Explore More

Subcription : Premium CDAC NOTES & MATERIAL @99

Contact to Join
Premium Group

Click to Join
Telegram Group

For More E-Notes

Join Our Community to stay Updated

TAP ON THE ICONS TO JOIN!

codewitharrays.in freelance project available to buy contact on 8007592194	
SR.NO **Project NAME**	**Technology**
1 Online E-Learning Platform Hub	React+Springboot+MySql
2 PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3 Tour and Travel management System	React+Springboot+MySql
4 Election commition of India (online Voting System)	React+Springboot+MySql
5 HomeRental Booking System	React+Springboot+MySql
6 Event Management System	React+Springboot+MySql
7 Hotel Management System	React+Springboot+MySql
8 Agriculture web Project	React+Springboot+MySql
9 AirLine Reservation System / Flight booking System	React+Springboot+MySql
10 E-commerce web Project	React+Springboot+MySql
11 Hospital Management System	React+Springboot+MySql
12 E-RTO Driving licence portal	React+Springboot+MySql
13 Transpotation Services portal	React+Springboot+MySql
14 Courier Services Portal / Courier Management System	React+Springboot+MySql
15 Online Food Delivery Portal	React+Springboot+MySql
16 Muncipal Corporation Management	React+Springboot+MySql
17 Gym Management System	React+Springboot+MySql
18 Bike/Car ental System Portal	React+Springboot+MySql
19 CharityDonation web project	React+Springboot+MySql
20 Movie Booking System	React+Springboot+MySql

freelance_Project available to buy contact on 8007592194

No.	Project	Technology
21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springwboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+NextJs+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql

41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48	Train Ticket Booking Project	React+Springboot+MySql
49	Quizz Application Project	JSP+Springboot+MySql
50	Hotel Room Booking Project	React+Springboot+MySql

51	Online Crime Reporting Portal Project	React+Springboot+MySql
52	Online Child Adoption Portal Project	React+Springboot+MySql
53	online Pizza Delivery System Project	React+Springboot+MySql
54	Online Social Complaint Portal Project	React+Springboot+MySql
55	Electric Vehical management system Project	React+Springboot+MySql
56	Online mess / Tiffin management System Project	React+Springboot+MySql
57		React+Springboot+MySql
58		React+Springboot+MySql
59		React+Springboot+MySql
60		React+Springboot+MySql

Spring Boot + React JS + MySQL Project List

| Sr.No	Project Name	YouTube Link
1	Online E-Learning Hub Platform Project	https://youtu.be/KMjyBaWmgzg?si=YckHuNzs7eC84-IW
2	PG Mate / Room sharing/Flat sharing	https://youtu.be/4P9cIHg3wvk?si=4uEsi0962CG6Xodp
3	Tour and Travel System Project Version 1.0	https://youtu.be/-UHOBywHaP8?si=KHHfE_A0uv725f12
4	Marriage Hall Booking	https://youtu.be/VXz0kZQi5to?si=llOS-QG3TpAFP5k7
5	Ecommerce Shopping project	https://youtu.be/vJ_C6LkhrZ0?si=YhcBylSErvdn7paq
6	Bike Rental System Project	https://youtu.be/FIzsAmIBCbk?si=7ujQTJqEgkQ8ju2H
7	Multi-Restaurant management system	https://youtu.be/pvV-pM2Jf3s?si=PgvnT-yFc8ktrDxB
8	Hospital management system Project	https://youtu.be/IynIouBZvY4?si=CXzQs3BsRkjKhZCw
9	Municipal Corporation system Project	https://youtu.be/cVMx9NVyI4I?si=qX0oQt-GT-LR_5jF
10	Tour and Travel System Project version 2.0	https://youtu.be/_4u0mB9mHXE?si=gDiAhKBowi2gNUKZ

Sr.No	Project Name	YouTube Link
11	Tour and Travel System Project version 3.0	https://youtu.be/Dm7nOdpasWg?si=P_Lh2gcOFhlyudug
12	Gym Management system Project	https://youtu.be/J8_7Zrkg7ag?si=LcxV51ynfUB7OptX
13	Online Driving License system Project	https://youtu.be/3yRzsMs8TLE?si=JRI_z4FDx4Gmt7fn
14	Online Flight Booking system Project	https://youtu.be/m755rOwdk8U?si=HURvAY2VnizIyJlh
15	Employee management system project	https://youtu.be/ID1iE3W_GRw?si=Y_jv1xV_BljhrD0H
16	Online student school or college portal	https://youtu.be/4A25aEKfei0?si=RoVgZtxMk9TPdQvD
17	Online movie booking system project	https://youtu.be/Lfjv_U74SC4?si=fiDvrhhrjb4KSlSm
18	Online Pizza Delivery system project	https://youtu.be/Tp3izreZ458?si=8eWAOzA8SVdNwlyM
19	Online Crime Reporting system Project	https://youtu.be/0UlzReSk9tQ?si=6vN0e70TVY1GOwPO
20	Online Children Adoption Project	https://youtu.be/3T5HC2HKyT4?si=bntP78niYH802I7N

CCEE Adj Java by Nayan Khade

LinkedIn

J2EE

- J2EE stands for Java 2 Platform, Enterprise Edition. It's a standard platform for developing applications for enterprises that run on servers. J2EE is a specification for a collection of software components that enable the development of multi-tiered web-based applications.

- J2EE programs are stand-alone and run on web servers. They help in developing web applications by using many types of frameworks like Hibernate and Spring. J2EE uses Enterprise Java Beans (EJBs), servlets, and Java Server Pages (JSPs).
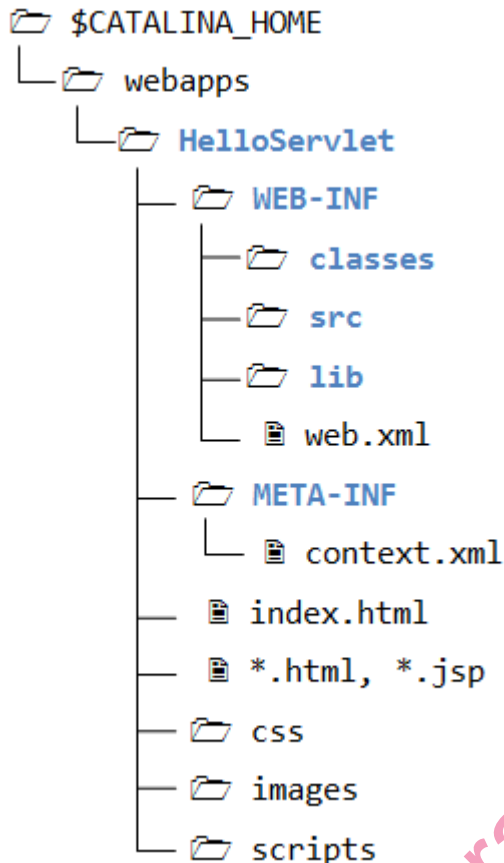
www.educba.com
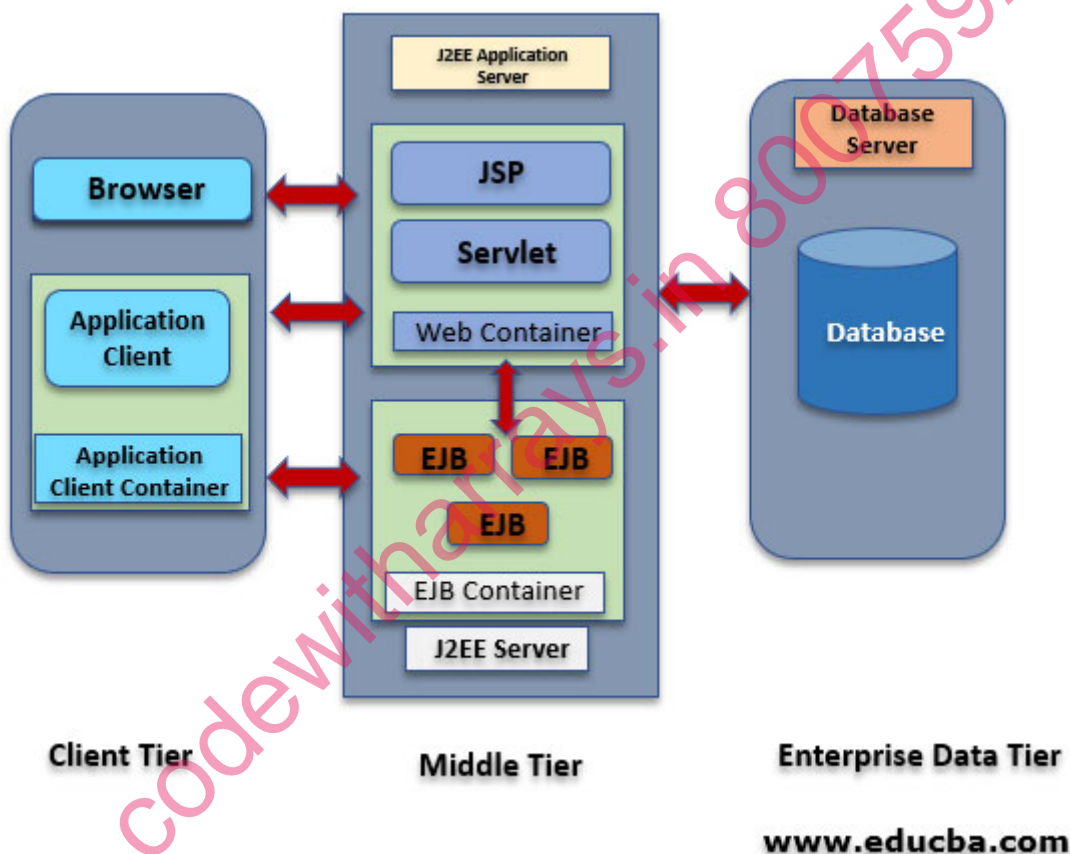
J2EE container

- is a runtime environment that provides services to the components (such as Servlets, JSP, and EJB) deployed within it. Containers manage the lifecycle, configuration, security, and other aspects of these components, allowing developers to focus on business logic rather than low-level infrastructure details.

1. JavaServer Pages (JSP): A technology that uses a paradigm where Java code is embedded into the HTML. Pages are written as HTML files with embedded Java source code known as scriptlets.
2. Servlets: A Java programming language class that is used to extend the capabilities of servers that host applications.

3. Enterprise JavaBeans (EJBs): Server-side Java components that implement a business task or entity.

Packaging Web applications

- Application packaging is the process of preparing software applications for distribution and deployment. It involves creating a package of files for each type of software that a company uses.

```
📂 $CATALINA_HOME
└── 📂 webapps
    └── 📂 HelloServlet
        ├── 📂 WEB-INF
        │   ├── 📂 classes
        │   ├── 📂 src
        │   ├── 📂 lib
        │   └── 📄 web.xml
        ├── 📂 META-INF
        │   └── 📄 context.xml
        ├── 📄 index.html
        ├── 📄 *.html, *.jsp
        ├── 📂 css
        ├── 📂 images
        └── 📂 scripts
```

Web Services Support

1. **JAX-RPC and JAX-WS:** APIs for XML-based RPC and web services.
2. **Annotations:** Simplify development through annotations.
3. **WSDL Support:** Tools and APIs for WSDL file generation and usage.
4. **SOAP Protocol:** Messaging protocol for service communication.
5. **Integration:** Seamless integration with Servlets and EJB.
6. **JAX-RS:** Simplifies RESTful web services development.
7. **ESB Integration:** Integration with Enterprise Service Buses for advanced messaging and orchestration.

Servlets

1. Servlet is a technology which is used to create a web application.
2. Servlet is an API that provides many interfaces and classes including documentation.
3. Servlet is an interface that must be implemented for creating any Servlet.
4. Servlet is a class that extends the capabilities of the servers and responds to the incoming requests.
5. It can respond to any requests.
6. Servlet is a web component that is deployed on the server to create a dynamic web page.

CGI (Common Gateway Interface)

- CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

- Advantages of Servlets over CGI :

1. Better performance: because it creates a thread for each request, not process.
2. Portability: because it uses Java language.
3. Robust: JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. Secure: because it uses java language.

packages :

- **javax.servlet** : These are not specific to any protocol.
- **javax.servlet.http** : These are specific to any protocol.

Interfaces in javax.servlet package :

1. **Servlet**: A Java class that extends the capabilities of a server to generate dynamic content and handle client requests. Servlets are a key component of Java-based web development. The servlet instance is created only once in the servlet life cycle.

2. **ServletRequest**: An interface representing the client's request to the server. It provides information about the request, such as parameters, headers, and input data.

3. **ServletResponse**: An interface representing the response sent by the server to the client. It allows a servlet to set response headers and write content to be sent back to the client.

4. **RequestDispatcher**: An interface that provides a way for a servlet to forward a request to another resource (servlet, JSP, or HTML file) or include the content of another resource in the response.

5. **ServletConfig**: An object that provides configuration information to a servlet. It allows a servlet to retrieve initialization parameters specified in the web deployment descriptor (web.xml).

6. **ServletContext**: An object that represents the servlet context, providing a way for servlets to communicate with the servlet container. It can be used to store and retrieve attributes, resources, and manage the lifecycle of servlets.

7. **SingleThreadModel**: An interface that a servlet can implement to indicate that it should handle only one request at a time. This was an older approach to dealing with thread safety but is now deprecated in favor of other synchronization mechanisms.

8. **Filter**: A component that performs filtering tasks on either the request to a resource, the response from a resource, or both. Filters are configured in the web deployment descriptor and can be used for tasks like authentication, logging, or data modification.

9. **FilterConfig**: An object that provides configuration information to a filter. It allows a filter to retrieve initialization parameters specified in the web deployment descriptor (web.xml).

10. **FilterChain**: An object that represents the chain of filters to be executed for a particular request. Filters can be chained together to perform multiple filtering tasks in sequence.

11. **ServletRequestListener**: An interface for receiving notifications about changes to servlet requests' lifecycle events. Implementing this interface allows the application to perform additional tasks based on request-related events.

12. **ServletRequestAttributeListener**: An interface for receiving notifications about changes to attributes in a servlet request's scope. Implementing this interface allows the application to perform tasks based on attribute-related events in the request.

13. **ServletContextListener**: An interface for receiving notifications about changes to the servlet context's lifecycle events. Implementing this interface allows the application to perform additional tasks based on context-related events.

14. **ServletContextAttributeListener**: An interface for receiving notifications about changes to attributes in a servlet context's scope. Implementing this interface allows the application to perform tasks based on attribute-related events in the context.

Classes in javax.servlet package :

1. **GenericServlet**: An abstract class that provides a generic, protocol-independent implementation of the Servlet interface. It is often extended to create servlets that handle specific protocols. GenericServlet class implements Servlet, ServletConfig and Serializable interfaces.

2. **ServletInputStream**: A stream that allows a servlet to read data from a client's request. It provides an input stream for reading binary data from the client.

3. **ServletOutputStream**: A stream that allows a servlet to send binary data to the client as part of the response. It provides an output stream for writing binary data.

4. **ServletRequestWrapper**: A class that extends the functionality of the ServletRequest interface by delegating its method calls to another ServletRequest instance. It is used for customization and extension of request handling.

5. **ServletResponseWrapper**: Similar to ServletRequestWrapper, this class extends the functionality of the ServletResponse interface by delegating its method calls to another ServletResponse instance. It is used for customization and extension of response handling.

6. **ServletRequestEvent**: An event class that represents events that occur within a servlet request's lifecycle. It can be used to listen for request-related events in a servlet container.

7. **ServletContextEvent**: An event class that represents events that occur within a servlet context's lifecycle. It can be used to listen for context-related events in a servlet container.

8. **ServletRequestAttributeEvent**: An event class that represents events related to attributes in a servlet request's scope. It is used to listen for attribute-related changes in the request.

9. **ServletContextAttributeEvent**: An event class that represents events related to attributes in a servlet context's scope. It is used to listen for attribute-related changes in the context.

10. **ServletException**: A generic exception class that servlets can throw when they encounter difficulty. It is a subclass of java.lang.Exception and is commonly used to indicate errors during servlet processing.

11. **UnavailableException**: A subclass of ServletException, indicating that a servlet is temporarily or permanently unavailable. It can be thrown to inform the servlet container that the servlet cannot handle requests for a specific period or permanently.

javax.servlet.http interfaces

1. **HttpServletRequest**: Represents an HTTP request made by a client to the server. It provides methods to retrieve information about the request, such as parameters, headers, and the request URI.

2. **HttpServletResponse**: Represents an HTTP response that the server sends back to the client. It allows a servlet to set response headers, set the content type, and send data back to the client.

3. **HttpSession**: Provides a way to identify a user across multiple requests and to store user-specific information on the server. It is used for session management in web applications.

4. **HttpSessionListener**: An interface for receiving notifications about changes to the session's lifecycle events. Implementing this interface allows the application to perform additional tasks based on session-related events.

5. **HttpSessionAttributeListener**: An interface for receiving notifications about changes to attributes in an HTTP session's scope. Implementing this interface allows the application to perform tasks based on attribute-related events in the session.

6. **HttpSessionBindingListener**: An interface for receiving notifications when an object is bound or unbound to an HTTP session. Implementing this interface allows the object to be notified of changes in its binding status.

7. **HttpSessionActivationListener**: An interface for receiving notifications about the activation or passivation of an HTTP session. Implementing this interface allows the application to perform tasks based on session activation or passivation events.

8. **HttpSessionContext** (deprecated): An interface that was used to manage a collection of sessions in a servlet container. However, it has been deprecated in favor of more advanced session management mechanisms, and developers are encouraged to use the HttpSession interface instead.

classes in the javax.servlet.http :

1. **HttpServlet**: An abstract class that provides a convenient base class for creating HTTP servlets. Servlets that extend HttpServlet can handle HTTP requests and generate responses, and they typically override methods like doGet and doPost to implement specific functionalities. The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

2. **Cookie**: Represents an HTTP cookie, which is a small piece of data sent from a web server and stored on the user's device. Cookies are commonly used for session management and tracking user preferences.

3. **HttpServletRequestWrapper**: A class that extends the functionality of the HttpServletRequest interface by delegating its method calls to another HttpServletRequest instance. It is used for customization and extension of request handling.

4. **HttpServletResponseWrapper**: Similar to HttpServletRequestWrapper, this class extends the functionality of the HttpServletResponse interface by delegating its method calls to another HttpServletResponse instance. It is used for customization and extension of response handling.

5. **HttpSessionEvent**: An event class that represents events related to HTTP session lifecycle changes. It can be used to listen for session-related events in a servlet container.

6. **HttpSessionBindingEvent**: An event class that represents events related to binding and unbinding objects to/from an HTTP session. It is used to listen for events related to objects being added or removed from the session.

7. **HttpUtils** (deprecated): A utility class that provided methods for encoding URLs and parsing query strings. It has been deprecated, and developers are encouraged to use other classes and methods, such as those in the java.net.URLEncoder and java.net.URLDecoder classes.

life cycle of a servlet :

1. **Servlet class is loaded:**

   - The servlet container loads the servlet class into memory. This typically happens when the web application is started or when the servlet is first requested.

2. **Servlet instance is created:**

   - An instance of the servlet is created using its no-argument constructor. This constructor is called by the servlet container during the loading phase.

3. **init method is invoked:**

   - The servlet container calls the init() method on the servlet instance.
   - The init() method is used for initialization tasks, such as loading configuration parameters or establishing connections to databases.
   - This method is executed only once during the servlet's lifecycle.

4. **service method is invoked:**

   - For each incoming HTTP request, the servlet container invokes the service() method on the servlet instance.
   - The service() method determines the type of request (GET, POST, etc.) and calls the appropriate doXXX() method (e.g., doGet() or doPost()).
   - The doXXX() methods handle the specific tasks associated with the incoming request and generate the response.

5. **destroy method is invoked:**

   - When the servlet container decides to take the servlet out of service (e.g., during application shutdown or if the servlet is no longer needed), it calls the destroy() method.
   - The destroy() method is used for cleanup tasks, such as releasing resources or closing database connections.
   - This method is executed only once during the servlet's lifecycle.

File Structure :

my-servlet-app/ |-- src/ | |-- main/ | |-- java/ | |-- com/ | |-- example/ | |-- MyServlet.java |-- webapp/ | |-- WEB-INF/ | |-- web.xml |-- pom.xml

Sure, I'll provide a basic file structure for a servlet-based web application along with information on each component. Let's assume you are using the standard Maven project structure for simplicity:

```
my-servlet-app/
|-- src/
|    |-- main/
|        |-- java/
|            |-- com/
|                |-- example/
|                    |-- MyServlet.java
|-- webapp/
|    |-- WEB-INF/
|        |-- web.xml
|-- pom.xml
```

- MyServlet.java: The servlet class file. This is where you define your servlet and override the necessary methods like init(), doGet(), doPost(), etc.

- webapp/: The web application directory.

- WEB-INF/: A special directory containing configuration files.

- web.xml: The deployment descriptor for your web application. It contains configuration information, servlet mappings, and other settings.

- pom.xml: If you are using Maven, this is the project object model (POM) file. It includes project configuration, dependencies, and build settings.

RequestDispatcher interface

- RequestDispatcher interface is a part of the Java Servlet API and is used to forward the request from one servlet to another or include the response from another servlet in the response of the current servlet.

- Forwarding Requests: Java RequestDispatcher dispatcher = request.getRequestDispatcher("/destinationServlet"); dispatcher.forward(request, response);

This code snippet creates a RequestDispatcher object for the specified destination servlet and forwards the request and response objects to that servlet. This is typically used when you want another servlet to handle the request further in the processing chain. The forward() method works at server side.

- Including Responses: Java RequestDispatcher dispatcher = request.getRequestDispatcher("/includedServlet"); dispatcher.include(request, response);

In this case, the response from the included servlet is incorporated into the response of the current servlet. This is useful when you want to include content from another servlet in the final response.

- Example : In this example, we are validating the password entered by the user. If password is servlet, it will forward the request to the WelcomeServlet, otherwise will show an error message: sorry username or password error!. In this program, we are cheking for hardcoded information. But you can check it to the database

Annotations

1. **@WebServlet:**

   - Used to declare a servlet.
   - Syntax example:

   ```
   @WebServlet("/example")
   public class ExampleServlet extends HttpServlet {
       // Servlet implementation
   }
   ```

2. **@WebInitParam:**

   - Used to specify an initialization parameter for a servlet.
   - Typically used in conjunction with @WebServlet.
   - Example:

   ```
   @WebServlet(value = "/example", initParams = {@WebInitParam(name =
   "paramName", value = "paramValue")})
   public class ExampleServlet extends HttpServlet {
       // Servlet implementation
   }
   ```

3. **@WebFilter:**

   - Used to declare a servlet filter.
   - Syntax example:

   ```
   @WebFilter("/secured/*")
   public class SecurityFilter implements Filter {
       // Filter implementation
   }
   ```

4. **@WebListener:**

   - Used to declare a listener class.
   - Syntax example:

```
@WebListener
public class MyContextListener implements ServletContextListener {
    // Listener implementation
}
```

5. **@HandlesTypes:**

   - Used to declare the class types that a ServletContainerInitializer can handle.
   - Typically used in conjunction with a ServletContainerInitializer implementation.

6. **@HttpConstraint:**

   - Used within the @ServletSecurity annotation to represent security constraints for all HTTP methods.
   - Example:

   ```
   @ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
   public class SecureServlet extends HttpServlet {
       // Servlet implementation
   }
   ```

7. **@HttpMethodConstraint:**

   - Used within the @ServletSecurity annotation to represent security constraints on specific HTTP methods.

8. **@MultipartConfig:**

   - Annotation for specifying that a servlet expects requests that conform to the multipart/form-data MIME type.
   - Example:

   ```
   @WebServlet("/upload")
   @MultipartConfig(fileSizeThreshold = 1024 * 1024 * 2, maxFileSize =
   1024 * 1024 * 10, maxRequestSize = 1024 * 1024 * 50)
   public class FileUploadServlet extends HttpServlet {
       // Servlet implementation
   }
   ```

9. **@ServletSecurity:**

   - Used on a Servlet implementation class to specify security constraints enforced by the servlet container on HTTP protocol messages.
   - Example:

```
@WebServlet("/secure")
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class SecureServlet extends HttpServlet {
    // Servlet implementation
}
```

The HttpServlet, HttpServletRequest, HttpServletResponse :

1. **HttpServlet:**

   - Purpose: Specialized class for handling HTTP requests in servlets.
   - Key Methods: `doGet()`, `doPost()`, `doHead()`, `doDelete()`, etc.

2. **HttpServletRequest:**

   - Purpose: Represents client requests to the server.
   - Common Methods: `getParameter()`, `getHeader()`, `getMethod()`, `getSession()`, `getRequestDispatcher()`.

3. **HttpServletResponse:**

   - Purpose: Represents the server's response to client requests.
   - Common Methods: `setStatus()`, `setContentType()`, `getWriter()`, `sendRedirect()`.

Servlet, DAO, POJO DB Layers :

1. **Servlet Layer: Definition:** Handles HTTP requests, orchestrates application flow, and acts as a controller in the MVC pattern.

```
@WebServlet("/user")
public class UserServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
{
        List<User> userList = UserService.getAllUsers();
        // Process data and send response to the client
        // ...
    }
}
```

2. **DAO Layer (Data Access Objects): Definition:** Manages data access, shields the application from database details, and encapsulates CRUD operations.

Code:

```
public interface UserDao {
    List<User> getAllUsers();
    void addUser(User user);
    // Other CRUD methods...
```

```
}

public class JdbcUserDao implements UserDao {
    // JDBC implementation of DAO methods
    // ...
}
```

3. **POJO Layer (Plain Old Java Objects): Definition:** Represents entities or data structures within the application, encapsulating data and behavior.

```
public class User {
    private int id;
    private String username;
    private String email;
    // Getters and setters
    // Additional business logic or validation methods
}
```

Session :

- A session is a mechanism to maintain state information between multiple requests from the same client.
- Enables the server to associate data with a specific user across different HTTP requests.

Session Tracking Techniques

There are four techniques used in Session tracking:

1. Cookies
2. Hidden Form Field
3. URL Rewriting
4. HttpSession

Cookies in Servlet :

- **Purpose:**

    ○ Cookies are small pieces of data stored on the user's device by the web browser.
    ○ Used for session management, user tracking, and storing preferences.

- **Creating Cookies:**

    ```
    Cookie usernameCookie = new Cookie("username", "user123");
    usernameCookie.setMaxAge(3600); // 1 hour
    response.addCookie(usernameCookie);
    ```

- **Retrieving Cookies:**

```java
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            String name = cookie.getName();
            String value = cookie.getValue();
            // Process cookie data
        }
    }
```

- **Setting Path and Domain:**

```java
    Cookie userCookie = new Cookie("username", "user123");
    userCookie.setPath("/myapp"); // Valid for "/myapp" and subdirectories
    userCookie.setDomain(".example.com"); // Accessible to subdomains of
    "example.com"
```

- **Deleting Cookies:**

```java
    for (Cookie cookie : cookies) {
        if ("username".equals(cookie.getName())) {
            cookie.setMaxAge(0); // Delete the cookie
            response.addCookie(cookie);
        }
    }
```

httpsession :

- In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks

- `HttpSession` is a server-side mechanism in servlets that allows the server to store information about a user's session across multiple requests.

- It enables the creation of stateful web applications by associating data with a specific user.

The HttpServletRequest interface provides two methods to get the object of HttpSession:

1. **public HttpSession getSession():** Returns the current session associated with this request, or if the request does not have a session, creates one.
2. **public HttpSession getSession(boolean create):** Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Usage:

1. **Creating or Retrieving HttpSession:**

```
// Create or retrieve a session
HttpSession session = request.getSession();
```

- ○ The `getSession()` method of the `HttpServletRequest` returns the current session associated with the request or creates a new session if one does not exist.

2. **Storing Data in HttpSession:**

```
// Store data in the session
session.setAttribute("username", "user123");
```

- ○ The `setAttribute()` method allows you to store data in the session using key-value pairs.

3. **Retrieving Data from HttpSession:**

```
// Retrieve data from the session
String username = (String) session.getAttribute("username");
```

- ○ The `getAttribute()` method retrieves data stored in the session. Casting is necessary as the method returns an `Object`.

4. **Setting Session Timeout:**

```
// Set session timeout (in seconds)
int timeoutInSeconds = 60 * 30; // 30 minutes
session.setMaxInactiveInterval(timeoutInSeconds);
```

- ○ The `setMaxInactiveInterval()` method sets the maximum time the session can be inactive (without requests) before it times out.

5. **Invalidating HttpSession:**

```
// Invalidate the session
session.invalidate();
```

- ○ The `invalidate()` method marks the session as invalid, and any associated data is removed.

6. **Session Listeners:**

```
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
```

```java
public class MySessionListener implements HttpSessionListener {
    public void sessionCreated(HttpSessionEvent se) {
        // Session created, perform actions if needed
    }

    public void sessionDestroyed(HttpSessionEvent se) {
        // Session invalidated, perform cleanup actions if needed
    }
}
```

- Implementing **HttpSessionListener** allows you to perform custom actions during session creation and destruction.

7. **Checking if HttpSession Exists:**

```java
// Check if a session exists
HttpSession session = request.getSession(false);

if (session != null) {
    // Session exists, retrieve data or perform actions
} else {
    // Session does not exist, handle accordingly
}
```

- The parameter *false* in **getSession(false)** ensures that a new session is not created if one does not exist.

Paging Navigation : Moving from one servlet to other servlet :

- The **sendRedirect()** method of HttpServletResponse interface can be used to redirect response to another resource, it may be servlet, jsp or html file. The **sendRedirect()** method works at client side.

```java
response.sendRedirect("destination.jsp");
```

Alternative approch for sendRedirect() :

```java
RequestDispatcher dispatcher = request.getRequestDispatcher("destination.jsp");
dispatcher.forward(request, response);
```

Forward VS Redirect

Comparison Chart

Forward	Redirect
It is used to forward the request from one JSP to another or from one JSP to a servlet or from one JSP to another resource.	It is used to redirect the client request to a different URL which is available on a different server or context.
The forward method is declared in the RequestDispatcher.	The SendRedirect( ) method is declared in HttPServletResponse.
The client/browser is not involved and the web container handles the process internally.	The control is transferred to client or browser as the client sees the URL as a new request.
It works best when a component perform business logic and share results with another component.	It works best when the client should be redirected from one page to another.
It operates within the server and executes faster than a redirect.	It is relatively slower than a forward as it requires round-trip communication with the client.
The original remains unchanged in case of forward.	The original URL gets changed in case of redirect.
Both the resources must be part of the same context.	The requests are redirected to a different URL that does not belong to the current context.

DB Difference Between.net

Working of SendRedirect()

Req. is generated for first Servlet.
req. & res object is generated for firstServlet.
due to sendRedirect, the control comes
back to client a new request is
generated (neq req. & resp. objcats) for
Second Servlet & sent to SecondServlet.

Working of forward()

Req. is generated For first Servlet.
req., res obj. is generated for firstServlet.
due to forward(). No new req. is
generated For SecondServlet i.e no new
req, & resp. object is generated. the same
requested is forwarded to Second Servlet.

req
chaining (Req. is generated for firstServlet but the
response is generated for by secondServlet)

Scopes of servlet :

1. **Request Scope:**

   - httpservletrequest object.
   - Objects stored in the request scope are accessible only during the processing of a single HTTP request. (forwared.include)
   - Once the request is processed, the objects are no longer available.
   - Commonly used to share data between different components within the same request.

   ```
   // Storing in request scope
   request.setAttribute("username", "user123");

   // Retrieving in another component (e.g., servlet, JSP)
   String username = (String) request.getAttribute("username");
   ```

2. **Session Scope:**

   - Objects stored in the session scope are accessible across multiple requests made by the same client.
   - The session starts when the client first interacts with the server and ends when the session is invalidated or times out.

  o Suitable for storing user-specific data that needs to persist across multiple requests.

```
// Storing in session scope
HttpSession session = request.getSession();
session.setAttribute("userRole", "admin");

// Retrieving in another component
String userRole = (String) session.getAttribute("userRole");
```

3. **Application Scope (ServletContext Scope):**

   o Objects stored in the application scope are accessible to all servlets and JSP pages within the
      same web application.
   o The scope is shared across all clients and persists for the entire duration of the web application.
   o Suitable for storing global configuration settings, shared resources, etc.

```
// Storing in application scope
ServletContext application = getServletContext();
application.setAttribute("appName", "MyWebApp");

// Retrieving in another component
String appName = (String) application.getAttribute("appName");
```

ServletConfig :

- nth Servlet file have nth servletConfig.
- getServletConfig() method of Servlet interface returns the object of ServletConfig.

ServletContext :

- Every Web App have only one servlet context.
- An object of ServletContext is created by the web container at time of deploying the project.
- If any information is shared to many servlet, it is better to provide it from the web.xml file using the
  context-param element.
- **Context-param** - create context scope using web.xml.

JDBC :

- It is an API that make data transaction possible between java application and any RDBMS.
- JDBC api is inside **java.sql** & **javax.sql** package.

JDBC Architecture :

The JDBC (Java Database Connectivity) architecture consists of two main components: the JDBC API and the
JDBC Driver Manager. These components work together to provide a standardized interface for Java
applications to interact with relational databases. Here's an overview of the JDBC architecture:

1. **JDBC API:**

   - The JDBC API provides a set of interfaces and classes that Java applications use to interact with databases. The key packages involved are `java.sql` and `javax.sql`.

   - Some important interfaces and classes in the JDBC API include:

     - **DriverManager:** Manages a list of database drivers. It is responsible for establishing a connection to the appropriate database using the correct driver. The `getConnection()` method of the `DriverManager` class is commonly used to obtain a database connection.

     - **Connection:** Represents a connection to a specific database. It provides methods for creating statements, committing transactions, and managing other aspects of the connection.

     - Methods :

     1. `createStatement:` Creates a Statement object for executing SQL queries.
     2. `prepareStatement:` Creates a PreparedStatement object for executing precompiled SQL statements.
     3. `commit and rollback:` Manages transaction control.

     - **Statement:** Represents a simple SQL statement to be executed. There are three main types of statements: `Statement` for general-purpose use, `PreparedStatement` for precompiled SQL statements, and `CallableStatement` for executing stored procedures.

     - **ResultSet:** Represents the result set of a query. It provides methods for retrieving data from the database, navigating through the results, and updating the data.

     - **ResultSet Interface:** The ResultSet interface represents the result set of a query. It provides methods for retrieving and navigating through the data returned by a query.

2. **JDBC Driver Manager:**

   - The JDBC Driver Manager is a part of the JDBC architecture responsible for managing a list of database drivers. It helps in establishing a connection to the appropriate database by selecting the appropriate driver.
   - When a Java application needs to connect to a database, it uses the `DriverManager.getConnection()` method, passing the database URL, username, and password. The `DriverManager` then tries to find a suitable driver from the registered drivers and establishes a connection.

3. **JDBC Drivers:**

   - The Driver interface is part of the JDBC API and defines methods for establishing a database connection.

   - A JDBC driver is a software component that allows Java applications to interact with databases. DBC drivers are client-side adapters that convert requests from Java programs to a protocol that the DBMS can understand. The JDBC driver interface allows applications to execute queries and update data across different database systems. There are four types of JDBC drivers:

- **Type 1 (JDBC-ODBC bridge):**
    1. Uses ODBC (Open Database Connectivity) to connect to databases. It's **platform-dependent** and less common in modern applications.
    2. written in native code (c/cpp). some native libraries are required on client m/c.
    3. not supported by java 8. and its is slowest.
- **Type 2 (Native-API driver):**
    1. Uses a database-specific API to connect to databases. It requires client-side native code and is less portable.
    2. faster than type I.
    3. **platform-independent**. Your statements about Type 3 (Network Protocol driver) and Type 4 (Thin driver/Pure Java Driver) are mostly accurate. Let me provide a bit more information for clarity:

- **Type 3 (Network Protocol driver):**

    1. No need for native libraries on the client machine.
    2. Type 3 drivers use a middleware component to convert JDBC calls into a database-specific protocol. This middleware is often referred to as the "bridge" or "gateway."
    3. With a Type 3 driver, you typically only need one driver to access any database for which a Type 3 driver is available. This can make it a more universal solution.
    4. Type 3 drivers are considered universal and platform-independent because the middleware component is responsible for translating the JDBC calls to the native protocol of the database.

- **Type 4 (Thin driver/Pure Java Driver):**

    1. Native-Protocol Driver is indeed associated with Type 4 JDBC drivers.
    2. Type 4 drivers directly communicate with the database using a protocol supported by the database. They are purely Java-based, eliminating the need for any client-side libraries or middleware.
    3. They are platform-independent and are commonly used in modern applications due to their simplicity and ease of use.
    4. Each database vendor provides its own implementation of a Type 4 driver, making it necessary to use a specific Type 4 driver for each database type.

Steps to write JDBC Code :

Steps to write JDBC code

I) Loading Driver Class (Configuring JDBC driver)

```
use
→  Class.ForName ("driver.class.Name");
```

used to load class manually

Built in class 'Class'   Static method

II) Creating JDBC Connection

```
use method
→  DriverManager.getConnection ("url", "username",
                                    "password");
```

III) Creating/writing query Statements.

→ Using one of the three ways.

* General Statement : Statement
* Prepared Statement : PreparedStatement
* Callable Statement : CallableStatement   } Interf

- query statement is represented as an Object using above interfaces.
- Object will encapsulate the query.

IV) Executing Query

→ Using one of the three ways.
- execute ()
   executeUpdate ()
   executeQuery ()

execute () :   for stored functions and procedures.

executeUpdate () : For DML queries.

executeQuery () : For DQL queries (SELECT)

↑

Recommended usage.
(e.g
We can exe. DML using execute()
but it is not recommended)

V) Close the connection

→ close () method.

It is not compulsory but it is a good practice to close the resources.

Sample Code :

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCArchitectureExample {
    static final String JDBC_URL = "jdbc:mysql://localhost:3306/sampledb";
    static final String USERNAME = "your_username";
    static final String PASSWORD = "your_password";

    public static void main(String[] args) {
        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");

            connection = DriverManager.getConnection(JDBC_URL, USERNAME,
PASSWORD);

            statement = connection.createStatement();

            String sqlQuery = "SELECT * FROM employees";
            resultSet = statement.executeQuery(sqlQuery);

            while (resultSet.next()) {
                int employeeId = resultSet.getInt("employee_id");
                String employeeName = resultSet.getString("employee_name");
                double salary = resultSet.getDouble("salary");

                System.out.println("Employee ID: " + employeeId +
                                   ", Employee Name: " + employeeName +
                                   ", Salary: " + salary);
            }
        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                if (resultSet != null) resultSet.close();
                if (statement != null) statement.close();
                if (connection != null) connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

URL : Uniform Resource Locator

- It's a unique identifier that points to a resource on the internet. It's also known as a web address.
- The four main components of a URL are: Protocol, Domain, Path, Query.

http :

- **Stateless Protocol:** HTTP is a stateless protocol, meaning each request from a client to a server is independent, and the server does not retain information about the client's previous requests.

- **Connectionless:** Each request and response in HTTP is independent of previous requests and responses. The connection is typically closed after each request is fulfilled.

JSP : JavaServer Pages

- **Purpose:** JSP simplifies the process of developing dynamic web pages by allowing Java code to be embedded directly into HTML pages.
- **Execution:** JSP pages are translated into servlets and then compiled to bytecode, providing efficient and dynamic content generation.
- Controller - Servlet.
- View - JSP.
- user data and business logic - JavaBeans.

Life cycle of a JSP page :

The life cycle of a JavaServer Pages (JSP) page involves several steps, from the translation of JSP code into a servlet to the handling of client requests and producing dynamic content.

Certainly! Let's break down the stages in the life cycle of a JSP page as described:

1. **Translation of JSP page to Servlet:**

   - When a JSP page is requested for the first time, the JSP container translates the JSP code into a servlet.
   - The generated servlet typically has a `.java` extension and is created in the work directory of the application server.

2. **Compilation of JSP page (Compilation of JSP into test.java):**

   - The generated `.java` file is then compiled into bytecode (`.class` file) by the Java compiler.
   - The resulting `.class` file contains the compiled servlet code.

3. **Classloading (test.java to test.class):**

   - The compiled `.class` file is loaded into the Java Virtual Machine (JVM) by the classloader when needed.
   - The classloader is responsible for loading classes into memory so that they can be executed.

4. **Instantiation (Object of the generated Servlet is created):**

- An instance of the servlet class is created by the servlet container during the first request to the JSP page.
- The container calls the `jspInit()` method during the instantiation phase, allowing the servlet to perform one-time initialization.

5. **Initialization (jspInit() method is invoked by the container):**

- The `jspInit()` method is called once during the servlet's initialization.
- This method is used for any necessary setup or resource allocation.
- Signature: `public void jspInit() throws ServletException`.

6. **Request processing (_jspService() is invoked by the container):**

- For each subsequent request to the JSP page, the servlet container calls the `service()` method (which is actually `_jspService()` in the generated servlet).
- The `_jspService()` method is responsible for processing the request, handling user input, and generating dynamic content.
- Signature: `public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException`.

7. **JSP Cleanup (jspDestroy() method is invoked by the container):**

- When the servlet container decides to unload the servlet (e.g., during application shutdown or reloading), the `jspDestroy()` method is invoked.
- The `jspDestroy()` method is used for cleanup tasks, such as releasing resources acquired during the servlet's life cycle.
- Signature: `public void jspDestroy()`.

Key Methods in the JSP Life Cycle:

1. `jspInit()` **Method:**

- Called during the instantiation of the servlet.
- Used for one-time initialization tasks, such as setting up resources or establishing database connections.
- Signature: `public void jspInit() throws ServletException`.

2. `service()` **Method:**

- Called for each client request to the JSP page.
- Handles the processing of the request, including interaction with the Model and rendering the dynamic content.
- Signature: `public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException`.

3. `jspDestroy()` **Method:**

- Called when the servlet is being unloaded or the container is shutting down.
- Used for cleanup tasks, such as releasing resources acquired during the servlet's life cycle.
- Signature: `public void jspDestroy()`.

Example:

Consider a simple JSP page that displays a greeting based on a parameter:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Greeting Page</title>
</head>
<body>
    <%
        String name = request.getParameter("name");
        if (name != null && !name.isEmpty()) {
            out.println("<h1>Hello, " + name + "!</h1>");
        } else {
            out.println("<h1>Hello, Guest!</h1>");
        }
    %>
</body>
</html>
```

- **JSP Directives:** JSP directives are used to provide global information about how the JSP page should be processed by the container. Directives are defined using the `<%@ ... %>` syntax.

  1. **Page Directive:**

- Defines attributes related to the page, such as language, contentType, and errorPage.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

- **The `import` Attribute:**

  - **Example:** `<%@ page import="java.util.*" %>`
  - Allows you to specify Java packages that should be imported by the servlet resulting from the JSP page. It helps in using classes from the specified packages within the JSP file.

- **The `contentType` Attribute:**

  - **Example:** `<%@ page contentType="text/plain" %>`
  - Sets the `Content-Type` response header, indicating the MIME type of the document being sent to the client. It specifies how the content should be interpreted by the browser.

- **The `isThreadSafe` Attribute:**

  - **Example:** `<%@ page isThreadSafe="true" %>`
  - Controls whether or not the servlet generated from the JSP page will implement the `SingleThreadModel` interface. The default value is `true`, assuming that the code in the JSP page is thread-safe.

- **The `session` Attribute:**

  - **Example:** `<%@ page session="true" %>`
  - Controls whether the page participates in HTTP sessions. A value of `true` (default) indicates that the `session` variable (of type `HttpSession`) should be bound to an existing session or a new session should be created and bound to `session`.

- **The `buffer` Attribute:**

  - **Example:** `<%@ page buffer="sizekb" %>`
  - Specifies the size of the buffer used by the `out` variable, which is of type `JspWriter` (a subclass of `PrintWriter`). It influences how much content is buffered before being sent to the client.

- **The `autoflush` Attribute:**

  - **Example:** `<%@ page autoflush="true" %>`
  - Controls whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows.

- **The `extends` Attribute:**

  - **Example:** `<%@ page extends="package.class" %>`
  - Indicates the superclass of the servlet that will be generated for the JSP page. It allows you to customize the base class of the generated servlet.

- **The `info` Attribute:**

  - **Example:** `<%@ page info="Some Message" %>`
  - Defines a string that can be retrieved from the servlet using the `getServletInfo` method. It provides additional information about the JSP page.

- **The `errorPage` Attribute:**

  - **Example:** `<%@ page errorPage="Relative URL" %>`
  - Specifies a JSP page that should process any exceptions (of type `Throwable`) thrown but not caught in the current page. The exception is automatically available to the designated error page via the `exception` variable.

- **The `isErrorPage` Attribute:**

  - **Example:** `<%@ page isErrorPage="true" %>`
  - Indicates whether or not the current page can act as the error page for another JSP page. If `true`, it means that the page can handle errors.

2. **Include Directive:**

- Includes the content of another resource during translation.

```
<%@ include file="header.jsp" %>
```

3. **Taglib Directive:**

- Declares the use of custom tag libraries.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

- **Implicit and Explicit Objects:** JSP provides a set of predefined objects that are available for use without explicit declaration.

1. **Implicit Objects:**

- Objects that are automatically available in JSP pages.
- Examples include `request`, `response`, `session`, `application`, `out` (JspWriter), `config`, and `pageContext`.

ii) **JSP implicit Object**

out
⤷ object of type JspWriter

request
⤷ of type HttpServletRequest

response
⤷ of type HttpServletResponse

page
⤷ represents current page (~ this)

page Context
⤷ of type PageContext

session
⤷ of type HttpSession

application
⤷ of type ServletContext

config
⤷ of the type ServletConfig

exception
⤷ of type Throwable.

exception object is not available in all jsp files
It is only available in error jsp.

2. **Explicit Objects:**

- Objects that need to be explicitly declared or obtained.

- Examples include objects created in JavaBeans, custom objects, and objects obtained from JSP tags.

- **Scriptlets:** Scriptlets are used to embed Java code within HTML in a JSP page. They are enclosed between `<%` and `%>` tags.

```
<%
  String message = "Hello, World!";
  out.println(message);
%>
```

While scriptlets provide a way to embed Java code, it's generally considered better practice to minimize the use of scriptlets and promote the use of other mechanisms like JSP expressions and custom tags. it code goes to the JspService method.

- **Expressions:** JSP expressions are used to embed dynamic values directly into the HTML content. They are enclosed between `<%= ... %>` tags.

```
<p>The current date is: <%= new java.util.Date() %></p>
```

Expressions automatically convert the result to a string and include it in the HTML output.

- **Declaration:** `<%! ... %>` syntax and are typically placed outside the main HTML content. They are used to declare members of the servlet class that are accessible throughout the JSP page. used to class variable or to define method.

- **Expression Language (EL):** Expression Language is a powerful feature introduced in JSP 2.0 that simplifies the retrieval and manipulation of data stored in JavaBeans components. EL expressions are written using `${ ... }` syntax.

JSTL (JavaServer Pages Standard Tag Library) :

JSTL, which stands for JavaServer Pages Standard Tag Library, is a collection of tag libraries that provide common functionality for JavaServer Pages (JSP). JSTL simplifies the development of dynamic web applications by providing a set of tags for tasks such as iteration, conditionals, formatting, database access, and more. It encourages a cleaner separation of concerns by reducing the need for extensive Java code within JSP pages. Here's an overview of some key aspects of JSTL:

1. **Core Tags (`<c:...>`):**

- **Iteration:**
  - `<c:forEach>`: Iterates over a collection.

```
<c:forEach var="item" items="${items}">
   ${item}<br>
</c:forEach>
```

- **Conditional Statements:**
    - `<c:if>`, `<c:choose>`, `<c:when>`, `<c:otherwise>`: Perform conditional logic.

```
<c:if test="${condition}">
   <!-- Content to be displayed if the condition is true -->
</c:if>
```

- **Variable Assignment:**
    - `<c:set>`: Assigns a value to a variable.

```
<c:set var="result" value="${someExpression}" />
```

- **Import and URL Management:**
    - `<c:import>`, `<c:url>`: Manages imports and URLs.

```
<c:import url="/path/to/resource"/>
<a href="<c:url value='/path/to/page'/>">Link</a>
```

2. **Formatting Tags (`<fmt:...>`):**

- **Localization and Formatting:**
    - `<fmt:setLocale>`, `<fmt:formatDate>`, `<fmt:formatNumber>`: Provides localization and formatting features.

```
<fmt:setLocale value="en_US"/>
<fmt:formatDate value="${date}" pattern="yyyy-MM-dd"/>
```

3. **SQL Tags (`<sql:...>`):**

- **Database Access:**
    - `<sql:setDataSource>`, `<sql:query>`: Perform database operations.

```
<sql:setDataSource var="ds" driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/db" user="username" password="password"/>
<sql:query var="result" dataSource="${ds}">
   SELECT * FROM tableName
</sql:query>
```

4. **XML Tags (`<x:...>`):**

- **XML Processing:**
  - `<x:parse>`, `<x:out>`, `<x:forEach>`: Deals with XML processing.

```
<x:parse xml="${xmlData}" var="parsedXML"/>
<x:forEach select="$parsedXML//element" var="element">
   ${element.text}<br>
</x:forEach>
```

5. **Functions (`fn:...`):**
   - The `fn:` prefix provides a set of functions for string manipulation, substring extraction, and other common operations.

```
<c:set var="name" value="John Doe"/>
<c:out value="${fn:length(name)}"/> <!-- Outputs the length of the string --
>
```

- Usage: To use JSTL in a JSP page, you need to include the JSTL library in your project and declare the taglib at the top of the JSP page:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<!-- ... other taglib declarations ... -->
```

JavaBeans :

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,

1. The class must implement either Serializable or Externalizable
2. The class must have a public no-arg constructor
3. All properties must have public setter and getter methods
4. All instance variables should be private
5. To access JavaBean properties, you can use the jsp:getProperty/ action to access the get methods and the jsp:setProperty/ action to access the set methods.

The provided information explains the basic usage of JavaBeans in JavaServer Pages (JSP). Let's break down the key concepts:

Using `<jsp:useBean>`:

The `<jsp:useBean>` action is used to load a bean into a JSP page. This allows you to leverage the reusability of Java classes within your JSP.

Syntax:

```
<jsp:useBean id="name" class="package.Class" />
```

This syntax means "instantiate an object of the class specified by `Class` and bind it to a variable with the name specified by `id`." You can also specify a `scope` attribute to associate the bean with more than just the current page.

Accessing Bean Properties:

Once a bean is instantiated, you can access its properties using `<jsp:getProperty>`. This tag takes a `name` attribute (matching the `id` in `<jsp:useBean>`) and a `property` attribute (naming the property of interest).

Example:

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

Setting Bean Properties:

To modify bean properties, you typically use `<jsp:setProperty>`.

1. **To set an individual specific property:**

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

2. **To set all properties or Associating All Properties with Input Parameters:**

   ```
   <jsp:setProperty name="book1" property="*" />
   ```

Sharing Beans:

Beans can be stored in different locations based on the optional `scope` attribute of `<jsp:useBean>`. The `scope` attribute determines the visibility and lifespan of the bean. Possible values for `scope` include:

- `page`: The bean is accessible only within the current JSP page.
- `request`: The bean is accessible across multiple pages in a single request.

- `session`: The bean is accessible across multiple requests within the same session.
- `application`: The bean is accessible globally to all users in the application.

Example:

```
<jsp:useBean id="book1" class="com.MyBook" scope="session" />
```

Hibernate Framework :

- It was started in 2001 by Gavin King as an alternative to EJB2 style entity bean.
- It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

ORM Tool :

- An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.
- The ORM tool internally uses the JDBC API to interact with the database.
- Hibernate automatically translates Java objects into corresponding database tables and manages the data interchange.

features of Hibernate:

1. **Object-Relational Mapping (ORM):** Hibernate provides a powerful and flexible ORM framework that allows developers to map Java objects to database tables and vice versa. This eliminates the need for writing SQL queries and provides a more object-oriented approach to database interaction.

2. **Transparent Persistence:** Hibernate enables transparent persistence, meaning that developers can work with plain Java objects in their application code, and Hibernate takes care of persisting these objects to the database.

3. **Database Independence:** Hibernate abstracts away database-specific details, allowing developers to write database-agnostic code. This means that applications can easily switch between different database systems without major code changes.

4. **Automatic Table Generation:** Hibernate can automatically generate database tables based on the Java classes, reducing the need for manual table creation and schema management.

5. **Query Language (HQL):** Hibernate Query Language (HQL) is a powerful and database-agnostic query language that allows developers to express queries in terms of their Java domain model, rather than SQL. It provides a higher level of abstraction and is particularly useful in complex scenarios.

6. **Caching:** Hibernate supports caching at both the session level (first-level cache) and application level (second-level cache). Caching can significantly improve performance by reducing the number of database queries.

7. **Lazy Loading:** Hibernate supports lazy loading, which means that it loads data from the database only when it is explicitly requested. This can help optimize resource usage and improve application

performance.

8. **Transaction Management:** Hibernate supports transaction management, allowing developers to define and manage transactions either programmatically or declaratively using annotations.

9. **Integration with Java EE and Spring:** Hibernate can be seamlessly integrated with Java EE applications and the Spring Framework, providing developers with additional features and flexibility when building enterprise-level applications.

10. **Annotations and XML Configuration:** Hibernate supports both annotation-based and XML-based configuration, allowing developers to choose the configuration approach that best suits their preferences and project requirements.

11. **Support for Composite Keys and Associations:** Hibernate supports complex associations between entities, including one-to-one, one-to-many, and many-to-many relationships. It also handles composite primary keys in a straightforward manner.

12. **Multi-tier Architecture Support:** Hibernate is well-suited for multi-tier architectures, enabling the development of scalable and maintainable applications by separating concerns between the presentation layer, business logic layer, and data access layer.

Hibernate Architecture :

- **Layers:**

1. Java application layer

2. Hibernate framework layer

3. Backend api layer

4. Database layer

5. **Persistent Object:**

   - **Definition:** A persistent object is a Plain Old Java Object (POJO) that is mapped to a database table using Hibernate. Instances of these objects represent data in the application and are persisted to and retrieved from the database.

```
@Entity
public class User {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String username;
 private String email;
 }
```

6. **Session Factory:**

- **Definition:** The Session Factory is a heavyweight, thread-safe object responsible for creating and managing Session instances. It is created once during application startup and is used to obtain Session instances throughout the application's lifecycle.
- **Purpose:** The Session Factory is a central point for configuring Hibernate and managing the mapping between Java objects and database tables. It caches the metadata about the object-relational mappings and provides efficient creation of Session instances.

```
Configuration configuration = new
Configuration().configure("hibernate.cfg.xml");
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

7. **Transaction Factory:**

- **Definition:** The Transaction Factory is responsible for creating Transaction instances. It is part of the Hibernate architecture to manage database transactions.
- **Purpose:** Transactions ensure the consistency and integrity of data by grouping database operations into atomic units. The Transaction Factory provides a way to create and manage transactions in Hibernate.

```
Transaction transaction = session.beginTransaction();
// Perform database operations
transaction.commit();
```

8. **Connection Factory:**

- **Definition:** The Connection Factory is responsible for creating database connections. It plays a crucial role in managing the connections used by Hibernate to interact with the database.
- **Purpose:** The Connection Factory helps with connection pooling, which is the practice of reusing existing database connections rather than creating a new connection for each database operation. This can significantly improve performance.

```
ComboPooledDataSource dataSource = new ComboPooledDataSource();
dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/mydatabase");
dataSource.setUser("username");
dataSource.setPassword("password");
```

9. **Session:**

- **Definition:** A Session is a short-lived, non-thread-safe object representing a single unit of work with the database. It is obtained from the Session Factory and is used to perform database operations.
- **Purpose:** The Session is a key interface in Hibernate that provides methods for CRUD operations, queries, and transaction management. It is responsible for maintaining a first-level cache to store objects that have been recently read or written.

10. **Transaction:**

    ○ **Definition:** The Transaction interface represents a database transaction. It is used to demarcate the boundaries of a transactional unit of work in Hibernate.
    ○ **Purpose:** Transactions ensure the atomicity, consistency, isolation, and durability (ACID properties) of database operations. The Transaction object provides methods to manage and control transactions, such as commit and rollback.

lifecycle of Hibernate :

The entities refers to the various states an entity goes through from the time it is instantiated until it is no longer associated with a Hibernate session. The Hibernate entity lifecycle consists of several states, and transitions between these states are managed by the Hibernate framework. The entity lifecycle is often represented as a finite state machine with the following states:

1. **Transient (New):**

   ○ An entity is in the transient state when it is just created using the `new` keyword but is not associated with any Hibernate session.
   ○ It doesn't have a database representation, and changes made to the object are not tracked by Hibernate.

   ```
   YourEntity entity = new YourEntity(); // Transient state
   ```

2. **Persistent (Managed):**

   ○ An entity becomes persistent when it is associated with a Hibernate session using methods like `save()`, `saveOrUpdate()`, `persist()`, or `update()`.
   ○ Changes to the entity are tracked, and these changes will be synchronized with the database when the session is flushed.

   ```
   session.save(entity); // Persistent state
   ```

3. **Detached:**

   ○ An entity becomes detached when the Hibernate session is closed, or the entity is explicitly detached using `evict()` or `clear()` methods.
   ○ The entity is no longer associated with any session, but changes made to it are not automatically synchronized with the database.

   ```
   session.evict(entity); // Detached state
   ```

4. **Removed:**

- An entity is in the removed state when it was persistent, but the `delete()` or `remove()` method is called on it.
- The entity is scheduled for deletion, but the deletion is not yet executed until the session is flushed.

```
session.delete(entity); // Removed state
```

JPA :

The Java Persistence API (JPA) is a Java specification for managing relational data in applications. It is part of the Java EE (Enterprise Edition) and Jakarta EE specifications. JPA simplifies the development of data-driven applications by providing a standardized way to interact with relational databases using object-relational mapping (ORM) techniques. Below is a detailed overview of JPA:

Key Concepts:

1. **Entity:**

   - An entity is a plain Java object (POJO) representing a data record in a database table.
   - Entities are annotated with `@Entity` to indicate their mapping to database tables.

   ```java
   @Entity
   public class Employee {
       // Entity properties and methods
   }
   ```

2. **EntityManager:**

   - `EntityManager` is the central interface in JPA for managing entities and their lifecycle.
   - It provides methods for CRUD operations (persist, find, merge, remove) and executing queries.

   ```java
   @PersistenceContext
   private EntityManager entityManager;

   public void saveEmployee(Employee employee) {
       entityManager.persist(employee);
   }
   ```

3. **Primary Key and Identity:**

   - The `@Id` annotation designates a field as the primary key of an entity.
   - The `@GeneratedValue` annotation defines the strategy for generating primary key values.

   ```java
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   ```

```
    private Long id;
```

4. **Relationships:**

   - JPA supports relationships between entities, including @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany.
   - Relationships are defined using annotations like @JoinColumn and @MappedBy.

```
@Entity
public class Department {
    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
}

@Entity
public class Employee {
    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

5. **JPQL (Java Persistence Query Language):**

   - JPQL is a query language similar to SQL, but it operates on entities.
   - Queries are written using entity names and fields.

```
TypedQuery<Employee> query = entityManager.createQuery(
    "SELECT e FROM Employee e WHERE e.salary > :salary", Employee.class);
query.setParameter("salary", 50000.0);
List<Employee> highPaidEmployees = query.getResultList();
```

Advanced Features:

6. **Inheritance Mapping:**

   - JPA supports mapping inheritance hierarchies with strategies like @Inheritance and @DiscriminatorColumn.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "employee_type", discriminatorType =
DiscriminatorType.STRING)
public class Employee {
    // Common properties and methods
}

@Entity
```

```java
@DiscriminatorValue("FTE")
public class FullTimeEmployee extends Employee {
    // Full-time employee properties and methods
}
```

7. **Named Queries:**

   ○ Named queries allow developers to define queries in metadata and refer to them by name.

```java
@Entity
@NamedQuery(name = "findEmployeeByName", query = "SELECT e FROM Employee e
WHERE e.name = :name")
public class Employee {
    // Entity properties and methods
}
```

8. **Criteria API:**

   ○ The Criteria API provides a programmatic and type-safe way to build queries using Java code.

```java
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> root = query.from(Employee.class);
query.select(root).where(cb.gt(root.get("salary"), 50000.0));
List<Employee> highPaidEmployees =
entityManager.createQuery(query).getResultList();
```

9. **Caching:**

   ○ JPA supports caching to improve performance by reducing the number of database queries.
   ○ Entities and query results can be cached in-memory.

```java
@Cacheable
@Entity
public class Employee {
    // Entity properties and methods
}
```

10. **Listeners:**

   ○ JPA provides entity lifecycle event listeners for executing custom logic during various lifecycle
     events (e.g., @PrePersist, @PostLoad).

```java
@EntityListeners(EmployeeListener.class)
public class Employee {
```

```
        // Entity properties and methods
    }

    public class EmployeeListener {
        @PrePersist
        public void prePersist(Employee employee) {
            // Custom logic before persisting an entity
        }
    }
```

11. **Validation:**

    o Bean Validation annotations (@NotNull, @Size, etc.) can be used on entity properties for data validation.

    ```
    @Entity
    public class Employee {
        @NotNull
        private String name;
        // Other properties and methods
    }
    ```

12. **Locking:**

    o JPA provides optimistic and pessimistic locking mechanisms to control concurrent access to entities.

    ```
    public class EmployeeService {
        @Transactional
        public void updateEmployeeWithOptimisticLock(Long id, String newName) {
            Employee employee = entityManager.find(Employee.class, id);
            employee.setName(newName);
            // Optimistic locking will be performed during the update
        }
    ```

Annotations:

Hibernate uses to provide metadata to map Java objects to database tables. Here are some common Hibernate annotations with small code examples: All the JPA annotations are defined in the javax.persistence package.

1. **@Entity:**

    o Marks a class as an entity, representing a table in the database.

    ```
    @Entity
    public class YourEntity {
    ```

```
        // Entity properties and methods
    }
```

2. **@Table**:

   - Specifies the details of the database table to which the entity is mapped.

   ```
   @Entity
   @Table(name = "your_table")
   public class YourEntity {
       // Entity properties and methods
   }
   ```

3. **@Id**:

   - Marks a field as the primary key of the entity.

   ```
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   ```

4. **@GeneratedValue**:

   - Specifies the generation strategy for the values of the primary key.

   ```
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   ```

5. **@Column**:

   - Maps a field to a specific column in the database.

   ```
   @Column(name = "name_column")
   private String name;
   ```

6. **@OneToMany**:

   - Indicates a one-to-many relationship between entities.

   ```
   @Entity
   public class ParentEntity {
       @OneToMany(mappedBy = "parent")
       private List<ChildEntity> children;
   ```

```
    }

    @Entity
    public class ChildEntity {
        @ManyToOne
        @JoinColumn(name = "parent_id")
        private ParentEntity parent;
    }
```

7. **@ManyToOne**:

    ○ Indicates a many-to-one relationship between entities.

    ```
    @Entity
    public class ChildEntity {
        @ManyToOne
        @JoinColumn(name = "parent_id")
        private ParentEntity parent;
    }
    ```

8. **@JoinColumn**:

    ○ Specifies the column used for joining an entity association or element collection.

    ```
    @Entity
    public class ChildEntity {
        @ManyToOne
        @JoinColumn(name = "parent_id")
        private ParentEntity parent;
    }
    ```

9. **@Transient**:

    ○ Marks a field as transient, indicating that it should not be persisted to the database.
    ○ Tells the hibernate, not to add this particular column.

    ```
    @Transient
    private transientField;
    ```

10. **@NamedQuery**:

    ○ Specifies a named query.

    ```
    @Entity
    @NamedQuery(name = "findByName", query = "SELECT e FROM YourEntity e WHERE
    ```

```
    e.name = :name")
public class YourEntity {
    // Entity properties and methods
}
```

11. **@Temporal:**

- This annotation is used to format the date for storing in the database.

12. **@Lob:**

- Used to tell hibernate that it's a large object and is not a simple object.

Hibernate caching:

1. **First Level Cache:**

   - The first level cache is associated with the Session object. It is enabled by default and cannot be disabled.
   - It is also known as the Session cache or the local cache.
   - The first level cache is used to store the entities that are currently being manipulated within the scope of a single Hibernate Session.
   - When you load an entity using `session.get()` or `session.load()`, the entity is stored in the first level cache.
   - Subsequent requests for the same entity within the same session will be served from the first level cache rather than hitting the database again.
   - The first level cache is transactional, meaning it is cleared when the transaction is committed or rolled back.

2. **Second Level Cache:**

   - The second level cache is a cache shared by all the sessions of a specific SessionFactory.
   - It is optional and needs to be configured explicitly. Hibernate supports various second-level cache providers like Ehcache, Infinispan, Hazelcast, etc.
   - Entities and collections loaded in one session can be stored in the second level cache and retrieved by another session, reducing the need for hitting the database.
   - It is effective in scenarios where multiple sessions need access to the same data, promoting data sharing and reducing redundant database queries.
   - The second level cache is not transactional by default. It needs to be managed manually to ensure consistency in a multi-user environment.

3. **Query Level Cache:**

   - Query caching is another level of caching that allows caching the results of a query.
   - It operates on the level of individual queries and caches the result sets, which can be shared among different sessions.
   - It can be configured independently of the second level cache.
   - Query caching is beneficial when the same query is executed multiple times with the same parameters, as it avoids re-executing the query and retrieves the results from the cache.

To enable and configure these caches, you would typically modify your Hibernate configuration file
(`hibernate.cfg.xml`) or use programmatic configuration. The specific configuration details depend on the
cache provider you choose for the second level cache.

Spring :

The Spring Framework is an open-source framework for building enterprise applications in Java. It provides
comprehensive infrastructure support, ensuring easier configuration and integration of your application.
Spring is modular, making it possible to use its components independently or together as needed. It
promotes a layered architecture, code organization, and best practices, facilitating the development of
scalable and maintainable applications.

Key features and components of the Spring Framework include:

1. **Inversion of Control (IoC):** Spring's IoC container manages the objects of your application, promoting
   loose coupling and making it easier to test and maintain code.

2. **Dependency Injection (DI):** Spring supports dependency injection, allowing you to inject
   dependencies into your classes rather than having the classes create or look up their dependencies.

3. **Aspect-Oriented Programming (AOP):** AOP is a programming paradigm that separates cross-cutting
   concerns, such as logging or transaction management, from the main business logic. Spring AOP
   provides a way to implement aspects in a modular and reusable manner.

4. **Data Access:** Spring provides support for data access through JDBC (Java Database Connectivity) and
   Object-Relational Mapping (ORM) frameworks like Hibernate. It simplifies database operations and
   abstracts away the complexities of working with data.

5. **Transaction Management:** Spring offers a consistent and flexible transaction management framework
   that supports both programmatic and declarative transaction management.

6. **Model-View-Controller (MVC):** Spring MVC is a web module that simplifies the development of web
   applications by providing a Model-View-Controller architecture. It integrates with other Spring features
   to create robust and scalable web applications.

7. **Security:** The Spring Security module provides comprehensive security services for Java EE-based
   enterprise software applications.

8. **Spring Boot:** Spring Boot is a project within the Spring ecosystem that simplifies the process of
   building production-ready applications. It offers defaults for configuration and provides a wide range of
   features to accelerate development.

9. **Spring Cloud:** Spring Cloud is a set of tools for building distributed systems. It provides solutions for
   service discovery, configuration management, load balancing, and more, facilitating the development of
   microservices-based architectures.

Overview of Spring Architecture :
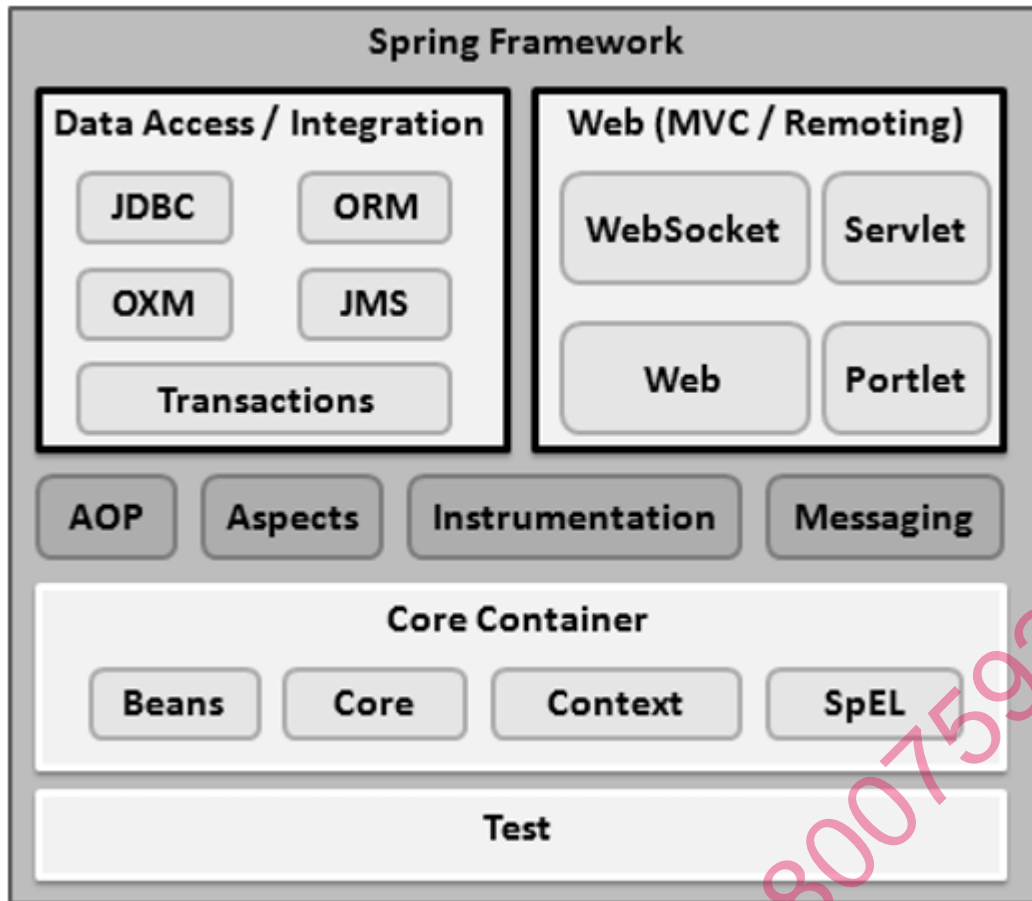
- **Spring Core Container:**

1. **Spring Core:**

    - **Functionality:** Provides the fundamental features of the Spring framework, focusing on IoC and DI.
    - **IoC Container:** Central component responsible for creating and managing instances of JavaBeans.
    - **Dependency Injection (DI):** Utilizes DI to wire beans together, reducing tight coupling and promoting modularity.

2. **Spring Beans:**

    - **Role:** Essential module constituting the basic building block of the IoC container.
    - **BeanFactory:** Core interface for accessing the IoC container, responsible for managing beans.
    - **BeanWrapper:** Manages the lifecycle of a bean, handling various aspects such as instantiation and destruction.

3. **Spring Context:**

    - **Role:** An advanced version of the BeanFactory, offering additional features for application context management.
    - **ApplicationContext:** Provides advanced functionalities such as internationalization, resource loading, and event publication and consumption.
    - **Enhancements:** Represents an evolution beyond the basic capabilities of the BeanFactory.

4. **Spring Expression Language (SpEL):**

- **Functionality:** Offers a powerful expression language for runtime querying and manipulation of objects.
- **Features:** Supports a diverse set of features, including property access, method invocation, conditionals, loops, and type conversion.
- **Integration:** Integrated with the application context, allowing access to variables and functions defined in the context.
- **Customization:** Permits the definition of custom functions and variables, enhancing flexibility.

- **Data Access/Integration:**

1. **Spring JDBC:**

   - **Purpose:** Provides a simplified JDBC abstraction layer to reduce boilerplate code.
   - **Functionality:** Enables developers to work with JDBC more efficiently.
   - **Transaction Management:** Supports transaction management, allowing for declarative management of database transactions using Spring's transaction management.

2. **Spring ORM:**

   - **Integration:** Designed for integration with Object-Relational Mapping (ORM) frameworks, such as Hibernate and JPA.
   - **Abstraction Layer:** Offers a higher-level abstraction layer on top of ORM frameworks.
   - **Benefits:** Reduces boilerplate code and facilitates seamless integration with other Spring features like transaction management and caching.

3. **Spring Data:**

   - **Purpose:** Provides a consistent and easy-to-use programming model for various data access technologies.
   - **Supported Technologies:** Includes databases, NoSQL, and cloud-based data services.
   - **Features:**
     - Automatic CRUD Operations: Simplifies Create, Read, Update, Delete operations.
     - Query Generation: Generates queries from method names.
     - Pagination and Sorting: Supports features like pagination and sorting.
     - Integration with Transaction Management: Integrates with Spring's transaction management.
     - Common Patterns: Supports common data access patterns like repositories and data access objects (DAOs).

4. **Spring Transaction:**

   - **Declarative Transaction Management:** Provides support for declarative transaction management in Spring applications.
   - **Transaction Levels:** Supports various transaction propagation and isolation levels.
   - **Granularity:** Allows developers to manage transactions at different levels of granularity.
   - **Transaction Management Strategies:** Offers support for different transaction management strategies, such as using a JTA transaction manager or a simple JDBC transaction manager.

- **Web layer:**

1. **Spring MVC:**

   - **Purpose:** Provides a Model-View-Controller (MVC) framework for developing web applications.
   - **Features:**
     - HTTP Request and Response Handling: Supports handling of HTTP requests and responses.
     - Form Handling: Facilitates the handling of forms in web applications.
     - Data Binding: Allows binding of data between the model and view.
     - Validation: Provides support for data validation in web forms.
     - View Technologies: Supports various view technologies, including JSP, Thymeleaf, and Velocity.
   - **Flexibility:** Allows developers to choose the view technology that best fits their needs.

2. **Spring WebFlux:**

   - **Reactive Programming Model:** Provides a reactive programming model for building web applications.
   - **Scalability and Concurrency:** Suited for applications requiring high concurrency and scalability.
   - **Technologies Supported:** Works with technologies such as Netty, Undertow, and Servlet 3.1+ containers.
   - **Features:**
     - Reactive Data Access: Supports reactive data access in web applications.
     - Reactive Stream Processing: Facilitates reactive stream processing.
     - Reactive HTTP Clients: Provides support for building reactive HTTP clients.

3. **Spring Web Services:**

   - **Support for Web Services:**
     - SOAP-Based: Supports the development of SOAP-based web services.
     - RESTful: Provides tools for building RESTful web services.
   - **WSDL Generation:** Supports the generation of WSDL from Java classes.
   - **Java Classes Generation:** Allows the generation of Java classes from WSDL.
   - **Contract Definition:** Enables developers to define the contract (interface) of web services using WSDL.

Spring MVC Architecture :
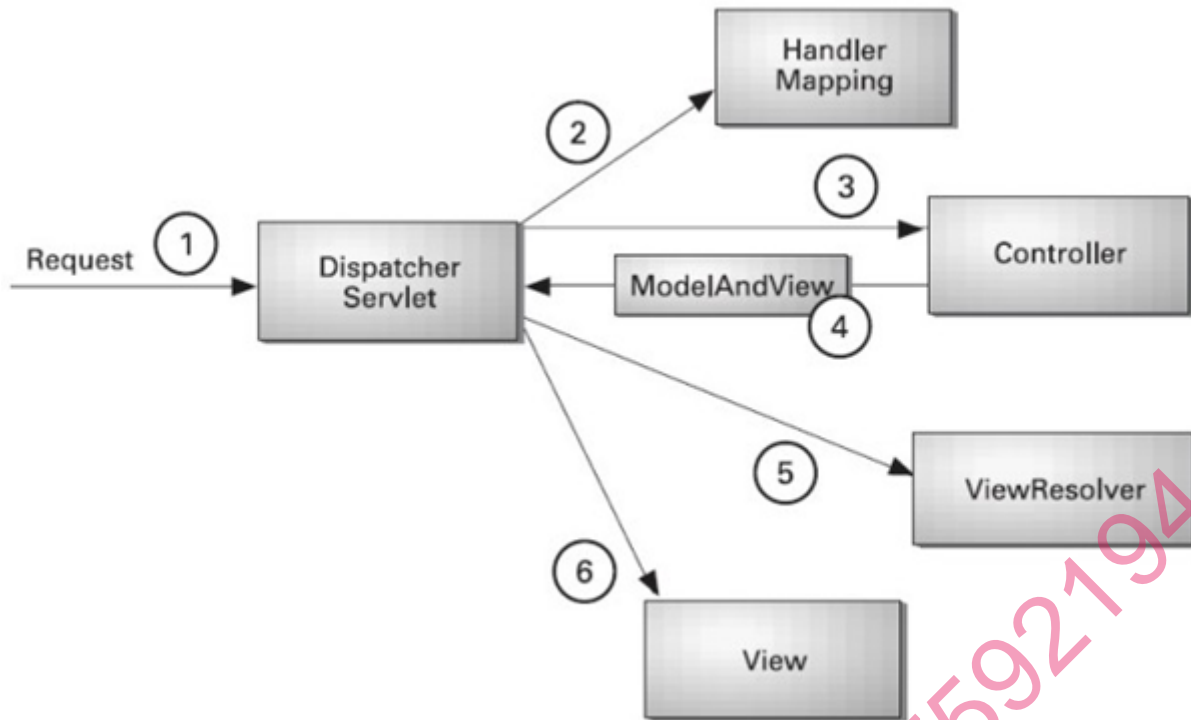
1. **Model:**

   - ○ **Purpose:** Represents the application's data and business logic.
   - ○ **Components:** Consists of Java classes that represent the application's business entities and logic.
   - ○ **Responsibilities:**
     - ■ Holds and manages application state.
     - ■ Performs business logic and data processing.
     - ■ Sends data to the view for rendering.

2. **View:**

   - ○ **Purpose:** Responsible for presenting the data to the user.
   - ○ **Components:** Typically includes templates or JSP pages.
   - ○ **Responsibilities:**
     - ■ Renders the model data received from the controller.
     - ■ Generates the user interface for presentation.
     - ■ No business logic; focuses on displaying information.

3. **Controller:**

   - ○ **Purpose:** Handles user requests, processes them, and manages the flow of data.
   - ○ **Components:** Spring MVC controller classes.
   - ○ **Responsibilities:**
     - ■ Receives and processes user requests from the browser.
     - ■ Invokes appropriate methods in the model to perform business logic.
     - ■ Selects the appropriate view to render based on the result.
     - ■ Manages the overall flow of the application.

4. **DispatcherServlet:**

   - ○ **Purpose:** Central servlet in the Spring MVC framework.

- **Components:** Configured in the web.xml file.
- **Responsibilities:**
  - Intercepts incoming requests and delegates them to the appropriate controller.
  - Manages the overall request lifecycle.
  - Coordinates the flow between the model, view, and controller.
  - Utilizes HandlerMapping to determine the appropriate controller for a request.

5. **HandlerMapping:**

  - **Purpose:** Determines which controller should handle a specific user request.
  - **Components:** Configured in the Spring configuration file.
  - **Responsibilities:**
    - Maps the incoming request to a specific controller based on defined rules.
    - Helps DispatcherServlet identify the appropriate controller to handle a request.

6. **ViewResolver:**

  - **Purpose:** Resolves logical view names to actual view implementations.
  - **Components:** Configured in the Spring configuration file.
  - **Responsibilities:**
    - Translates logical view names provided by the controller into actual view objects.
    - Helps DispatcherServlet locate the correct view for rendering.

7. **ModelAndView:**

  - **Purpose:** Represents both the model and the view.
  - **Components:** Returned by controller methods.
  - **Responsibilities:**
    - Contains data (model) that needs to be rendered.
    - Specifies the logical view name or actual view instance.

8. **Interceptor:**

  - **Purpose:** Provides a way to intercept the processing of a request by the controller.
  - **Components:** Spring MVC interceptor classes.
  - **Responsibilities:**
    - Allows pre-processing and post-processing of requests.
    - Can modify the model or view before rendering.

Annotations:

1. `@Component`

  - **Definition:** Indicates that a class is a Spring component.
  - **Example:**

    ```
    @Component
    public class MyComponent {
        // Class definition
    }
    ```

2. **@Service**

- ○ **Definition:** Specialized version of @Component for marking a class as a service component.
- ○ **Example:**

```
@Service
public class MyService {
    // Class definition
}
```

3. **@Repository**

- ○ **Definition:** Specialized version of @Component for marking a class as a repository component.
- ○ **Example:**

```
@Repository
public class MyRepository {
    // Class definition
}
```

4. **@Controller**

- ○ **Definition:** Specialized version of @Component for marking a class as a controller component in Spring MVC.
- ○ **Example:**

```
@Controller
public class MyController {
    // Class definition
}
```

5. **@Configuration**

- ○ **Definition:** Indicates that a class declares one or more @Bean methods.
- ○ **Example:**

```
@Configuration
public class AppConfig {
    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

6. `@Autowired`

- ○ **Definition:** Marks a constructor, field, method, or configuration method to be autowired by Spring.
- ○ **Example:**

```java
@Service
public class MyService {
    private final MyRepository myRepository;

    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

7. `@Qualifier`

- ○ **Definition:** Used in conjunction with `@Autowired` to specify the bean to be autowired when multiple beans of the same type exist.
- ○ **Example:**

```java
@Service
public class MyService {
    @Autowired
    @Qualifier("specificBean")
    private MyBean myBean;
}
```

8. `@Value`

- ○ **Definition:** Injects values from properties files or other sources into fields.
- ○ **Example:**

```java
@Component
public class MyComponent {
    @Value("${my.property}")
    private String myProperty;
}
```

9. `@Scope`

- ○ **Definition:** Specifies the scope of a bean (e.g., singleton, prototype).
- ○ **Example:**

```
@Component
@Scope("prototype")
public class MyPrototypeBean {
    // Class definition
}
```

10. **@PostConstruct**

    ○ **Definition:** Indicates a method to be invoked after a bean has been constructed and configured.
    ○ **Example:**

```
@Component
public class MyComponent {
    @PostConstruct
    public void init() {
        // Initialization logic
    }
}
```

11. **@PreDestroy**

    ○ **Definition:** Indicates a method to be invoked before a bean is destroyed.
    ○ **Example:**

```
@Component
public class MyComponent {
    @PreDestroy
    public void cleanup() {
        //Cleanup logic
    }
}
```

12. **@RequestMapping**

    ○ **Definition:** Maps HTTP requests to handler methods in Spring MVC controllers.
    ○ **Example:**

```
@Controller
@RequestMapping("/my")
public class MyController {
    @GetMapping("/endpoint")
    public String handleRequest() {
        return "myPage";
    }
}
```

13. **@ResponseBody**

   ○ **Definition:** Indicates that a method return value should be bound to the web response body.
   ○ **Example:**

```java
@RestController
public class MyRestController {
    @GetMapping("/api/data")
    @ResponseBody
    public String getData() {
        return "Some data";
    }
}
```

14. **@PathVariable**

   ○ **Definition:** Binds a method parameter to a URI template variable in Spring MVC.
   ○ **Example:**

```java
@Controller
public class MyController {
    @GetMapping("/user/{id}")
    public String getUser(@PathVariable Long id, Model model) {
        // Logic to retrieve and display user with the specified id
        return "userDetails";
    }
}
```

15. **@Transactional**

   ○ **Definition:** Specifies that a method is transactional and should be wrapped with a transaction.
   ○ **Example:**

```java
@Service
public class MyTransactionalService {
    @Transactional
    public void performTransactionalOperation() {
        // Transactional logic
    }
}
```

16. **@Query:**

   • @Query annotation in Spring Data JPA allows the definition of custom queries using JPQL or native SQL.

```
@Query("SELECT u FROM User u WHERE u.email = :email")
User findByEmail(@Param("email") String email);
```

AOP Terminologies:

1. **Aspect:**

   - **Description:** A module that encapsulates cross-cutting concerns, providing a set of APIs. For example, logging can be an aspect.
   - **Example:** LoggingAspect is an aspect that provides logging functionalities.

2. **Join Point:**

   - **Description:** A point in the application where the AOP aspect can be applied. It represents the actual place in the application where an action will be taken using Spring AOP.
   - **Example:** Method execution in a class can be a join point.

3. **Advice:**

   - **Description:** The actual action to be taken either before or after the method execution. It is a piece of code invoked during program execution by the Spring AOP framework.
   - **Example:** A piece of code that logs method execution details.

4. **Pointcut:**

   - **Description:** A set of one or more join points where an advice should be executed. It can be specified using expressions or patterns.
   - **Example:** A pointcut that selects all methods in a class.

5. **Introduction:**

   - **Description:** Allows adding new methods or attributes to existing classes.
   - **Example:** Introducing new methods to existing classes for additional functionality.

6. **Target Object:**

   - **Description:** The object being advised by one or more aspects. It is the proxied object or the advised object.
   - **Example:** An instance of a class where advice is applied.

7. **Weaving:**

   - **Description:** The process of linking aspects with other application types or objects to create an advised object. It can occur at compile time, load time, or runtime.
   - **Example:** Combining aspects with application code during compilation.

Types of Advice:

1. **Before:**

   - Run advice before the method execution.

2. **After:**

   - Run advice after the method execution, regardless of its outcome.

3. **After-Returning:**

   - Run advice after the method execution only if the method completes successfully.

4. **After-Throwing:**

   - Run advice after the method execution only if the method exits by throwing an exception.

5. **Around:**

   - Run advice before and after the advised method is invoked, allowing manipulation of the method call.

IOC Container in Spring:

1. **Definition:**

   - IOC Container is a core concept in the Spring Framework that manages the lifecycle of objects (beans) and is responsible for creating, configuring, and wiring them together.
   - It follows the Inversion of Control principle, where the control over object creation and lifecycle is inverted from the application code to the container.

2. **Responsibilities:**

   - **Instantiation:** The container is responsible for creating instances of beans.
   - **Configuration:** It configures the properties and dependencies of beans.
   - **Lifecycle Management:** Manages the complete lifecycle of beans, including their initialization, usage, and destruction.
   - **Dependency Injection:** Injects dependencies into beans, achieving loose coupling.

3. **Types of IOC Container:**

   a. **BeanFactory:**

   - **Description:** The simplest IOC container in Spring. BeanFactory is the simplest IOC container in Spring. It provides basic functionalities for managing beans.
   - **Features:**
       - Lazy loading of beans (creates a bean when requested).
       - Lightweight and suitable for resource-constrained environments.
   - **Usage:**
       - Often used in scenarios where resource consumption needs to be minimized.

   b. **ApplicationContext:**

   - **Description:** An advanced IOC container building on top of BeanFactory.
   - **Features:**
       - Eager loading of beans (creates and configures beans at container startup).
       - Provides additional features like event propagation, AOP integration, and more.

- Suitable for most applications due to its rich feature set.
  - **Types of ApplicationContext:**
    - **ClassPathXmlApplicationContext:** Loads context configuration from XML files located in the classpath.
    - **FileSystemXmlApplicationContext:** Similar to ClassPathXmlApplicationContext but loads XML files from an absolute path.
    - **AnnotationConfigApplicationContext:** Uses Java-based configuration classes annotated with @Configuration.
    - **GenericWebApplicationContext:** Web-aware ApplicationContext suitable for web applications.
  - **Usage:**
    - Recommended for most applications due to its extended capabilities.

Example Usage:

1. **Bean Definition:**

```java
public class MyService {
    // Class definition
}
```

2. **Configuration Metadata (XML-based):**

```xml
<!-- applicationContext.xml -->
<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">

    <bean id="myService" class="com.example.MyService"/>
</beans>
```

3. **ApplicationContext Usage:**

```java
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
MyService myService = context.getBean("myService", MyService.class);
// Use the MyService bean
```

Types of Bean Scopes:

Bean Scope in Spring:

In the Spring framework, the scope of a bean defines the lifecycle and visibility of the bean within the Spring IoC (Inversion of Control) container. The scope determines how the container should manage and provide

instances of a bean. Spring provides several bean scopes, each serving a specific purpose in different scenarios.

1. **Singleton:**

   - **Definition:** In Singleton scope, the Spring IoC container creates and manages a single instance of the bean for the entire application.
   - **Usage:** It is the default scope. When a bean is declared without specifying a scope, it is assumed to be a singleton.
   - **Example Configuration:**

     ```
     <bean id="myBean" class="com.example.MyBean" />
     ```

2. **Prototype:**

   - **Definition:** In Prototype scope, a new instance of the bean is created each time it is requested from the container.
   - **Usage:** Use this scope when you want a new instance of the bean every time it is needed.
   - **Example Configuration:**

     ```
     <bean id="myBean" class="com.example.MyBean" scope="prototype" />
     ```

3. **Request:**

   - **Definition:** In Request scope, a new instance of the bean is created for every HTTP request.
   - **Usage:** Applicable in web applications to have a separate instance for each request.
   - **Example Configuration:**

     ```
     <bean id="myBean" class="com.example.MyBean" scope="request" />
     ```

4. **Session:**

   - **Definition:** In Session scope, a new instance of the bean is created for every new HttpSession.
   - **Usage:** Useful for maintaining stateful information across multiple requests in a web session.
   - **Example Configuration:**

     ```
     <bean id="myBean" class="com.example.MyBean" scope="session" />
     ```

5. **Global Session:**

   - **Definition:** Similar to Session scope, but specifically used in a Portlet environment for global sessions across multiple portlets.
   - **Usage:** Applicable in a Portlet environment where the concept of a global session is used.

- ○ **Example Configuration:**

```
<bean id="myBean" class="com.example.MyBean" scope="globalSession" />
```

The limitations of Cookies are:

- 20 cookies per site each of max 4Kb

load-on-startup :

The element in the servlet configuration within the web.xml file is used to ensure that a servlet is loaded and initialized during the application startup, even before the first request is received. This can be beneficial in scenarios where you want to perform certain initialization tasks or ensure that the servlet is ready to handle requests immediately after the application starts.

SpringBoot :

- SOAP (Simple Object Access Protocol): Protocol for exchanging structured information in web services.
- REST (Representational State Transfer): Architectural style for designing networked applications.

Certainly, I'll provide a brief overview of each of the mentioned topics:

1. Why Spring Boot:

Definition: Spring Boot is a framework developed by Pivotal that simplifies the process of building production-ready applications with the Spring Framework. It provides an opinionated approach to application configuration, eliminates boilerplate code, and allows developers to create standalone, production-grade Spring-based Applications.

Key Features:

- Simplified configuration.
- Embeds a lightweight server (Tomcat, Jetty, or Undertow).
- Automatic dependency management.
- Production-ready defaults for application settings.
- A wide range of plugins for commonly used tasks.
- Spring Boot Starter Projects for various application types.

2. Spring Boot Overview:

Definition: Spring Boot is a part of the Spring Framework and focuses on making it easier to build Spring-powered applications. It provides a set of conventions for building production-ready applications with minimal effort.

Key Concepts:

- **Opinionated Defaults:** Spring Boot provides defaults for configuration settings, reducing the need for explicit configuration.

- **Standalone:** Spring Boot applications can be run as standalone JARs, simplifying deployment.
- **Auto-Configuration:** Automatically configures components based on project dependencies.
- **Spring Boot Starters:** Pre-configured dependencies for specific types of applications.
- **Spring Boot Actuator:** Provides production-ready features like health checks, metrics, etc.

3. Basic Introduction of MAVEN:

Definition: Apache Maven is a powerful project management tool that is used for project build automation, dependency management, and project documentation. It simplifies the build process and manages project dependencies.

Key Concepts:

- **Project Object Model (POM):** Describes the configuration and dependencies of a project.
- **Lifecycle:** Defines a sequence of phases for building and distributing the project.
- **Plugins:** Enhance Maven's capabilities for specific tasks.
- **Dependencies:** External libraries required for the project.
- **Repository:** Stores and manages project artifacts.

4. Building Spring Web Application with Boot Spring Data JPA:

Definition: Spring Data JPA simplifies data access using the Java Persistence API (JPA). It is part of the larger Spring Data project, which aims to provide a consistent and familiar programming model for various data stores.

Key Concepts:

- **Entity:** A JPA annotated class representing a persistent entity.
- **Repository:** An interface that provides CRUD operations on entities.
- **CrudRepository & JpaRepository:** Interfaces provided by Spring Data JPA for basic CRUD operations.
- **Query Methods:** Derived queries based on method names.

5. CRUD Repository & JPA Repository:

Definition:

- **CrudRepository:** A Spring Data interface providing CRUD operations for an entity.
- **JpaRepository:** Extends CrudRepository with additional JPA-specific features.

Key Concepts:

- **Create:** `save(entity)` method.
- **Read:** `findById(id)`, `findAll()`, etc.
- **Update:** `save(entity)` method.
- **Delete:** `delete(entity)` method.

6. Query Methods:

Definition: Query Methods in Spring Data JPA allow the definition of queries by method names. Spring Data JPA derives the query from the method name.

Example:

```java
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastName(String lastName);
}
```

7. Using Custom Query (@Query):

Definition: @Query annotation in Spring Data JPA allows the definition of custom queries using JPQL or native SQL.

Example:

```java
@Query("SELECT u FROM User u WHERE u.email = :email")
User findByEmail(@Param("email") String email);
```

8. Introduction to Web Services:

Definition: Web services are a technology stack for building distributed systems, allowing applications to communicate over a network.

Types:

- **SOAP (Simple Object Access Protocol):** Protocol for exchanging structured information in web services.
- **REST (Representational State Transfer):** Architectural style for designing networked applications.

9. SOAP Vs RESTful Web Services:

Differences:

- **Protocol:** SOAP uses XML-based messaging, while REST uses various formats, typically JSON.
- **Flexibility:** REST is more flexible and simpler than SOAP.
- **State:** REST is stateless, while SOAP can be stateful.
- **Usage:** SOAP is often used in enterprise-level applications, while REST is widely used in web applications.

10. Create RESTful Web Service in Java using Spring Boot:

Example:

```java
@RestController
@RequestMapping("/api")
public class UserController {

    @GetMapping("/users")
    public List<User> getAllUsers() {
```

```
        // Retrieve and return all users
    }

    @PostMapping("/users")
    public ResponseEntity<User> createUser(@RequestBody User user) {
        // Create and return a new user
    }
}
```

11. RESTful Web Service JSON Example:

Example:

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com"
}
```

12. RESTful Web Service CRUD Example:

Example:

```
// Controller methods for CRUD operations
```

13. Using POSTMAN Client to Invoke REST APIs:

Definition: Postman is a popular API client that simplifies the process of developing, testing, and documenting APIs.

Usage:

- Design and test APIs.
- Create and send HTTP requests.
- Inspect responses.

14. REST Service Invocation using REST Template:

Definition: `RestTemplate` is a Spring class that simplifies communication with RESTful services.

Example:

```
RestTemplate restTemplate = new RestTemplate();
User user = restTemplate.getForObject("https://api.example.com/users/1",
User.class);
```

Spring Stereotype Annotations:

- @Component: Indicates that a class is a Spring component and should be automatically detected and registered as a bean.
- @Service: Similar to @Component, specifically used for service layer classes.
- @Repository: Indicates that a class is a Spring Data repository, typically used for database access.
- @Controller: Marks a class as a Spring MVC controller.

```
Example:
```java
@Component
public class MyComponent {
    // Class implementation
}
```
```

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays   Group Link: https://t.me/ccee2025notes

+91 8007592194   +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project