalDate;
import java.util.List;

import org.apache.commons.io.FileUtils;
import org.hibernate.*;

public class UserDaoImpl implements IUserDao {

  @Override
  public String saveUser(User user) {
    // get hibernate session from SF using openSession API
    Session session = getFactory().openSession(); // L1 cache created along with hib session n db conn pooled out
    Session session2 = getFactory().openSession();
    System.out.println(session.getClass());
    System.out.println("same sessions? " + (session == session2));// f
    // begin hibernate transaction(tx)
    Transaction tx = session.beginTransaction();
    System.out.println("is open " + session.isOpen() + " is connected " + session.isConnected());// t t
    try {
      // Session API : public Serializable save(Object entity)
      session.save(user);
      // => success --commit
      tx.commit();// hib fires insert query (DML)
      System.out.println("is open " + session.isOpen() + " is connected " + session.isConnected());// t t

    } catch (RuntimeException e) {
      // failure -- rollback
      if (tx != null)
        tx.rollback();
      throw e;// re throwing SAME exc to the caller for info.
    } finally {
      if (session != null)
        session.close();// L1 cache is destroyed n pooled out db cn , rets to the pool
    }
    System.out.println("is open " + session.isOpen() + " is connected " + session.isConnected());// f f

    return "User details inserted with user ID " + user.getUserId();
  }

  @Override
  public String saveUserWithGetCurntSession(User user) {
    // user : TRANSIENT -- exists only in heap , neither in
    // L1 cache nor in DB
    // get hibernate session from SF using getCurrentSession API
    Session session = getFactory().getCurrentSession(); // L1 cache created along with hib session n db conn pooled
                                                        // out

    // begin hibernate transaction(tx)
    Transaction tx = session.beginTransaction();
```

```
    try {
      // Session API : public Serializable save(Object transientEntity)
      // Session API : public void persist(Object transientEntity)
      session.persist(user);
      // session.save(user);// user : PERSISTENT -- part of L1 cache BUT rec not yet
      // inserted in DB
      // => success --commit
      tx.commit();// Hibernate performs : auto dirty chking (=flushing the session on DB) --- hib
              // fires insert query (DML)
      // close session -- destroys L1 cache-- pooled out db cn rets to the pool.


    } catch (RuntimeException e) {
      // failure -- rollback
      if (tx != null)
        tx.rollback();
      System.out.println("is open " + session.isOpen() + " is connected " + session.isConnected());// f f
      throw e;// re throwing SAME exc to the caller for info.
    }
    // user : DETACHED
    return "User details inserted with user ID " + user.getUserId();
  }


  @Override
  public User getUserDetails(int userId) {
    User user = null;// user : does not exist!
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      // API of Session : public <T> T get(Class<T> cls, Serializable id) throws
      // HibernateExc
      user = session.get(User.class, userId);// in case of valid id --user : PERSISTENT , has a corresponding rec
                          // in DB n it's part of L1 cache


      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    return user;
  }


  @Override
  public List<User> getAllUserDetails() {
    List<User> users = null;
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      users = session.createQuery("select u from User u", User.class).getResultList();
```



```
//      users = session.createQuery("select u from User u", User.class).getResultList();
//      users = session.createQuery("select u from User u", User.class).getResultList();
      // users -- list of PERSISTENT entities(part of L1 cache n has DB identity)
      tx.commit();// auto dirty chking --no DMLs -- L1 cache destroyed , session closed --rets
              // pooled out db cn to the pool
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    return users;// users -- list of DETACHED entities
  }


  @Override
  public List<User> getSelectedUsers(LocalDate startDate, LocalDate endDate, Role role1) {
    String jpql = "select u from User u where u.regDate between :start and :end and u.userRole=:role";
    List<User> users = null;
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      users = session.createQuery(jpql, User.class).setParameter("start", startDate).setParameter("end", endDate)
          .setParameter("role", role1).getResultList();
      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    return users;
  }


  @Override
  public List<String> getSelectedUserNames(LocalDate start) {
    String jpql = "select u.name from User u where u.regDate > :dt";
    List<String> names = null;
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      names = session.createQuery(jpql, String.class).setParameter("dt", start).getResultList();
      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    return names;
  }


  @Override
  public List<User> getSelectedUserDetails(LocalDate start) {
```

```
    String jpql = "select new pojos.User(name,email,regAmount,regDate) from User u where u.regDate > :dt";
    List<User> users = null;
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      users = session.createQuery(jpql, User.class).setParameter("dt", start).getResultList();
      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    return users;
  }


//old pwd : 1234 ,new pwd 5678
  @Override
  public String changePassword(String email, String oldPwd, String newPwd) {
    String mesg = "Password updating failed....";
    String jpql = "select u from User u where u.email=:em and u.password=:pass";
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    User u = null;
    try {
      u = session.createQuery(jpql, User.class).setParameter("em", email).setParameter("pass", oldPwd)
          .getSingleResult();// in case of invalid login -- throws javax.persistence.NoResultExc
      // => valid login , u : PERSISTENT
      u.setPassword(newPwd);// modifying state of the PERSISTENT entity in L1 cache
      // session.evict(u);//u : DETACHED from L1 cache
      tx.commit();// hib perform auto dirty chking --session.flush -- DML : update --- session
                  // closed
      mesg = "Password updated....";
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    u.setPassword("334455");// u : DETACHED , i.e modifying state of the DETACHED entity--hib CAN NOT
                            // propagate the changes to db
    return mesg;
  }


  @Override
  public String applyBulkDiscount(LocalDate date, double discount) {
    String mesg = "Bulk updation failed!!!!!!!!!!!!";
    String jpql = "update User u set u.regAmount=u.regAmount-:disc where u.userRole=:role and u.regDate< :dt";
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
```



```
      int updateCount = session.createQuery(jpql).setParameter("disc", discount)
          .setParameter("role", Role.CUSTOMER).setParameter("dt", date).executeUpdate();
      // L1 cache : EMPTY (i.e update query by passes the cache!)
      tx.commit();
      mesg = updateCount + " users got discount....";
    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }
    return mesg;
  }


  @Override
  public String deleteUserDetails(int userId) {
    String mesg = "Un subscription failed : invalid user id !!!!!!!!!!!!!!!";
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      // get user from it's id
      User user = session.get(User.class, userId);// int --> Integer ---> up casting --> Serializable
      if (user != null) {
        // user : PERSISTENT
        session.delete(user); // entity is simply marked for removal! : user --REMOVED
        mesg = "User with name " + user.getName() + " un -subscribed....";
      }
      tx.commit();// session.flush --auto dirty chking --delete query --rec is deleted from DB ,
            // session.close=> L1 cache is destroyed , DB cn rets to the pool
      // user : TRANSIENT


    } catch (RuntimeException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    }


    return mesg;
  }// Are there any objs marked for GC ? YES (user)


  @Override
  public String bulkDeleteUsers(LocalDate date, double amount) {
    String mesg = "Bulk Deletion failed !!!!!!!!!!!!";
    String jpql = "delete from User u where u.regDate > :dt and u.regAmount < :amt";
    // get Session from SF
    Session session = getFactory().getCurrentSession();
    // begin tx
    Transaction tx = session.beginTransaction();
    try {
      int updateCount = session.createQuery(jpql).setParameter("dt", date).setParameter("amt", amount)
          .executeUpdate();
      tx.commit();
```

```
            mesg = updateCount + " users deleted.....";


        } catch (RuntimeException e) {
            if (tx != null)
                tx.rollback();
            throw e;
        }
        return mesg;
    }


    @Override
    public String saveImage(int userId, String fileName) throws IOException {
        StringBuilder mesg = new StringBuilder("Image storing :  ");
        // get session from SF
        Session session = getFactory().getCurrentSession();
        // begin tx
        Transaction tx = session.beginTransaction();
        try {
            // 1. get persistent user from user id --session.get
            User user = session.get(User.class, userId);
            if (user != null) {
                // => valid user id i.e user : PERSISTENT
                // get byte[] from the file name
                // 2. File cls instance
                File file = new File(fileName);
                if (file.isFile() && file.canRead()) {
                    // If you modify the state of the PERSISTENT entity --hib will perform auto
                    // dirty chking n make changes to db
                    user.setImage(FileUtils.readFileToByteArray(file));
                    mesg.append("image saved.....");
                } else
                    mesg.append(" Failed : Invalid File name!!!!!!!!!!!!!!");


            } else
                mesg.append(" Failed : Invalid user id!!!!!!!!!!!!!");
            tx.commit();// hib : update
        } catch (Exception e) {
            if (tx != null)
                tx.rollback();
            throw e;
        }
        return mesg.toString();
    }


    @Override
    public String restoreImage(int userId, String newFileName) throws IOException {
        StringBuilder mesg = new StringBuilder("Image restoring :  ");
        // get Session from SF
        Session session = getFactory().getCurrentSession();
        // begin tx
        Transaction tx = session.beginTransaction();
        try {
            // 1. get user from it's id
```



```
            User user = session.get(User.class, userId);
            if (user != null) {
                //create file cls instance from file name
                File file=new File(newFileName);
                FileUtils.writeByteArrayToFile(file, user.getImage());
                mesg.append("successful...");
            } else
                mesg.append(" Failed : Invalid user id!!!!!!!!!!!!!");
            tx.commit();


        } catch (Exception e) {
            if (tx != null)
                tx.rollback();
            throw e;
        }


        return mesg.toString();
    }


}
```

Testers - as per requirement and request
1. GetUserDetails

```
package tester;


import static utils.HibernateUtils.getFactory;


import java.util.Scanner;


import org.hibernate.SessionFactory;


import dao.UserDaoImpl;


public class GetUserDetails {


    public static void main(String[] args) {
        try (SessionFactory sf = getFactory(); Scanner sc = new Scanner(System.in)) {
            UserDaoImpl dao = new UserDaoImpl();
            System.out.println("Enter user id");
            System.out.println(dao.getUserDetails(sc.nextInt()));
        } // sf.close() => immediate closing of cn pool , sc.close
        catch (Exception e) {
            e.printStackTrace();
        }


    }
```

```
}
```

2.

```
package tester;

import static utils.HibernateUtils.getFactory;

import java.time.LocalDate;
import java.util.Scanner;

import org.hibernate.SessionFactory;

import dao.UserDaoImpl;
import pojos.Role;
import pojos.User;

public class SaveUserDetails {

  public static void main(String[] args) {
    try (SessionFactory sf = getFactory(); Scanner sc = new Scanner(System.in)) {
      //create user dao instance
      UserDaoImpl dao=new UserDaoImpl();
      System.out.println(
          "Enter user details name, email, password, confirmPassword, userRole, regAmount
regDate(yr-mon-day)");
      User user=new User(sc.next(), sc.next(), sc.next(), sc.next(),
          Role.valueOf(sc.next().toUpperCase()),sc.nextDouble(),LocalDate.parse(sc.next()));
      System.out.println(dao.saveUser(user));
    } // sf.close() => immediate closing of cn pool
    catch (Exception e) {
      e.printStackTrace();
    }

  }

}
```


Spring framework

Spring is complete application development framework which is use to develop all type of application like standalone, distributed, EJB, Enterprise application.

Advantage of spring framework

1. Spring is **light-weight.**
2. Spring is **versatile.**
3. It is a non-invasive framework.
4. It provides **rapid application development.**
5. It has given boiler plate code to reduce development effort of programmer.
6. It has strong dependency management.
7. It provides loose coupling.

What is Non-Invasive Framework?

A framework that does not forces us to implement or extend any interface or class is called **non invasive framework.**

Example Spring is non-invasive because it passes all that need to our application development at run time.

What is Invasive Framework?

A framework that forces us to implement or extend any interface or class is called **invasive** framework.

e.g. Struts is invasive framework because it forces us to **extend** their own **ActionServlet** class to create normal application.

Why Spring

It simplifies Java development. It's one-stop-shop.
Excellent for integration with existing frameworks.
Reduces boiler-plate code.
Allows to build applications using loose coupling achieved via IoC(Inversion of control) & AOP(aspect oriented programming)

What is it ?

It is a container + framework.

Why Container

It manages the life cycle of spring beans (eg : controller,rest controller , dao,service)
spring bean : java object whose life cycle is managed by spring container(SC)

Why Framework ?

It provides readymade implementation of patterns & helps in building enterprise applications.

It is the most popular application development framework for enterprise Java. It is used to create high performant, easily testable, and reusable code.

Spring framework is an open source Java platform.
Founder is Rod Johnson and was first released under the Apache license in June 2003.
Currently hosted on Pivotal/VMware

Why Spring

1. Spring is lightweight .The basic version of Spring framework is around 2MB.

2. It supports in developing any Java application, but there are extensions(web MVC) for building web applications on top of the Java EE platform.

3. It helps programmers to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.
4. Excellent n easy testing support.(Thanks to D.I)

5. Supports smooth integration with ORM

6. Easy integration with web MVC applications including web sockets
(for async communication between server & client)
Spring's web framework is a web MVC framework, which is a great alternative to web frameworks such as Struts

7. It is organized in a modular fashion. Even though it's extensive , you have to worry only about the those modules that you need and ignore the rest.

8. Spring does not reinvent the wheel, instead it makes use of already existing frameworks like Hibernate , making its integration easier.

9. It translates technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

10. It provides a consistent transaction management interface that can support a local transaction (using a single database) as well as global transactions (using JTA over multiple databases).

Main winning feature of Spring is : loose coupling between the modules.

How does it achieve loose coupling ?
1. IoC – IoC is achieved using Dependency Injection(D.I)
2. Aspect Oriented Programming(AOP)

Why Spring framework came?
Designing application with J2EE was time consuming as well as complex to develop and has to write too much boiler plate code
What is Problem with J2EE?
J2EE API has lots of problems that is why people switched to spring framework.

IACSD Advance Java

1. It is **not giving boiler plate code,** It enforces us to write **boiler plate code**.
2. It is taking more time develop application.
3. It is taking **more cost to develop** the application.
4. It provides **tightly coupling.**

What is boilerplate code?
The code which is repeatedly used in our application development is called **Boiler Plate Code.**
For example in **JDBC** we have to write **4 or 5** line to **save a single record in our database.** J2EE API enforces us to write boiler plate code.

Why Spring is Light-Weight?
Spring framework is **Light-Weight** because it has provided all the modules separately.
When we want to use only **Core Module of Spring** then we can use only **core module,** no need to integrate other module like **J2EE** or **JDBC** ..etc

Modules of spring framework
There are six modules in spring framework, But this spring framework will vary with version to version.
Spring Modules
1. **Spring Core(IOC Container)**
2. **Spring MVC**
3. **Spring JDBC**
4. **AOP**
5. **ORM**
6. **JEE**

What is dependency injection in spring?
Dependency Injection is a process which is use to inject dependent object into target object by container(IOC Container) itself.

Advantage of Dependency Injection
1. **Dependency injection** makes the code loosely coupled so easy to maintain
2. It makes **configuration** and **code separation.**
3. makes the code easy to test

Who does manage dependency injection?
IOC Container is responsible to manage our object in spring framework.
What is IOC container in spring?
IOC Container is a principle which is use to manage and collaborate life cycle of object and it is responsible to instantiate and manage the objects.IOC Container is use to manage dependency between two class in spring framework.IOC Container is logical memory which is located on top of java JVM.
Types of IOC Container
1. BeanFactory
2. ApplicationContext

Spring Application Context

The ApplicationContext works on top of the BeanFactory (it's a subclass) and adds other functionality such as easier integration with Springs AOP features, message resource handling (for use in internationalization), event propagation, declarative mechanisms to create the ApplicationContext and optional parent contexts.

ApplicationContext, a superset of BeanFactory does support the Annotation-based dependency Injection in spring application.

Spring ApplicationContext implementations

ApplicationContext is an interface which is use to provide configuration information of bean to Spring ioc container and it is use to manage life cycle of object within the ioc container.

We can say ApplicationContext is another name of ioc container which is used to create the Spring ioc container.

Spring ApplicationContext implementations example

1. ClassPathXmlApplicationContext
2. FileSystemXmlApplicationContext
3. AnnotationConfigApplicationContext
4. AnnotationConfigWebApplicationContext
5. XmlPortletApplicationContext
6. XmlWebApplicationContext

e.g. ApplicationContext ctx= new ClasspathXmlApplicationContext("spring-context.xml");

What is Dependency Injection?

The Dependency injection is when all dependent objects are automatically binded when an instance is initialized.

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("/application-context.xml");

For example when we create bean in Spring XML configuration file then all dependent object will be binded at the time of creation itself.

<bean id="myBean" class="class_name"/>

What is Dependency Lookup?

Dependency lookup is concept which is trying to find a dependency .For example when we create ApplicationContext object in spring application then this ApplicationContext object is trying to get all dependency object from Spring XML Configuration file.

Note :Dependency Lookup **is slow** and Dependency injection **is fast**.

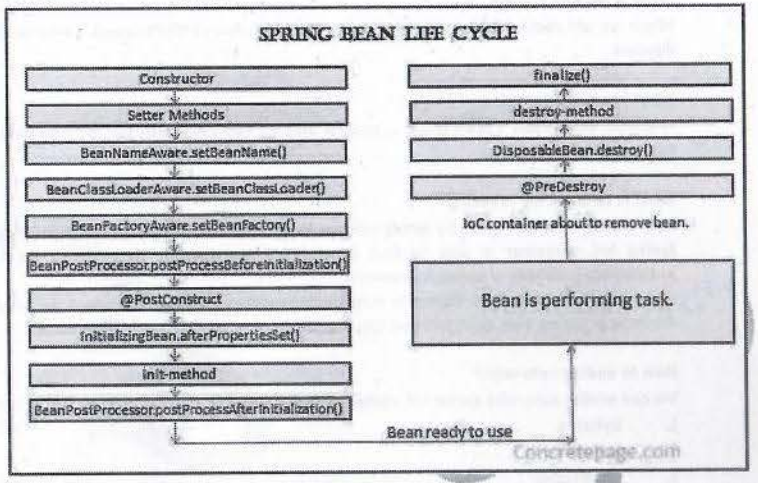
What is bean in spring?

A bean is an object which is instantiated, assembled, and managed by a Spring IoC container.

These beans are created with the spring XMLconfiguration file that you given to the IoC container at the time of creating BeanFactory or ApplicationContext.

Beans are use to configure database connection parameters, security etc in spring XML Confugurationfile. Beans are use to avoid hardcoding.Beans are use to manage the dependency of related class objects in context of IoCContainer.

How to create bean in spring?



Here id you have to pass in main class at the time of creating **BeanFactory** or **ApplicationContext**. And You have to pass fully qualified class path and name in main class.

What is bean inheritance in spring?

Inheritance is one of the most important features of OOPS of any language by which we can reuse the existing functionality.

Like any language spring also has given parent attribute to reuse existing bean feature.

What is bean scope in spring?

In spring framework all bean has a scope , means every **bean** has its own visibility.

When we declare a class as a **bean** then by **default** the bean will be created under **singleton** scope. How many types of bean scope

1. Singleton
2. Prototype
3. Request
4. Session
5. Global Session

What is singleton scope?

When will try to create multiple object of same bean then spring **IoC** container will return same object.

Singleton scope is default scope in bean. **Singleton** scope functionality is same as **singleton** class in java.

What is prototype scope?

When we will declare bean as **prototype** scope then every time **new object** will be return by spring **IoC** container.

What is request scope?
When we will declare a bean scope as **request** then for every **HTTPRequest** a **new bean instance** will be injected.

What is session scope?
When we will declare a bean scope as session then for every new **HttpSession new bean instance** will be injected.

What is auto-wiring in spring?
Auto-wiring is feature given by spring framework which is use to manage the dependency automatically. **Spring IoC** container is able to find dependency and manage dependency implicitly. By **default AutoWiring is disable in spring framework.**
If we want to enable **Auto-Wiring** to manage dependency automatically then we have to use autowire attribute **in spring XML configuration** file.

How to enable auto-wire?
We can enable auto-wire **spring IoC container** in four ways.
1. byName
2. byType
3. Constructor
4. Autodetect

Spring Project Structure -
Spring project
src/maim/java
src/main/resources
Bean-config.xml - require bean namespace- finish

More details about **<bean> tag**
Attributes
1. id --mandatory --bean unique id
2. class --- mandatory -- Fully qualified bean class name
Scope-pg|sing|proto - Singleton default in spring.
3. scope --- In Java SE --- singleton | prototype
In web app singleton | prototype | request | session
Default scope = singleton
singleton --- SC will share single bean instance for multiple requests/demands(via ctx.getBean)
prototype -- SC creates NEW bean instance per request/demand.

Lazy- at request time
Eager- at deployment time

Servlet object is created by wc at 1st request - which is lazy manner only single obj created if we want to make EagerManner- loadOnStartup- at deployment time

4. lazy-init --- boolean attribute. default value=false.
Applicable only to singleton beans. Means Egar default eager- lazy-init=false

And with prototype its lazy means not default value lazy-init=false not work its true only cause prototype

SC will auto create a singleton spring bean instance --- @ SC start up.

```
<!-- dependency config -->
 <!-- singleton n eager -->
 <bean id="test" class="dependency.TestTransport" />
 <!-- prototype n lazy -->
 <bean id="http" class="dependency.HttpTransport" scope="prototype" lazy-init="false"/>//ignore in prototype its lazy only
 <!-- singleton n lazy -->
 <bean id="soap" class="dependency.SoapTransport" scope="prototype" lazy-init="true"/> //lazy- true ignored at soap creation before calling setter
 <!-- dependent bean config -->
 <!--singleton n eager -->
 <bean id="my_atm" class="dependent.ATMImpl" scope="prototype">
   <!-- setter Based D.I -->
      <property name="myTransport" ref="soap" /> //first create soap then call setter method setmyTransport(ref)
 </bean>
```

Steps for Spring based Java SE application

0. In eclipse , change perspective to Java
1. Create Maven based Java SE project with spring dependencies
2. Create dependent n dependency classes
4. Refer : <resources> & create spring bean-config xml file.(Using STS support)
5. Add namespace <beans>
6. Configure dependency and dependent beans
7. Create a tester application , to start the Spring container & run this as java application , to confirm spring in core java.

Tester.java

```
package tester;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import dependent.ATMImpl
public class TestSpring {
 public static void main(String[] args) {
        try(ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("bean-config.xml")) {
 System.out.println("SC up n running !");
                // deposit funds : B.L
                // get rdy to use atm bean
 ATMImpl ref1 = ctx.getBean("my_atm", ATMImpl.class);// 1st demand
                ref1.deposit(12345);
 ATMImpl ref2 = ctx.getBean("my_atm", ATMImpl.class);// 2nd demand
 System.out.println(ref1 == ref2);// f
        } // ctx.close --> SC shut down --searches for singleton -- destroy -method
 // --invoked -- bean GC
        catch (Exception e) {
                e.printStackTrace();
        }

 }
}
```

Dependencies -

Transport.java - interface

```
package dependency;

public interface Transport {
 void informBank(byte[] data);
}
```

1. TestTransport.java - Implementation class

```
package dependency;

public class TestTransport implements Transport {
 public TestTransport() {
```



```
        System.out.println("in cnstr of " +getClass().getName());
 }
 @Override
 public void informBank(byte[] data) {
        System.out.println("informing bank using " + getClass().getName() + " layer");
 }
}
```

2. SoapTransport.java

```
package dependency;

public class SoapTransport implements Transport {
 public SoapTransport() {
        System.out.println("in cnstr of " +getClass().getName());
 }

 @Override
 public void informBank(byte[] data) {
        System.out.println("informing bank using " + getClass().getName() + " layer");
 }
}
```

3. HttpTransport.java

```
package dependency;

public class HttpTransport implements Transport {
 public HttpTransport() {
        System.out.println("in cnstr of " +getClass().getName());
 }

 @Override
 public void informBank(byte[] data) {
        System.out.println("informing bank using " + getClass().getName() + " layer");
 }
}
```

Dependent -

ATM.java - interface

```
package dependent;

public interface ATM {
  void deposit(double amt);
  void withdraw(double amt);
}
```

AtmImpl.java - class

```
package dependent;

import dependency.Transport;

public class ATMImpl implements ATM {
 private Transport myTransport;
 public ATMImpl() {
        System.out.println("in cnstr of " +getClass().getName()+" "+myTransport);
 }

 @Override
 public void deposit(double amt) {
        System.out.println("depositing "+amt);
        byte[] data=("depositing "+amt).getBytes();
        myTransport.informBank(data);
 }

 @Override
 public void withdraw(double amt) {
        System.out.println("withdrawing "+amt);
        byte[] data=("withdrawing "+amt).getBytes();
        myTransport.informBank(data);
 }
 //setter Based D.I
 public void setMyTransport(Transport myTransport) {
        this.myTransport = myTransport;
        System.out.println("in setter");
 }
 //custom init method
```



```
 public void myInit() {
        System.out.println("in init "+myTransport);//not null
 }
 //custom destroy method
        public void myDestroy() {
                System.out.println("in destroy "+myTransport);//not null
        }
 }
```

configuration file - bean-config.xml-

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
 <!-- configure dependency beans -->
 <!-- singleton n eager -->
 <bean id="test" class="dependency.TestTransport" />
 <!-- singleton n lazy : http -->
 <bean id="http" class="dependency.HttpTransport" lazy-init="true" />
 <!-- prototype soap -->
 <bean id="soap" class="dependency.SoapTransport" scope="prototype" />
 <!-- dependent bean : my_atm singleton n eager -->
 <bean id="my_atm" class="dependent.ATMImpl" init-method="myInit"
        destroy-method="myDestroy" scope="prototype">
        <!-- xml child tag for setter based D.I -->
        <property name="myTransport" ref="soap" />
 </bean>
</beans>
```

Execution -

Run Tester as java application

Spring Application using annotation approach along with database connectivity

Wiring=dependency injection=Making dependencies available to dependent beans (POJO) @ runtime via 3rd Party (currently spring container)

Types Of Wiring (Depenency Injection) in Spring Framework

LHS

Explicit Wiring --Must supply setters/constrs/factory methods + XML configuration for beans

① Setter based D.I
1. Provide setters in dependent bean.
2. Provide <property name & value/ref > tag in xml config file

② Constructor based D.I
1. Provide parameterized constructor
2. Provide <constructor-arg name/type/index value/ref > tag in xml config file.

③ Factory Method Based D.I
1. Can provide private constructor & no setters in dependent bean.
2. Provide parameterized factory method (public static instance returning) in dependent bean.
3.Provide in <bean > tag factory-method name & provide <constructor-arg> tags one per method argument.

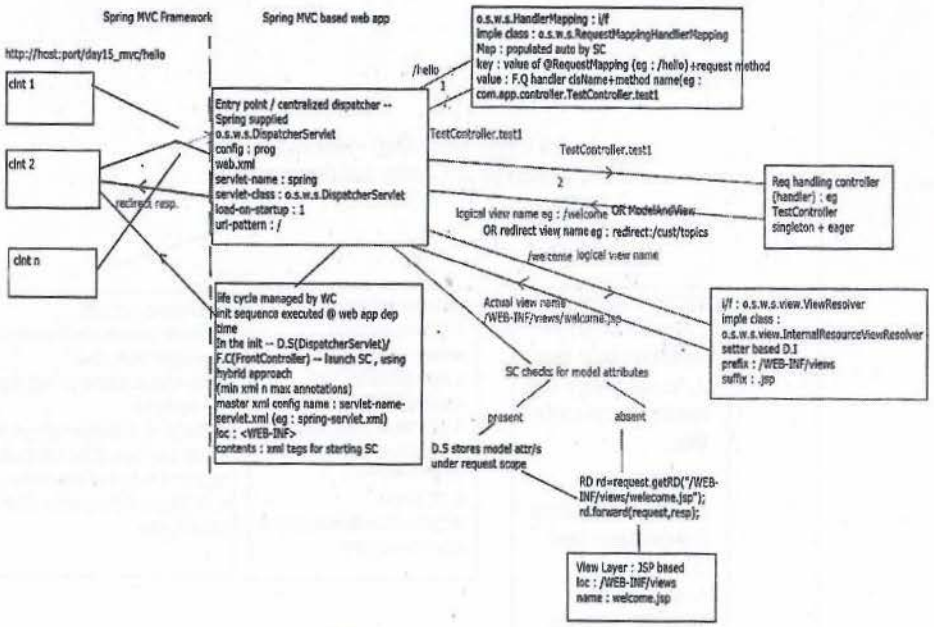
RHS

Implicit Wiring (auto wiring) --Must supply setters or constrs . No XML config. Just choose autowire mode

autowire=byName
1. Provide setters in dependent bean class.
2. No <property> tags required in xml config files.
3.Must match property setter names to dependency bean id.
4. No exception thrown if match not found.

autowire=byType
1. Provide setters in dependent bean class.
2. No <property> tags required in xml config file.
3. SC tries to match data type of the property to type of the dependency bean.
4. SC throws NoUniqueBeanDefException in case of ambiguity

autoWire=constructor
1.Provide parameterised constructor in dependent bean class.
2. No <constructor-arg> tag required in xml config file.
3. Similar to autoWire=byType. SC tries to match data type of the constructor argument to type of dependency bean.
4. SC throws NoUniqueBeanDefExc in case of ambiguity



Regarding form binding in Spring -

1. If there are multiple request params(use case -- register/update) --- bind POJO directly to a form.
(2 way form binding technique) —> How ?

1.1 For loading the form (in showForm method of the controller) , bind empty POJO (using def constr) in model map

How ?

Explicit --add Model as dependency & u add

map.addAttribute(nm,val)

OR better way

implicit -- add POJO as a dependency

eg : User registration

@GetMapping("/reg")

public String showForm(User u) {...}

What will SC do ?

SC --- User u=new User();

chks --- Are there any req params coming from client ? --- typically --no --- only getters will be called --

adds pojo as model attr (in Model map)

map.addAttribute("user",new User());

1.2 In form (view ---jsp) -use spring form tags along with modelAttribute

Steps

1. import spring supplied form tag lib

2. Specify the name of modelAttribute under which form data will be exposed.(name of model attr mentioned in the controller)

```
<s:form method="post" modelAttribute="user">
 <s:input path="email"/>.....
</s:form>
```

1.3 Upon form submission (clnt pull !)

clnt sends a new req --- containing req params

@PostMapping("/reg")

public String processForm(User u,RedirectAttributes flashMap,HttpSession hs) {

//SC calls

User u=new User();

SC invokes MATCHING (req param names --POJO prop setters)

setters. -- conversational state is transferred to Controller.

adds pojo as model attr (in Model map)

map.addAttribute("user",u)

Thus you get a populated POJO directly in controller w/o calling <jsp:setProperty> & w/o using any java bean.

Spring form tags

1. <form:input path="name" />

2. <form:input type="email" path="email" />

3. <form:password path="password" />

4. <form:textarea path="notes" rows="3" cols="20"/>

5. <form:checkboxes items="${languages}" path="favouriteLanguage" />

6. Male: <form:radiobutton path="sex" value="M"/>
Female: <form:radiobutton path="sex" value="F"/>

7. <form:radiobuttons items="${jobItem}" path="job" />

8.
```
<form:select path="book">
  <form:option value="-" label="--Please Select--"/>
  <form:options items="${books}" />
</form:select>
```

9. <form:hidden path="id" value="12345" />

10. <form:errors path="name" cssClass="error" />

Spring Data JPA

In entire web application ,the DAO layer usually consists of a lot of boilerplate code that can be simplified.

Benefits of simplification
1. Decrease in the number of layers that we need to define and maintain
2. Consistency of data access patterns and consistency of configuration.

Spring Data JPA framework takes this simplification one step forward and makes it possible to remove the DAO implementations entirely. The interface of the DAO is now the only artifact that we need to explicitly define.

For this , a DAO interface needs to extend the JPA specific Repository interface – JpaRepository or its super i/f CrudRepository. This will enable Spring Data to find this interface and automatically create an implementation for it.

By extending the interface we get the most required CRUD methods for standard data access available in a standard DAO.

eg : CRUDRepository methods
long count()

Returns the number of entities available.
void delete(T entity)
Deletes a given entity.
void deleteAll()
Deletes all entities managed by the repository.
void deleteAll(Iterable<? extends T> entities)
Deletes the given entities.
void deleteById(ID id)
Deletes the entity with the given id.
boolean existsById(ID id)
Returns whether an entity with the given id exists.
Iterable<T> findAll()
Returns all instances of the type.
Iterable<T> findAllById(Iterable<ID> ids)
Returns all instances of the type with the given IDs.
Optional<T> findById(ID id)
Retrieves an entity by its id.
<S extends T>
S save(S entity)
Saves a given entity.
<S extends T>
Iterable<S> saveAll(Iterable<S> entities)
Saves all given entities.

Method of JpaRepository
void deleteAllInBatch()
Deletes all entities in a batch call.
void deleteInBatch(Iterable<T> entities)
Deletes the given entities in a batch which means it will create a single Query.
List<T> findAll()
<S extends T>
List<S> findAll(Example<S> example)
<S extends T>
List<S> findAll(Example<S> example, Sort sort)
List<T> findAll(Sort sort)
List<T> findAllById(Iterable<ID> ids)
void flush()
Flushes all pending changes to the database.

T getOne(ID id)

Returns a reference to the entity with the given identifier.

<S extends T>

List<S> saveAll(Iterable<S> entities)

3. Custom Access Method and Queries

By extending one of the Repository interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented.

To define more specific access methods, Spring JPA supports quite a few options:

1. simply define a new method in the interface

provide the actual JPQL query by using the @Query annotation

When Spring Data creates a new Repository implementation, it analyzes all the methods defined by the interfaces and tries to automatically generate queries from the method names. While this has some limitations, it's a very powerful and elegant way of defining new custom access methods with very little effort.

eg :

```
Customer findByName(String name);
List<Person> findByAddressAndLastname(String address, String lastname);

// Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);


// Enabling ignoring case for an individual property
List<Person> findByLastnameIgnoreCase(String lastname);


// Enabling static ORDER BY for a query
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

List<Person> findByAddressZipCode(String zipCode);
```

Assuming a Person p has an Address with a String zipCode. In that case, the method creates the property traversal p.address.zipCode.

Limiting the result size of a query with Top and First

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

2. Or in case of custom queries , can add directly in DAO i/f.

eg :

```
@Query("select u from User u where u.emailAddress = :em")
 User findByEmailAddress(@Param("em")String emailAddress);

@Query("SELECT p FROM Person p WHERE LOWER(p.name) = LOWER(:nm)")
Foo retrieveByName(@Param("nm") String name);
```

3. Transaction Configuration

The actual implementation of the Spring Data managed DAO is hidden since we don't work with it directly. It's implemented by – the SimpleJpaRepository – which defines default transaction mechanism using annotations.

These can be easily overridden manually per method.

4. Exception Translation is still supplied

Exception translation is still enabled by the use of the @Repository annotation internally applied on the DAO implementation class.

The relationship between Spring Data JPA, JPA, and Hibernate/EclipseLink/Kodo /JDBC

JPA is a part of Java EE/Jakarta EE specification that defines an API for ORM and for managing persistent objects. Hibernate and EclipseLink are popular implementations of this specification.

Spring Data JPA adds a layer on top of JPA. That means it uses all features defined by the JPA specification, especially the entity and association mappings, the entity lifecycle management, and JPA's query capabilities. On top of that, Spring Data JPA adds its own features like a no-code implementation of the repository pattern and the creation of database queries from method names.

If the JPA specification and its implementations provide most of the features that you use with Spring Data JPA, do you really need the additional layer? Can't you just use the Hibernate directly ?

You can, of course, do that. That's what a lot of Java applications do. Spring ORM provides a good integration for JPA (eg : Spring native Hibernate Integration or Spring JPA)

But the Spring Data team took the extra step to make your job a little bit easier. The additional layer on top of JPA enables them to integrate JPA into the Spring stack easily.

They also provide a lot of functionality that you otherwise would need to implement yourself.

Why Spring Data JPA

1. No-code Repositories

The repository pattern is one of the most popular persistence-related patterns. It hides the DB specific implementation details and enables you to implement your business logic with higher abstraction level.

eg : For Author Entity

How ?

to persist, update and remove one or multiple Author entities,

to find one or more Authors by their primary keys,

to count, get and remove all Authors and

to check if an Author with a given primary key exists.

All you need to do is :

```
public interface AuthorRepository extends JpaRepository<Author, Integer> {}
```

2. Reduced boilerplate code

To make it even easier, Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces. You don't need to implement these operations.

3. Generated queries

IACSD Advance Java

Another cool feature of Spring Data JPA is the generation of database queries based on method names.(finder method pattern)

eg : Write a method that gets a Book entity with a given title. Internally, Spring generates a JPQL query based on the method name, sets the provided method parameters as bind parameter values, executes the query and returns the result.

```
//JpaRepository<T,ID>
//T : entity type
//ID : Data type of id property(PK)
public interface BookRepository extends JpaRepository<Book, Integer> {

  Optional<Book> findByTitle(String title123);
}
```

Assumption : title : property of Book POJO

Using Spring Data JPA with Spring Boot

You only need to add the spring-boot-starter-data-jpa and your JDBC driver to your maven build. The Spring Boot Starter includes all required dependencies and activates the default configuration.

Add DB config properties in application.properties file

By default, Spring Boot expects that all repositories are located in a sub-packages of the class annotated with @SpringBootApplication.

If your application doesn't follow this default, you need to configure the packages of your repositories using an @EnableJpaRepositories annotation.

Repositories(API) in Spring Data JPA

package : o.s.data.repository

CrudRepository

PagingAndSortingRepository

JpaRepository

The CrudRepository interface defines a repository that offers standard create, read, update and delete operations.

The PagingAndSortingRepository extends the CrudRepository and adds findAll methods that enable you to sort the result and to retrieve it in a paginated way.

The JpaRepository adds JPA-specific methods, like flush() to trigger a flush on the persistence context or findAll(Example<S> example) to find entities

Defining an entity-specific repository

eg :

Book entity is a normal JPA entity with a generated primary key of type Long, a title and a many-to-many association to the Author entity.

```
@Entity
@Table(name="books")
public class Book {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Version
  private int version; //for optimistic locking

  private String title;

  @ManyToMany
  @JoinTable(name = "book_author",
        joinColumns = { @JoinColumn(name = "fk_book") },
        inverseJoinColumns = { @JoinColumn(name = "fk_author") })
  private Set<Author> authors = new HashSet<>();

  ...
}
```

If you want to define a JPA repository for this entity, you need to extend Spring Data JpaRepository interface and type it to Book and Long.

```
public interface BookRepository extends JpaRepository<Book, Long> {

  Book findByTitle(String title);
}
```

Working with Repositories

After you defined your repository interface, you can use the @Autowired annotation to inject it into your service implementation. Spring Data will then provide you with a proxy implementation of your

repository interface. This proxy provides default implementations for all methods defined in the interface.

For More API Details

refer " regarding Spring Data JPA"

Spring Boot

What is REST ?

REST stands for REpresentational State Transfer.

It's software architectural style to develop distributed applications.

REST is web standards based architecture and uses HTTP Protocol for data communication.

It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.

Resource = Any thing that you want to expose to clients (i.e outside world) through your application.

eg : In Student Management System : student , course , subject , faculty , admission

In eCommerce : product , customer , order , order items , payment

REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources.

Here each resource is identified by URIs

URI : Uniform resource identifier. It's used to identify the resource.

eg : refer to URIs.png

HTTP Methods

Following well known HTTP methods are commonly used in REST based architecture.

GET - Provides a read only access to a resource.

POST - Used to create a new resource.

DELETE - Used to remove a resource.

PUT - Used to update a existing resource or create a new resource.

PATCH -- Used to perform partial updates to the resource.

REST uses various representations to represent a resource like text, JSON and XML. Most popular light weight data exchange format used in web services = JSON

What is Restful Web Service?

REST is used to build Web services that are lightweight, maintainable, and scalable in nature. A service which is built on the REST architecture is called a RESTful service. The underlying protocol for REST is HTTP, which is the basic web protocol. REST stands for REpresentational State Transfer

The key elements of a RESTful implementation are as follows:

1. Resources The first key element is the resource itself. Let assume that a web application on a server has records of several employees. Let's assume the URL of the web application is http://www.server.com. Now in order to access an employee record resource via REST, one can issue the command http://www.server.com/employee/1 - This command tells the web server to please provide the details of the employee whose employee number is 1.

2. Request Verbs - These describe what you want to do with the resource. A browser issues a GET verb to instruct the endpoint it wants to get data. However, there are many other verbs available including things like POST, PUT, and DELETE. So in the case of the example http://www.server.com/employee/1 , the web browser is actually issuing a GET Verb because it wants to get the details of the employee record.

3. Request Headers These are additional instructions sent with the request. These might define the type of response required or the authorization details.

4. Request Body - Data is sent with the request. Data is normally sent in the request when a POST request is made to the REST web service. In a POST call, the client actually tells the web service that it wants to add a resource to the server. Hence, the request body would have the details of the resource which is required to be added to the server.

IACSD Advance Java

5. Response Body This is the main body of the response. So in our example, if we were to query the web server via the request http://www.server.com/employee/1 , the web server might return an XML document with all the details of the employee in the Response Body.

6. Response Status codes These codes are the general codes which are returned along with the response from the web server. An example is the code 200 which is normally returned if there is no error when returning a response to the client.

The six architectural constraints of REST APIs

1. Client-server architecture

An API's job is to connect two pieces of software without limiting their own functionalities. This objective is one of the core restrictions of REST: the client (that makes requests) and the server (that gives responses) stay separate and independent.

When done properly, the client and server can update and evolve in different directions without having an impact on the quality of their data exchange. This is especially important in various cases where there are plenty of different clients a server has to cater to. Think about weather APIs — they have to send data to tons of different clients (all types of mobile devices are good examples) from a single database.

2. Statelessness

For an API to be stateless, it has to handle calls independently of each other. Each API call has to contain the data and commands necessary to complete the desired action.

An example of a non-stateless API would be if, during a session, only the first call has to contain the API key, which is then stored server-side. The following API calls depend on that first one since it provides the client's credentials.

In the same case, a stateless API will ensure that each call contains the API key and the server expects to see proof of access each time.

Stateless APIs have the advantage that one bad or failed call doesn't derail the ones that follow.

3. Uniform Interface

While the client and the server change in different ways, it's important that the API can still facilitate communication. To that end, REST APIs impose a uniform interface that can easily accommodate all connected software.

In most cases, that interface is based on the HTTP protocols. Besides the fact that it sets rules as to how the clients and the server may interact, it also has the advantage of being widely known and used on the Internet. Data is stored and exchanged through JSON files because of their versatility.

4. Layered system

To keep the API easy to understand and scale, RESTful architecture dictates that the design is structured into layers that operate together.

With a clear hierarchy for these layers, executing a command means that each layer does its function and then sends the data to the next one. Connected layers communicate with each other, but not with every component of the program. This way, the overall security of the API is also improved.

If the scope of the API changes, layers can be added, modified, or taken out without compromising other components of the interface.

5. Cacheability

It's not uncommon for a stateless API's requests to have large overhead. In some cases, that's unavoidable, but for repeated requests that need the same data, caching said information can make a huge difference.

The concept is simple: the client has the option to locally store certain pieces of data for a predetermined period of time. When they make a request for that data, instead of the server sending it again, they use the stored version.

The result is simple: instead of the client sending several difficult or costly requests in a short span of time, they only have to do it once.

6. Code on Demand

Unlike the other constraints we talked about up to this point, the last one is optional. The reason for making "code on demand" optional is simple: it can be a large security risk.

The concept is to allow code or applets(now obsolete!) to be sent through the API and used for the application. As you can imagine, unknown code from a shady source could do some damage, so this constraint is best left for internal APIs where you have less to fear from hackers and people with bad intentions. Another drawback is that the code has to be in the appropriate programming language for the application, which isn't always the case.

The upside is that "code on demand" can help the client implement their own features on the go, with less work being necessary on the API or server. In essence, it permits the whole system to be much more scalable and agile.

RestController vs MVC Controller n Annotations

A key difference between a traditional MVC controller and the RESTful web service controller is the way

that the HTTP response body is created.Instead of relying on a view technology(JSP / Thymeleaf) to perform server-side rendering of the data to HTML, typically a RESTful web service controller simply populates and returns a java object. The object data will be written directly to the HTTP response as JSON/XML/Text

To do this, the @ResponseBody annotation on the ret type of the request handling method tells Spring MVC that it does not need to render the java object through a server-side view layer.

Instead it tells that the java object returned is the response body, and should be written out directly.

The java object must be converted to JSON. Thanks to Spring's HTTP message converter support, you don't need to do this conversion manually. Because Jackson Jar is on the classpath, SC can automatically convert the java object to JSON & vice versa (using 2 annotations @ReponseBody & @RequestBody)

API --Starting point
o.s.http.converter.HttpMessageConverter<T>
--Interface that specifies a converter that can convert from and to HTTP requests and responses.
T --type of request/response body.

Implementation classes
1. o.s.http.converter.xml.Jaxb2RootElementHttpMessageConverter
-- Implementation of HttpMessageConverter that can read and write XML using JAXB2.(Java architecture for XML binding)

2. o.s.http.converter.json.
MappingJackson2HttpMessageConverter
--Implementation of HttpMessageConverter that can read and write JSON using Jackson 2.x's ObjectMapper class API

Important Annotations
1. @ResponseBody

Applied at : return value of the request handling method , annotated with @RequestMapping or @GetMapping / @PostMapping / @PutMapping / @DeleteMapping

It is used to marshall(serialize) the return value into the HTTP response body. Spring comes with converters that convert the Java object into a format understandable for a client(text/xml/json)

eg :
```
@Controller
@RequestMapping("/employees")
public class EmpController
{
  @GetMapping(....)
  public @ResponseBody Emp fetchEmpDetails(int empId)
  {
    //get emp dtls from DB through layers
    return e;
```

```
  }
}
```

2. @RestController
Class level annotation

Good news is @RestController = @Controller(at the class level) + @ResponseBody added on ret types of ALL request handling methods

eg :
```
@RestController
@RequestMapping("/employees")
public class EmpController
{
  @GetMapping(....)
  public Emp fetchEmpDetails(int empId)
  {
    //get emp dtls from DB through layers
    return e;
  }.....
}
```

3. @PathVariable --- handles URI templates.(URI variables or path variables)
Where to apply : on the method argument
Purpose : to access a path variable

eg : URL -- http://host:port/products/100 , method=GET

Method of ProductController
```
@RestController
@RequestMapping("/products") {
@GetMapping("/{pid}")
public Product getDetails(@PathVariable(name="pid") int pid1234)
{...}
}
```
In the above URL , the path variable {pid} is mapped to an int . Therefore all of the URIs such as /products/1 or /products/10 will map to the same method in the controller.

4. The @RequestBody annotation, unmarshalls the HTTP request body into a Java object injected in the method.
Applied on the method argument of the reuqest handling methods , containing request body
eg : Typically in POST , PUT , PATCH

@RequestBody must be still added on a method argument of request handling method , for un marshaling(de serialization)

5. @CrossOrigin
Class/Method level annotation

What is CORS ?
Cross-Origin Resource Sharing (CORS)

CORS is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading of resources.

A cross-origin HTTP request is a request to a specific resource, which is located at a different origin, namely a domain, protocol and port, than the one of the client performing the request.

For security reasons, browsers can request several cross-origin resources, including images, CSS, JavaScript files etc. By default, a browser' security model will deny any cross-origin HTTP request performed by client-side scripts.

While this behavior is desired, to prevent different types of Ajax-based attacks, sometimes we need to instruct the browser to allow cross-origin HTTP requests from JavaScript clients with CORS.

eg : React client running on http://localhost:3000, and a Spring Boot RESTful web service API listening at http://host:port/products

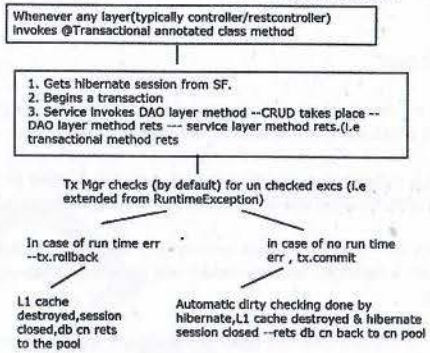
In such a case, the client should be able to consume the REST API, which by default would be forbidden.

To make this work , enable CORS by simply annotating the class / methods of the RESTful web service API responsible for handling client requests with the @CrossOrigin annotation

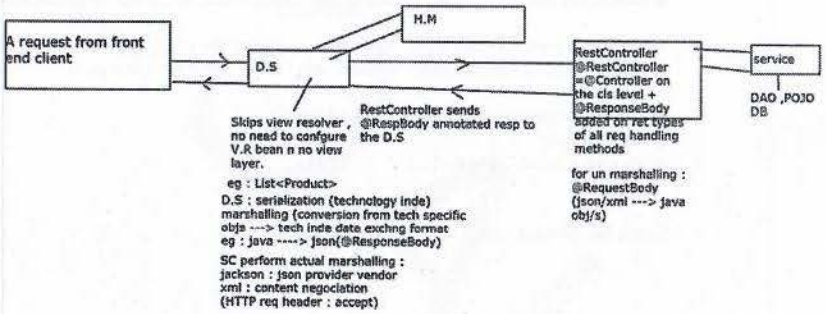
eg : @CrossOrigin(origins = "http://localhost:3000")
```
@RestController
public class ProductController{....}
```

Spring Txn. Management

How Spring supplied o.s.orm.hibernate5.HibernateTransactionManager manages Txs? OR in case of spring boot : JpaTransactionManager bean



Spring MVC Flow in case of REST API Based Backend



Spring Boot

1. What is Spring Boot?
Spring Boot is a Framework from "The Spring Team" to ease the bootstrapping and development of new Spring Applications.

It provides defaults for code and annotation configuration to quick start new Spring projects within no time.

It follows "Opinionated Defaults Configuration" Approach to avoid lot of boilerplate code and configuration to improve Development, Unit Test and Integration Test Process.

2. What is NOT Spring Boot?
Spring Boot Framework is not implemented from the scratch by The Spring Team
It's implemented on top of existing Spring Framework (Spring IO Platform).
It is not used for solving any new problems. It is used to solve same problems like Spring Framework.
(i.e to help in writing enterprise applications)

3. Advantages of Spring Boot:
It is very easy to develop Spring Based applications with Java
It reduces lots of development time and increases productivity.
It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
It follows "Opinionated Defaults Configuration" Approach to reduce Developer effort
It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.
It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
It provides lots of plugins to work with embedded and in-memory Databases very easily.

In short
Spring Boot = Spring Framework + Embedded HTTP Server(eg Tomcat) - XML Based configuration - efforts in identifying dependencies in pom.xml

4. What is that "Opinionated Defaults Configuration" ?

When we use Hibernate/JPA, we would need to configure a datasource, a session factory, a transaction manager among lot of other things.
Refer to our hibernate-persistence.xml

Spring Boot says can we look at it differently ?
Can we auto-configure a Data Source(connection pool) / session factory / Tx manager if Hibernate jar is on the classpath?

When a spring mvc jar is added into an application, can we auto configure some beans automatically?
(eg HandlerMapping , ViewResolver n configure DispatcherServlet)

By the way :
There would be of course provisions to override the default auto configuration.

5. How does it work ?
Spring Boot looks at
1. Frameworks available on the CLASSPATH
2. Existing configuration for the application.

Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks. This is called Auto Configuration.

6. What is Spring Boot Starter ?
Starters are a set of convenient dependency descriptors that you can include in your application's pom.xml
.
eg : Suppose you want to develop a web application.

W/o Spring boot , we would need to identify the frameworks we want to use, which versions of frameworks to use and how to connect them together.

BUT all web application have similar needs.
These include Spring MVC, Jackson Databind (for data binding), Hibernate-Validator (for server side validation using Java Validation API) and Log4j (for logging). Earlier while creating any web app, we had to choose the compatible versions of all these frameworks.

With Spring boot : You just add Spring Boot Starter Web.

Dependency for Spring Boot Starter Web

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

Just by adding above starter , it will add lot of JARs under maven dependencies

Another eg : If you want to use Spring and JPA for database access, just include the spring-boot-starter-data-jpa dependency in your project, and you are good to go.

7. Another cool feature of Spring boot is : we don't have to worry about deploying our applications to external container. It comes with an embedded servlet container.

8.Important components of a Spring Boot Application

Below is the starting point of a Spring Boot Application

```
@SpringBootApplication
public class HelloSpringBootApplication {

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

}
```

About : org.springframework.boot.SpringApplication
It's Class used to bootstrap and launch a Spring application from a Java main method.

By default class will perform the following steps to bootstrap the application
1. Create an ApplicationContext instance (representing SC)
2. Manages life cycle of spring beans

@SpringBootApplication - This is where all the spring boot magic happens.
It consists of following 3 annotations.

1. @SpringBootConfiguration
It tells spring boot that this class here can have several bean definitions. We can define various spring beans here and those beans will be available at run time .

2. @EnableAutoConfiguration
It tells spring boot to automatically configure the spring application based on the dependencies that it sees on the classpath.

eg:
If we have a MySql dependency in our pom.xml , Spring Boot will automatically create a data source,using the properties in application.properties file.

If we have spring web in pom.xml , then spring boot will automatically create the dispatcher servlet n other beans (HandlerMapping , ViewResolver)

All the xml, all the java based configuration is now gone.It all comes for free thanks to spring boot to enable auto configuration annotation.

3. @ComponentScan (equivalent to xml tag : context:component-scan)

So this tells us that spring boot to scan through the classes and see which all classes are marked with the stereotype annotations like @Component Or @Service @Repository and manage these spring beans . Default base-pkg is the pkg in which main class is defined.
Can be overridden by
eg :

```
@ComponentScan(basePackages = "com")
```

For scanning entities : (equivalent to packagesToScan)

```
@EntityScan(basePackages = "com.app.pojos")
```

Steps

1. File --New --Spring starter project -- add project name , group id ,artifact id ,pkg names , keep packaging as JAR for Spring MVC web application.

2. Add dependencies
web -- web
sql -- Spring Data JPA, MYSQL
Core -- DevTools
Lombok
validation

3. Copy the entries from supplied application.properties & edit DB configuration.

4.For Spring MVC (with JSP view layer demo) using spring boot project

Add following dependencies ONLY for Spring MVC with JSP as View Layer in pom.xml

```
<!-- Additional dependencies for Spring MVC -->
        <dependency>
                <groupId>org.apache.tomcat.embed</groupId>
                <artifactId>tomcat-embed-jasper</artifactId>
        </dependency>

        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>jstl</artifactId>
        </dependency>
```

5. Create under src/main/webapp : WEB-INF folder

6. Create `r n test it.

7. Port earlier spring MVC app , observe the problems.
& fix it.

Problem observed : ??????
Reason : Could not find org.hibernate.SessionFactory (since Spring boot DOES NOT support any native hibernate implementation directly)
Solution : Replace hibernate's native API (org.hibernate) by JPA

In DAO layer : replace native hibernate API by JPA
i.e instead of auto wiring SF in DAO layer : inject JPA' EntityManager directly in DAO.
How ?

```
@PersistenceContext
//OR can continue to use @AutoWired : spring supplied annotation
private EntityManager mgr;
//uses def persistence unit , created auto by spring boot using db setting added //in application.properties file , 1 per app / DB
```

Use directly EntityManager API (refer :)
OR
Unwrap hibernate Session from EntityManager

```
Session session = mgr.unwrap(Session.class);
```

Which one is preferred ? 1st soln.

8. Test Entire application.

Steps of Creating spring boot app

1. Create spring boot app : using spring boot starter project (choose packaging as JAR)
2. Use same spring boot starters as earlier.
Spring web , Mysql driver , Spring data JPA , Lombok , Spring Dev Tools,validation
3. NO additional dependencies for view layer(i.e no jstl n no tomcat-embeded jasper dependencies , in pom.xml

4. Copy application.properties from earlier spring boot project
(Do not add view resolver related properties)

5. Build the layers in bottoms up manner, for the following objectives.

Objective : Complete backend for Emp management front end

1. Get All Employees :
Later Use ResponseEntity , to wrap response body + response headers.
2. Add Emp Details :
3. Delete Emp Details
4. Get Emp details by id
5. Update Emp details

Spring Boot AOP
Aspect Oriented Programming(AOP)

WHY

Separating cross-cutting concerns(=repetitive tasks) from the business logic/ request handling /persistence

IMPORTANT

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other(eg : Controller separate from Service or DAO layers) and AOP helps you decouple cross-cutting concerns from the objects that they affect.(eg : transactional code or security related code can be kept separate from B.L or exception handling code from request handling method)
eg of ready made aspects provided by SC : tx management,security , exc handling

eg scenario --

Think of a situation Your manager tells you to do your normal development work(eg - write stock trading appln) + write down everything you do and how long it takes you.

A better situation would be you do your normal work, but another person observes what you're doing and records it and measures how long it took.

Even better would be if you were totally unaware of that other person and that other person was able to also observe and record , not just yours but any other peoples work and time.

That's separation of responsibilities. --- This is what spring offers you through AOP.

It is NOT an alternative to OOP BUT it complements OOP.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.
Aspects enable the modularization of concerns.(concern=task/responsibility) such as transaction management,logging,security --- that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP jargon)

Enables the modularization of cross cutting concerns(=task)
Eg : Logging,Security,Transaction management, Exception Handling

Similar in functionality to ---In EJB framework -- EJBObject
Struts 2 -- interceptors
Servlet -- filters.
RMI -- stubs
Hibernate --- proxy (hib frmwork -- lazy --- load or any--many associations --rets typically un-inited proxy/proxies)

AOP with Spring Framework

One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework.

Like DI, AOP supports loose coupling of application objects.

The functionalities that span multiple points of an application are called cross-cutting concerns.

With AOP, applicationwide concerns(common concerns-responsibilities or cross-cutting concerns like eg - declarative transactions , security,logging,monitoring,auditing,exception handling....)
are decoupled from the objects to which they are applied.

Its better for application objects(service layer/controller/rest controller/DAO) to focus on the business domain problems that they are designed for and leave certain ASPECTS to be handled by someone else.

Job of AOP framework is --- Separating these cross-cutting concerns(repeatative tasks) from the core business logic

AOP is like triggers in programming languages such as Perl, .NET.

Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

Key Terms of AOP

Advice : Action(=cross cutting concern) to take either before/after or around the method (B.L) execution.
eg: transactional logic(begin tx,commit,rollback,session closing)

Advice describes WHAT is to be done & WHEN it's to be done.

eg : logging. It is sure that each object will be using the logging framework to log the event happenings , by calling the log methods. So, each object will have its own code for logging. i.e the logging functionality requirement is spread across multiple objects (call this as Cross cutting Concern, since it cuts across multiple objects). Wont it be nice to have some mechanism to automatically execute the logging code, before executing the methods of several objects?

2. Join Point : Place in application WHERE advice should be applied.(i.e which B.L methods should be advised)
(Spring AOP, supports only method execution join point)

3. Pointcut : Collection of join points.
It is the expression used to define when a call to a method should be intercepted.
eg :
@Pointcut("execution (Vendor com.app.bank.*.*(double))")
advice logic{....}

4. Advisor Group of Advice and Pointcut into a single unit.

5. Aspect : class representing advisor(advice logic + point cut definition)-- @Asepct -- class level annotation.

6. Target : Application Object containing Business logic.(To which advice gets applied at specified join points) --supplied by Prog

7. Proxy : Object created after applying advice to the target object(created by SC dynamically by implementing typically service layer i/f) ---consists of cross cutting concern(repeatative jobs , eg : tx management,security, exc handling)

8.Weaving -- meshing(integration) cross cutting concern around B.L
(3 ways --- compile time, class loading time or spring supported --dynamic --method exec time or run time)

Examples of readymade aspects :
Transaction management & security.

Types of Advice --appear in Aspect class

@Before : This advice (cross cutting concern) logic gets Executed only before B.L method execution.
@AfterReturning Executes only after method returns in successful manner
@AfterThrowing - Executes only after method throws exception

@After -- Executes always after method execution(in case of success or failure)
@Around -- Most powerful, executes before & after.

Regarding pointcuts
Sometimes we have to use same Pointcut expression at multiple places, we can create an empty method with @Pointcut annotation and then use it as expression in advices.

eg of PointCut annotation syntax

```
@Before("execution (* com.app.bank.*.*(..))")

@Pointcut("execution (* com.app.bank.*.*(double))")

// point cut expression
@Pointcut("execution (* com.app.service.*.add*(..))")
 // point cut signature -- empty method .
 public void test() {
 }
```

eg of Applying point cut

```
1. @Before(value = "test()")
public void logBefore(JoinPoint p) {.....}

2.
@Pointcut("within(com.app.service.*)")
   public void allMethodsPointcut(){}

@Before("allMethodsPointcut()")
   public void allServiceMethodsAdvice(){...}

3.
@Before("execution(public void com.app.model..set*(*))")
 public void loggingAdvice(JoinPoint joinPoint){pre processing logic ....}

4.  //Advice arguments, will be applied to bean methods with single String argument
   @Before("args(name)")
   public void logStringArguments(String name){....}

5. //Pointcut to execute on all the methods of classes in a package
   @Pointcut("within(com.app.service.*)")
   public void allMethodsPointcut(){}

6.@Pointcut("execution(* com.core.app.service.*.*(..))") // expression
private void meth1() {} // signature

7.@Pointcut("execution(* com.app.core.Student.getName(..))")
private void test() {}
```

Steps in AOP Implementation

1. Create core java project.
2. Add AOP jars to runtime classpath.
3. Add aop namespace to spring config xml.
4. To Enable the use of the @AspectJ style of Spring AOP & automatic proxy generation, add <aop:aspectj-autoproxy/>
5. Create Business object class. (using stereotype anotations)
6. Create Aspect class, annotated with @Aspect & @Component
7. Define one or more point cuts as per requirement

Eg of Point cut definition.

```
@PointCut("execution (* com.aop.service.Account.add*(..))")
public void test() {}
OR
@Before("execution (* com.aop.service.Account.add*(..))")
public void logIt()
{
  //logging advice code
}
```

Use such point cut to define suitable type of advice.

Test the application.

execution --- exec of B.L method

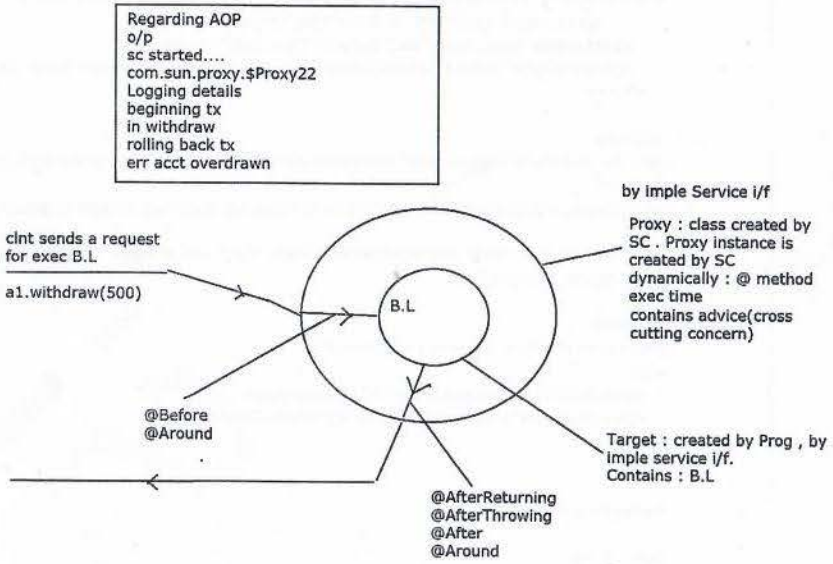
```
eg : @Before("execution (* com.app.bank.*.*(..))")
public void logIt()  {...}
```

Above tell SC ---- to intercept ---ANY B.L method ---
having ANY ret type, from ANY class from pkg -- com.app.bank
having ANY args
Before its execution.

Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type org.aspectj.lang.JoinPoint (In around advice this is replaced by ProceedingJoinPoint, which is a subclass of JoinPoint.)

The org.aspectj.lang.JoinPointJoinPoint interface methods
1. Object[] getArgs() -- returns the method arguments.
2. Object getThis() --returns the proxy object
3 Object getTarget() --returns the target object
4. Signature getSignature() -- returns a description of the method that is being advised
5. String toString() -- description of the method being advised



1. Regarding logging framework in Java

Spring Boot being extremely helpful framework , it allows us to forget about the majority of the configuration settings, many of which it opinionatedly auto-tunes.

In the case of logging, we don't have to explicitly import it's starter , since a starter, like our spring-boot-starter-web, depends on spring-boot-starter-logging, which already pulls in spring-jcl for us.

(Jakarta Commons Logging API (JCL) is the only mandatory external dependency for Spring!)

When using starters, Logback is used for logging by default.

eg : different logging levels

Logger logger = LoggerFactory.getLogger(LoggingController.class);
OR use Lombok annotation
@SLF4J : at the class level -- It will auto inject a Logger in the field : log

eg :
@RestController
public class LoggingController {


```
  Logger logger = LoggerFactory.getLogger(LoggingController.class);

  @RequestMapping("/")
  public String index() {
//it's in asc manner : logging levels
    logger.trace("A TRACE Message");
    logger.debug("A DEBUG Message");
    logger.info("An INFO Message");
    logger.warn("A WARN Message");
    logger.error("An ERROR Message");

    return "Testing logging here....";
  }
}
```

Default setting in application.properties file :
logging.level.root=INFO
OR can also be mentioned in : logback-spring.xml , under <resources> : classpath
eg : logging.level.org.springframework.orm.hibernate5=DEBUG
logging.level.com.app.service=DEBUG

For Swagger

swagger

```
<!-- Swagger UI -->
        <dependency>
                <groupId>org.springdoc</groupId>
                <artifactId>springdoc-openapi-ui</artifactId>
                <version>1.6.4</version>
        </dependency>
        <dependency>
                <groupId>org.springdoc</groupId>
                <artifactId>springdoc-openapi-ui</artifactId>
                <version>1.6.4</version>
        </dependency>
```

URL : http://localhost:8080/swagger-ui.html

Thymeleaf

Reference : https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html

What is Thymeleaf?

The Thymeleaf is an open-source Java library licensed under the Apache License 2.0.
It is a HTML5/XHTML/XML template engine.

It is a server-side Java template engine for web (servlet-based) environments. It provides full integration with Spring Framework.

It is based on XML tags and attributes. These XML tags define the execution of predefined logic on the DOM (Document Object Model) It is a substitute for JSP.

The architecture of Thymeleaf allows the fast processing of templates that depends on the caching of parsed files. It uses the least possible amount of I/O operations during execution.

Unlike JSP , it does not get compiled into a servlet.

1. Create a spring boot project , with additional dependency of thymeleaf
2. Default location of thymeleaf templates is <src>/main/resources/templates
Can also be added under subfolders or can be replaced by any other folder.

3. Create a new HTML under <templates>
index.html
4. Add XML namespace for thymeleaf.
<html xmlns:th="http://www.thymeleaf.org">

5. To display , model attributes
<h4 th:text="'some text' +${model attribute name} "></h4>

6. <a th:href="@{/order/list}">List Orders
If our app is installed at http://localhost:8080/myapp, after clicking on this link :
http://localhost:8080/myapp/order/list

7.
Switch case
```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
  <p th:case="*">Some other User role</p>
</div>
```

8. Links with dynamic attributes
For adding a query string , request param
eg : <a th:href="@{/students/edit(id=${student.id})}">Edit
Will produce : http://localhost:8080/students/edit?id=1

9. For adding a path variable :
<a th:href="@{/students/delete/{id}(id=${student.id})}">Delete
Will produce : http://localhost:8080/students/delete/1

10. Form Handling
eg :


```
<form action="#" th:action="@{/greeting}" th:object="${greeting}" method="post">
        <p>Id: <input type="text" th:field="*{id}" /></p>
    <p>Message: <input type="text" th:field="*{content}" /></p>
    <p><input type="submit" value="Submit" /> <input type="reset" value="Reset" /></p>
  </form>
```

Meaning :
10.1 The th:action="@{/greeting}" expression submits the form to POST to the http://host:port/greeting endpoint
10.2 th:object="${greeting}" binds form data to model attribute by the name of greeting.

10.3 The two form fields, expressed with th:field="*{id}" and th:field="*{content}", correspond to the properties of Greeting POJO

11. if-else
(there is no else here ,unless is a negation of if)
```
<td>
  <span th:if="${teacher.gender == 'F'}">Female</span>
  <span th:unless="${teacher.gender == 'F'}">Male</span>
</td>
```

Spring Security

Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework.
It is supplied as a "ready made aspect" , from spring security framework , that can be easily plugged in spring MVC application.

It is "THE" standard for securing Spring-based applications. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements.

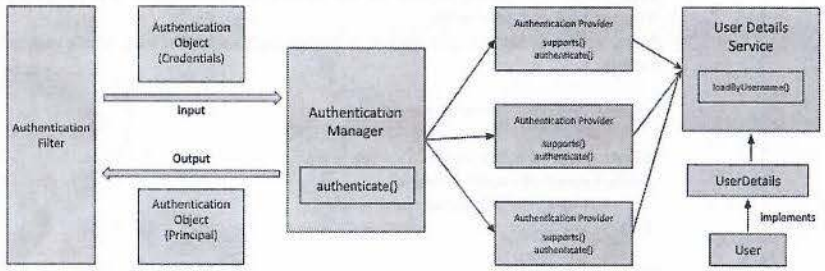
Features

1. Comprehensive and extensible support for both Authentication and Authorization
2. Protection against attacks like session fixation, clickjacking, cross site request forgery, etc..
3.Servlet API integration (Uses Servlet Filter chain)
4.Integration with Spring Web MVC.

Common Security Terms

Credentials : Way of confirming the identity of the user (email /username , password typically)
Principal: Currently logged in user.
Authentication: Confirming truth of credentials(i.e confirming who you are)

Authorisation: Defines access policy for the Principal.(i.e confirming your permissions, i.e what you can do)
GrantedAuthority: Permission granted to the principal.
AuthenticationManager (i/f): Controller in the authentication process. Authenticates user details via authenticate() method.



Steps

1. Create a new spring boot project (RESTful web service) , adding usual dependenices . DO NOT add spring security yet. Copy earlier working application.properties.

2. Add a TestController , with 3 end points
/home
/user
/admin
responding to GET method , with simple string response.

3. Test it .(using browser/postman)
Did it work ????

4. Add spring security dependency . Test the end points again.
Did it work ? NO

Observation : Suddenly n automatically all end points are now protected. So on browser it will prompt you to login form (spring security supplied) On postman it will give you HTTP 401 (Un authorized error)

We have not yet supplied any credentials .
Def credentials are : user n password(UUID : universally unique ID : 128 bit) from server console.

So w/o configuring anything , the moment spring security JARs are added , all your end points are secured automatically .

Thus Spring Boot(running on the top of the Spring Framework) , provides a ready made aspect(solution to cross cutting concern like authetication n authorization) in form of spring security

After supplying correct credentials(i.e after authentication) , spring security will redirect you to the resource : http://localhost:8080/home ,and you will be able to access it.
Supplies you automatically with a logout page (test it on the browser)

Observe on postman(w/o setting authorization header)
Response : (HTTP 401)

From authorization , choose Basic Authentication (referred as Basic Auth) ,
Add user name n password.

It will be encoded using base64 encoding.

Basic authentication, or "basic auth" is formally defined in the HTTP standard. When a client (your browser) connects to a web server, it sends a "WWW-Authenticate: Basic" message in the HTTP header. After that, it sends your login credentials to the server using a mild concealment technique called base64 encoding.

Not desirable , to use such credentials , so continue to next step.

5. Can you configure username n password , in a property file ?
YES .

Add these 2 properties in application.properties file
spring.security.user.name=
spring.security.user.password=

So now instead of spring security generated user name n pwd , these will be used for authentication.

6. Ultimate goal is using DB to store the authentication details .
BUT immediate next goal , to understand spring security is : Basic In memory authentication

The credentials will be stored in memory.
Comment earlier properties from app property file.

6.1 Add security config class, extending org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter
It's a convenient base class , to customize security configuration

6.2 Class level annotations
@EnableWebSecurity
@Configuration (annotation based approach equivalent to bean config xml file containing <bean id xml..../>)

6.3 Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
for supplying authentication details

6.4 Refer to diag : spring security architecture
Refer to readme : "spring sec auth flow"
Diagram : detailed flow.png

Add in memory authentication to the AuthenticationManagerBuilder , which will allow customization of the same.

API Methods
inMemoryAuthentication
withUser
password
roles
and

eg :
```
auth.inMemoryAuthentication().withUser("kiran").password(encoder().encode("1234")).roles("USER")
.and().withUser("rama").password(encoder().encode("3456")).roles("ADMIN");
```

6.5 For supplying authorization details :
Objective :
/home : accessible to all
/admin : only to admin user
/user : accessible to user n admin role

Override
protected void configure(HttpSecurity http) throws Exception

By extending from WebSecurityConfigurerAdapter and only a few lines of code we can do the following:
1. Require the user to be authenticated prior to accessing any URL within our application
2. Create a user with the username "user", password "password", and role of "USER"
3. Enables HTTP Basic and Form based authentication
4. Spring Security will automatically render a login page and logout success page for you

Refer to it's super class 's implementation n use it for overriding
Methods :
authorizeRequests()
antMatchers(String matchers...)
hasRole(String roleName) : no ROLE prefix
httpBasic()
formLogin

eg :
```
http.authorizeRequests().antMatchers("/admin").
        hasRole("ADMIN").
        antMatchers("/user").hasAnyRole("USER", "ADMIN")
```



```
        .antMatchers("/", "/home").permitAll()
        .and().httpBasic()
        .and().formLogin();
```

6.6 Run the application.
Problem : java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"

Reason -- Prior to Spring Security 5.0 the default PasswordEncoder was NoOpPasswordEncoder which required plain text passwords.
From Spring Security 5, the default is DelegatingPasswordEncoder, which requires Password Storage Format.

Solution : provide Password encoder bean

```
@Bean // equivalent to <bean id ...> tag in xml)
 public PasswordEncoder encoder() {
        return new BCryptPasswordEncoder();
 }
```

Test the application.

7. Replace in memory authentication by DB based authentication.
Using Spring Data JPA.

7.1 Edit application.properties file with DB settings.
Can optionally add these for debugging.
debug=true
logging.level.org.springframework.security=DEBUG

7.2 In Security config class
replace in memory authentication , by UserDetailsService based auth mgr builder
refer to diag : detailed flow.png

API of AuthenticationManagerBuilder

public DaoAuthenticationConfigurer userDetailsService(UserDetailsService service) throws Exception

Add authentication based upon the custom UserDetailsService that is passed in. It then returns a DaoAuthenticationConfigurer to allow customization of the authentication.

So auto wire UserDetailsService n use it. Set password encoder.

eg :
```
@Autowired
private UserDetailsService userDetailsService;
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //since there is no out-of-box imple for JPA based auth , u have to create a custom class imple
```

UserDetailsService n inject it here, n set password encoder
auth.userDetailsService(userDetailsService).passwordEncoder(encoder());

}

7.3
The org.springframework.security.core.userdetails.UserDetailsService interface is used to retrieve user-related data. It has one method named loadUserByUsername() which can be overridden to customize the process of finding the user.

It is used by the DaoAuthenticationProvider to load details about the user during authentication.
It is used throughout the framework as a user DAO and is the strategy used by the DaoAuthenticationProvider.

class DaoAuthenticationProvider :
Represents an AuthenticationProvider implementation that retrieves user details from a UserDetailsService.

Method
UserDetails loadUserByUsername(java.lang.String username)
throws UsernameNotFoundException

7.4 How to load user by user name ?

1. Create POJOs User n Role with many-many (UserEntity *---->* Role) EAGER or can later replace it by UserEntity *---->1 Role
UserEntity extending from BaseEntity
Properties : userName,email, password,active, roles : Set<Role>
Role : id , enum UserRole (ROLE_USER....)

2. DAO layer : UserRepository -- findByUserName
RoleRepository

3. Create custom implementation of org.springframework.security.core.userdetails.UserDetailsService n implement
UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException
In case , user entity not found , raise UsernameNotFoundException , with suitable error message.

4. In case of success , create custom implementation of, org.springframework.security.core.userdetails.UserDetails i/f, by passing to it's constructor , User entity details , lifted from DB
o.s.s.c.userdetails.UserDetails : represents core user information. It stores user information which is later encapsulated into Authentication object. This allows non-security related additional user information (eg : email acct expiry, user enabled ...) in addition to user name n password to be stored in a convenient location.

One important method in above i/f to implement is
public Collection<? extends GrantedAuthority> getAuthorities() , which should return , granted

authorities (role based) for the loaded user.
eg : user => UserEntity
user.getRoles().stream().map(role -> new SimpleGrantedAuthority(role.getUserRole().name()))
.collect(Collectors.toList());

5. Implement all other methods , suitably .

How to run ?
1. Write dao layer test case : to add 2 roles : ROLE_ADMIN n ROLE_USER

2. Write dao layer test case : to add 2 users , with admin n user role each.

3. For hashing the password :

use :
https://bcrypt-generator.com/

Project Tip :
Later to test it with React/Angular front end :
use below for authorization.

```
http.csrf().disable().
        cors().and().
        authorizeRequests().

        antMatchers(HttpMethod.OPTIONS, "/**").permitAll().
        antMatchers("/", "/home", "/api/signup").permitAll().
        antMatchers("/admin").hasRole("ADMIN").
        antMatchers("/user").hasAnyRole("USER", "ADMIN").
        and().httpBasic();
```

How does spring security works internally?

Spring security is enabled automatically , by just adding the spring security starter jar. But, what happens internally and how does it make our application secure?

Common Terms

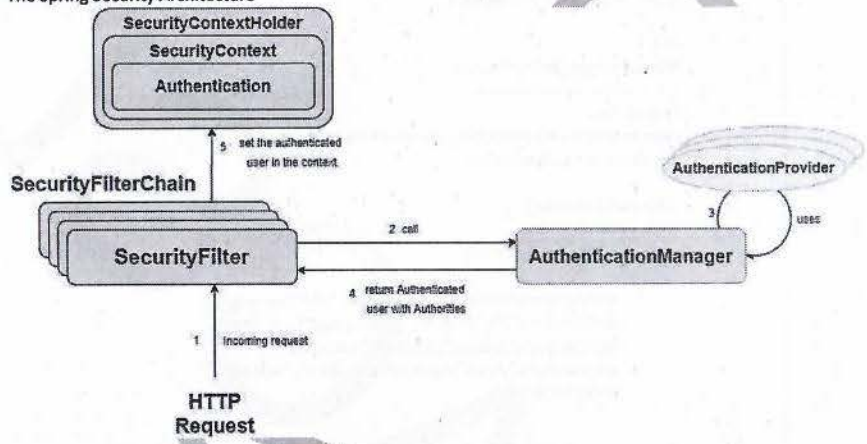
Principal: Currently logged in user.
Authentication: Confirming truth of credentials.
Authorisation: Defines access policy for the Principal.
GrantedAuthority: Permission granted to the principal.
AuthenticationManager (i/f): Controller in the authentication process. Authenticates user details via authenticate() method.

AuthenticationManager i/f implemented by : ProviderManager class .

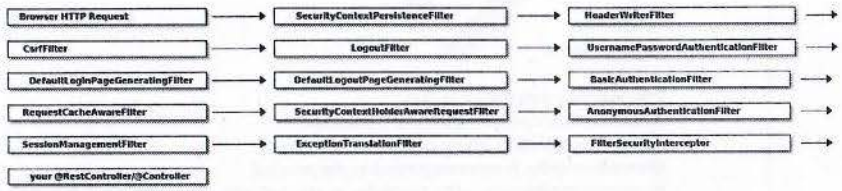
Diagram : detailed flow .png
It Iterates an Authentication request through a list of AuthenticationProviders.
AuthenticationProviders are usually tried in order until one provides a non-null response. A non-null response indicates the provider had authority to decide on the authentication request and no further providers are tried.

AuthenticationProvider: Interface that maps to a data store that stores your data.
Authentication Object: Object that is created upon authentication. It holds the login credentials. It is an internal spring security interface.
UserDetails: Data object that contains the user credentials but also the role of that user.
UserDetailsService: Collects the user credentials, authorities (roles) and build an UserDetails object.

The Spring Security Architecture



When we add the spring security starter jar, it internally adds Filter to the application. A Filter is an object that is invoked at pre-processing and post-processing of a request. It can manipulate a request or even can stop it from reaching a servlet. There are multiple filters in spring security out of which one is the Authentication Filter, which initiates the process of authentication.



Once the request passes through the authentication filter, the credentials of the user are stored in the Authentication object. Now, what actually is responsible for authentication is AuthenticationProvider

(Interface that has method authenticate()). A spring app can have multiple authentication providers, one may be using Dao based , JWT based , OAuth, LDAP ... To manage all of them, there is an AuthenticationManager.

The authentication manager finds the appropriate authentication provider by calling the supports() method of each authentication provider. The supports() method returns a boolean value. If true is returned, then the authentication manager calls its authenticate() method.

After the credentials are passed to the authentication provider, it looks for the existing user in the system by UserDetailsService. It returns a UserDetails instance which the authentication provider verifies and authenticates. If success, the Authentication object is returned with the Principal and Authorities otherwise AuthenticationException is thrown.

JWT-
JWT Details
Session-based Authentication & Token-based Authentication

For using any website, mobile app or desktop app , you need to create an account, then use it to login for accessing features of the app : Authentication.

So, how to authenticate a user?
Simple method used : Session-based Authentication.
(refer : session-based-authentication.png)

As per the diag , when a user logs into a website, the Server will create a new Session for that user and store it (in Memory or Database). Server also returns a SessionId for the Client to save it in Browser Cookie.

The Session on Server has an expiration time. After that time, this Session has expired and the user must re-login to create another Session.

If the user has logged in and the Session has not expired yet, the Cookie (including SessionId) always goes with all HTTP Request to Server. Server will compare this SessionId with stored Session to authenticate and return corresponding Response.

If it's been working fine , then why do we need Token-based Authentication?
The answer is we don't have the only consumer as a browser (end user : client a person), we may have different consumers of our services here.

So if you have a website which works well with thin client(browser client) & want to implement system for Mobile (Native Apps) and use the same Web app. You can't authenticate users who use Native App using Session-based Authentication because these native apps don't support cookies. Or if you are implementing a REST server , every REST request HAS TO be stateless , meaning you can't think of maintaining a Http Session in the back end. On the other hand , you can't expect your front end app , to send you the credentials (username / password) along with every request.

That's why Token-based Authentication was born.

With this method, the user login state is encoded into a JSON Web Token (JWT) by the Server and sent to the Client after successful authentication.

Instead of creating a Session, the Server generates a JWT from user login data and sends it to the Client. The Client saves the JWT and then onwards sends this JWT along with every request , typically in a header to the server. The Server will validate the JWT and return the Response.
(Giving access to the secured resources)

For storing JWT on Client side, it depends on the platform you use:

Browser: Local Storage
IOS: Keychain
Android: SharedPreferences

3 important parts of a JWT:

Header
Payload
Signature

1. Header
The Header answers the question: How will we calculate JWT?
It's a JSON object
eg :

```
{
  "typ": "JWT",
  "alg": "HS512"
}
```

– typ is 'type', indicates that Token type here is JWT.
– alg stands for 'algorithm' which is a hash algorithm for generating Token signature. Here HS256 is HMAC-SHA256/512 – the algorithm which uses Secret Key.

2. Payload
The Payload helps us to answer: What do we want to store in JWT?
This is a payload sample:

```
{
  "userId": "abcd123456",
  "username": "rama",
  "email": "rama@gmail.com",
  // standard fields
  "iss": "Issuer at developers.com",
  "iat": 1570238918,
  "exp": 1570238992
}
```

In the JSON object above, we store 3 user fields: userId, username, email. You can save any field you

want.

We also have some Standard Fields. They are optional.

iss (Issuer): who issues the JWT
iat (Issued at): time the JWT was issued at
exp (Expiration Time): JWT expiration time

3. Signature
This part is where we use the Hash Algorithm : HMAC-SHA256
Look at the code for getting the Signature below:
In Java code :
java.util.Base64 class offers encoding n decoding.
Base64 is a binary-to-text encoding scheme. It represents binary data in a printable ASCII string format by translating it into a radix-64 representation.

The basic encoder keeps things simple and encodes the input as is, without any line separation.
The output is mapped to a set of characters in A-Za-z0-9+/ character set, and the decoder rejects any character outside of this set.
eg : In Java

```
String originalInput = "test input";
String encodedString = Base64.getEncoder().encodeToString(originalInput.getBytes());
```

In Javascript

```
const data = Base64UrlEncode(header) + '.' + Base64UrlEncode(payload);
const hashedData = Hash(data, secret);
const signature = Base64UrlEncode(hashedData);
```

– First, it encodes Header and Payload, join them with a dot .
- Next, makes a hash of the data using Hash algorithm (defined at Header) with a secret string.
– Finally, encodes the hashing result to get Signature.
-
After having Header, Payload, Signature, combines them into JWT standard structure: header.payload.signature.

Does JWT secures our data ?
JWT does NOT secure your data

JWT does not hide, obscure, secure data at all. You can see that the process of generating JWT (Header, Payload, Signature) only encode & hash data, not encrypt data.

The purpose of JWT is to prove that the data is generated by an authentic source.

So, what if there is a Man-in-the-middle attack that can get JWT, then decode user information?
Yes, that is possible, so always make sure that your application has the HTTPS encryption.

How Server validates JWT from Client ?
For creating a JWT , we use a Secret string to create Signature. This Secret string is unique for every Application and must be stored securely in the server side.

When receiving JWT from Client, the Server get the Signature, verify that the Signature is correctly hashed by the same algorithm and Secret string as above. If it matches the Server's signature, the JWT is valid.

JWT Steps

Detailed steps

Refer : JWT Details n diagrams

0. Copy earlier project created for Basic Authentication

1. Add jjwt dependency in pom.xml
<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt</artifactId>
<version>0.9.1</version>
</dependency>

2. Add these properties in application.properties
#JWT properties
#JWT Secret key for signing n Verification , later can be encrypted using Jasypt
SECRET_KEY=mySecretKey1234
#JWT expiration timeout in msec : 24*3600*1000
EXP_TIMEOUT=86400000

3. Create JWTUtils class (can be supplied as readymade utility) : add @Component for enabling it's D.I in the controller later.
Methods in JWTUtils
1. generate JWT token

2. get user name/email from the token

3. validate token

4.To Intercept all incoming reugests (except those mentioned with permit All in sec config class), create a custom filter class : extending from o.s.web.filter.OncePerRequestFilter
Spring secutity will call it's doFilterInternal method , once per request
Steps
4.1 Extract JWT from the request header
Get Authorization header from request n check if it is not null n starting with "Bearer "
4.2 If it's present , extract n validate it (using JWT utils validation method : to check if it's not been tampered !)
4.3 In case of valid token ,
extract user name /email from the token
4.4 If user name is not null n if not already authenticated (i.e authentication info is not yer stored in sec context)
then extract UserDetails from UserDetailsService n create Authentication object(class :

UserNamePasswordAuthenticationToken)
4.5 Set authentication details to :
Implementation of AuthenticationDetailsSource which builds the details objectfrom an HttpServletRequest object, creating a WebAuthenticationDetails.
4.6 Save this authentication token in the sec ctx.

5. Configure JWT in security configuration class.
5.1 NO changes in the authentication .
5.2 Modify authorization rules
Summary :
1. enable cors n disable CSRF
http.cors().and().csrf().disable()
2. In addition to other patterns : allow any HTTP OPTIONS request (which is typically a pre flight request coming from react like front end . NOTE : it's not required for testing it with postman
antMatchers(HttpMethod.OPTIONS, "/**").permitAll()

3. Configure session management policy
To tell Spring Security NEVER create an HttpSession & DO NOT use HttpSession to obtain the SecurityContext

sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
4. Add custom jwt filter before a filter for processing an authentication based upon form submission.
http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

Typical example code :
```
http.cors().
        and().
        csrf().disable().
        authorizeRequests().
        antMatchers("/user").hasAnyRole("USER", "ADMIN")
        .antMatchers("/admin").hasRole("ADMIN").
        antMatchers("/home").permitAll().
        antMatchers(HttpMethod.OPTIONS).permitAll(). // for react / angular clnts
        antMatchers("/home", "/api/auth/**").permitAll()
        .and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).
        and().
        // adding custom jwt filter before a filter : UsernamePasswordAuthenticationFilter --which processes form submission
        // form submission.
        addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);
```

6. Before implementing our authentication API end point , we need access to the authentication manager. By default, it's not publicly accessible, and we need to explicitly expose it as a bean in our configuration class.
```
@Bean
public AuthenticationManager manager() throws Exception
{
```

return super.authenticationManagerBean();
}

7. Create AuthController : for signin n signup(later)
request payload : Auth req DTO : email n password
resp payload : In case of success : Auth Resp DTO : mesg + JWT token + SC 200
IN case of failure : SC 401

Dependencies : Auth Mgr n JWTUtils

7.1
Create UserNamePasswordAuth token to wrap ONLY user supplied credentials(em n pass) , no authorities n authenticated=false
API : UsernamePasswordAuthenticationToken(String userName,String password)

7.2 Invoke Auth mgr's authenticate method , by passing above auth token
It rets : fully populated Authentication object (including granted authorities)if successful.
In case of failure :
Throws :
DisabledException : if an account is disabled
LockedException : if an account is locked
BadCredentialsException : if incorrect credentials are presented by client

7.3 In case of success : return
ResponseEntity : SC OK
Resp DTO : Auth successful mesg & generate JWT using utils n add it .

----------More Details----------
For JWT details : https://jwt.io/introduction
Refer to debugger n introduction sections

For spring security javadocs : https://docs.spring.io/spring-security/site/docs/current/api/

3.1 API : org.springframework.security.core.Authentication i/f
Methods : getAuthorities , getPrincipal , getCredentials ....

3.2 Authentication i/f method
java.lang.Object getPrincipal()
The identity of the principal being authenticated. In the case of an authentication request with username and password, this would be the username.
Since we have added these user details in the customized implementation of o.s.s.c.userdetails.UserDetails , you can type cast it to CustomUserDetailsImpl.

3.3 o.s.s.c.userdetails.UserDetails : i/f
Provides core user information.
You have to implement it to store user information which is later encapsulated into Authentication object.

3.4 org.springframework.security.authentication.AuthenticationManager
Implementing Classes: ProviderManager
Method :
Authentication authenticate(Authentication authentication) throws AuthenticationException
Attempts to authenticate the passed Authentication object, returning a fully populated Authentication object (including granted authorities) if successful.
An AuthenticationManager must honour the following contract concerning exceptions:

A DisabledException must be thrown if an account is disabled and the AuthenticationManager can test for this state.
A LockedException must be thrown if an account is locked and the AuthenticationManager can test for account locking.
A BadCredentialsException must be thrown if incorrect credentials are presented. While the above exceptions are optional, an AuthenticationManager must always test credentials.

Exceptions should be tested for and if applicable thrown in the order expressed above (i.e. if an account is disabled or locked, the authentication request is immediately rejected and the credentials testing process is not performed). This prevents credentials being tested against disabled or locked accounts.
Parameters:
authentication - the authentication request object(credentials)
Returns:
a fully authenticated object including credentials (principal)
Throws:
AuthenticationException - if authentication fails

3.5
org.springframework.security.authentication.UsernamePasswordAuthenticationToken : class

An implementation class of Authentication i/f which is designed for simple presentation of a username and password.
Simply said it's the holder of username n password (i.e user credentials)

Look at generateToken method

```
public String generateJwtToken(Authentication authentication) {
        System.out.println("generate jwt token " + authentication);
        UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication.getPrincipal();
//JWT : userName,issued at ,exp date,digital signature(does not contain password n authorities)
        return Jwts.builder() // JWTs : a Factory class , used to create JWT tokens
        .setSubject((userPrincipal.getUsername())) // setting subject of the token(typically user name)
:sets

// subject claim part of the token
        .setIssuedAt(new Date())// Sets the JWT Claims iat (issued at) value of current date
        .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))// Sets the JWT Claims exp
```

```
// (expiration) value.
        .signWith(SignatureAlgorithm.HS512, jwtSecret) // Signs the constructed JWT using the
specified

// algorithm with the specified key, producing a

// JWS(Json web signature=signed JWT)

// Using token signing algo : HMAC using SHA-512
        .compact();// Actually builds the JWT and serializes it to a compact, URL-safe string
}
```

3. Create UserController for authentication

Description : For authentication endpoint : /api/signin
Accept user credentials(request body from postMapping) , validate it , in case of success --create JWT response , using JWTUtils n send it as a response to the clnt.

autowire 2 dependencies : AuthenticationManager n JWTUtils
Oveeride authenticationManager method , in SecurityConfig class , annotatte it with @Bean , so that it will be avaliable for D.I

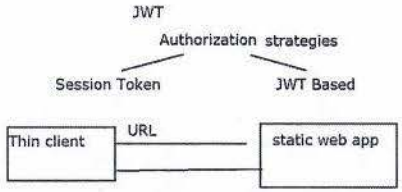
```
// add end point for user authentication
 @PostMapping("/signin")
 public ResponseEntity<?> authenticateUser(@RequestBody AuthenticationRequest request) {
        System.out.println("in auth " + request);
        try {
                // Tries to authenticate the passed Authentication object, returning a fully
                // populated Authentication object (including granted authorities)if successful.
                Authentication authenticate = authManager.authenticate
                // An o.s.s.c.Authentication i/f implementation used for simple presentation of  a
username and password.
                // Actual dao based authentication takes place here internally(first email : here replaced
username by email for authentication

                // n then pwd n then authorities gets validated)
                (new                    UsernamePasswordAuthenticationToken(request.getUserName(),
request.getPassword()));
                // => successful authentication : create JWT n send it to the clnt in the
                // response.
                System.out.println("auth success " + authenticate);
                return                                                  ResponseEntity.ok(new
AuthenticationResponse(jwtUtils.generateJwtToken(authenticate)));
        } catch (Exception e) {
                e.printStackTrace();
                throw new RuntimeException("User authentication Failed", e);
        }
```


Step 2 : Intercept all incoming reuqests (except those mentioned with permit All in sec config class), extract JWT from authorization header --validate it --set it in exec context(SecurtiyContextHolder)
To intercept all the requests --u need to create a filter
Spring sec already has configured lots of sec filters --u just extend from it n override method org.springframework.web.filter.OncePerRequestFilter
Details : User registration part : NO change ! (refer to spring_security4) for this.

Flow :
JwtRequestFilter:doFilterInternal --->request.getHeader("Authorization"); --rets null (since no bearer token sent from clnt still) --> continues with filter chain -->finally comes to the end point of UserSignupSignInController:authenticateUser -->
Authentication authenticate = authManager.authenticate(new UsernamePasswordAuthenticationToken(request.getUserName(), request.getPassword())); --->

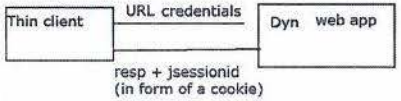
eg :

```
@PostMapping("/signin")
 public ResponseEntity<?> authenticateUser(@RequestBody AuthenticationRequest request) {
        System.out.println("in auth " + request);
        try {
                // Tries to authenticate the passed Authentication object, returning a fully
                // populated Authentication object (including granted authorities)if successful.
                Authentication authenticate = authManager.authenticate
                // An o.s.s.c.Authentication i/f implementation used for simple presentation of
                // a username and password.
                //Actual dao based authentication takes place here internally(first username n then pwd
is validated)
                (new                    UsernamePasswordAuthenticationToken(request.getUserName(),
request.getPassword()));
                // => successful authentication : create JWT n send it to the clnt in the
                // response.
                System.out.println("auth success "+authenticate);
                return                                                  ResponseEntity.ok(new
AuthenticationResponse(jwtUtils.generateJwtToken(authenticate)));
        } catch (Exception e) {
                e.printStackTrace();
                throw new RuntimeException("User authentication Failed", e);
        }
```



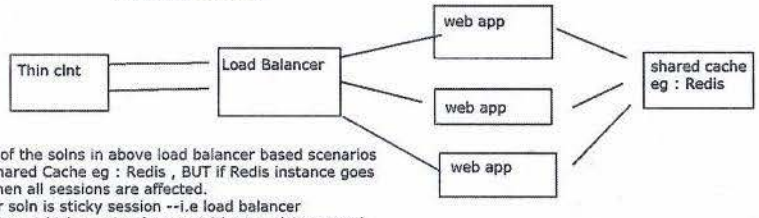
Most popular approach : has been session based authentication.

In case of static web app : HTTP as stateless protocol works perfectly , since in every req it's just the URL of the resource. BUT in case of dyn web app , you also need to send identity of who you are(who is accessig the web app)



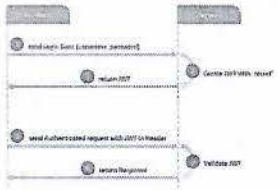
resp + jsessionid
(in form of a cookie)

This was a fine approach in a monolithic server side app. BUT in the modern world of microservices today , u have multiple web servers sitting behind a load balancer.



So one of the solns in above load balancer based scenarios is : a shared Cache eg : Redis , BUT if Redis instance goes down then all sessions are affected.
Another soln is sticky session --i.e load balancer remembers which session (represeniting user) is created on which web server n always sends the request to the same server--this soln is not scalable. Especially in microservices when one web app calls another.

JWT based authentication



Nothing is stored on the server.

Testing in Spring
Unit Testing of Spring MVC Controllers
Unit Testing of Spring Service Layer
Integration Testing of Spring MVC Applications: REST API
Unit Testing Spring MVC Controllers with REST
Securing Web Application with Spring Security
What is Spring Security
Spring Security with Spring Boot
Basic Authentication
Authentication with User credentials from Database and Authorization
JWT Authorization