JAVA EE (JAKARTA EE)

INDICE:

1 Introducción:

Java EE es un conjunto de especificaciones para crear aplicaciones empresariales que sean escalables, seguras y rápidas. Utiliza herramientas y servicios estándar para ayudar a crear y administrar aplicaciones web, manejar transacciones y conectar diferentes sistemas informáticos.

En 2019, la organización que gestiona Java EE lo entregó a la Fundación Eclipse, que ahora se conoce como Jakarta EE.

1.1 Características Principales de Java EE

- **Arquitectura Multicapas:** Permite dividir las aplicaciones en varias capas (presentación, lógica de negocio y datos), lo que facilita el mantenimiento y la escalabilidad.
- **Portabilidad y Estándar Abierto:** Las aplicaciones Java EE pueden ejecutarse en diferentes servidores de aplicaciones compatibles (como WildFly, GlassFish o WebLogic) sin cambios significativos en el código.
- **Componentes Modulares:** Java EE ofrece una variedad de componentes listos para usar, como **servlets**, **Enterprise JavaBeans (EJB)** y **JSP**, que simplifican el desarrollo.
- **Seguridad Empresarial:** Java EE proporciona APIs estándar para gestión de autenticación, autorización y control de acceso (como **Java Authentication and Authorization Service, JAAS**).

1.2 Arquitectura

- **Capa de presentación**:
    - **JavaServer Pages (JSP)** y **JavaServer Faces (JSF)** permiten crear interfaces web dinámicas.
    - **Servlets** gestionan solicitudes HTTP y actúan como controladores en aplicaciones web.
- **Capa de negocio**:
    - **Enterprise JavaBeans (EJB)** proporciona lógica de negocio transaccional y distribuida.
    - **Context and Dependency Injection (CDI)** gestiona la inyección de dependencias y facilita el desarrollo de aplicaciones desacopladas.
- **Capa de integración**:

    - **Java Persistence API (JPA)** facilita el mapeo objeto-relacional para interactuar con bases de datos.
    - **Java Message Service (JMS)** proporciona mensajería asíncrona entre sistemas.
    - **Java Transaction API (JTA)** gestiona transacciones distribuidas.
- **Servicios web**:
    - **JAX-RS** para construir servicios RESTful.
    - **JAX-WS** para servicios web basados en SOAP.

1.3 Ventajas de Java EE

- **Escalabilidad**: Soporta aplicaciones desde pequeñas hasta sistemas empresariales distribuidos.
- **Modularidad**: La arquitectura modular permite desarrollar, probar y desplegar componentes de forma independiente.
- **Comunidad y Estándares**: Java EE tiene una amplia comunidad de desarrolladores y sigue estándares abiertos.
- **Integración Empresarial**: APIs como JTA y JMS facilitan la integración con otros sistemas.

1.4 Desventajas de Java EE

- **Curva de Aprendizaje**: Puede ser complejo para nuevos desarrolladores por la cantidad de APIs y estándares.
- **Sobrecarga**: Para proyectos pequeños, usar toda la plataforma Java EE puede ser excesivo.
- **Transición a Jakarta EE**: La migración de Java EE a Jakarta EE ha requerido algunos ajustes en los nombres de los paquetes.

2 Herramientas de desarrollo

2.1 Entornos de desarrollo integrado (IDE)

- **IntelliJ IDEA**: Popular por su velocidad, soporte de refactorización avanzada y desarrollo inteligente.
- **Eclipse**: Gratuito y extensible, con gran soporte para Java EE, Spring, y herramientas de depuración.
- **NetBeans**: IDE oficial de Oracle (ahora de Apache), con soporte completo para Java SE y EE.

2.2 Build Tools (Herramientas de construcción)

Facilitan la compilación, pruebas, y empaquetado de aplicaciones.

- **Maven**: Administra dependencias y permite construir aplicaciones mediante archivos POM.
- **Gradle**: Más moderno y flexible que Maven, especialmente útil para proyectos grandes o modulares.
- **Ant**: Más antiguo, permite construir proyectos a través de scripts XML personalizados, aunque está siendo menos utilizado hoy en día.

2.3 Pruebas y depuración

- **JUnit**: Framework estándar para pruebas unitarias.
- **Mockito**: Framework para crear pruebas usando *mocks* (dobles de prueba).
- **Selenium**: Automatización de pruebas para aplicaciones web.
- **SonarQube**: Herramienta de análisis estático para detectar problemas en el código (bugs, vulnerabilidades, etc.).

2.4 Frameworks y librerías

- **Spring Framework**: Muy utilizado para aplicaciones empresariales. Su módulo **Spring Boot** facilita la creación de aplicaciones rápidamente.
- **Hibernate**: Framework ORM (Object-Relational Mapping) que simplifica la interacción con bases de datos.
- **Jakarta EE** (antes Java EE): Conjunto de especificaciones para aplicaciones empresariales, ahora mantenido por Eclipse Foundation.

3 Servidores de aplicaciones en entornos Java

Un servidor de aplicaciones proporciona un entorno para ejecutar aplicaciones empresariales, gestionando la seguridad, la escalabilidad y la conexión con bases de datos.

3.1 Principales servidores de aplicaciones Java

- **Apache Tomcat**:
    - Ligero y rápido.
    - Soporta las especificaciones de **Servlets** y **JSP** (pero no completamente Java EE o Jakarta EE).
    - Ideal para aplicaciones web más sencillas.
- **WildFly / JBoss**:
    - Proyecto mantenido por Red Hat.

    - Soporta Java EE/Jakarta EE completo.
    - Es flexible, permitiendo ejecutar tanto aplicaciones tradicionales como microservicios.
- **GlassFish**:
    - Servidor de referencia oficial de Jakarta EE.
    - Usado principalmente para pruebas y demostraciones, pero menos popular en producción.
- **WebLogic**:
    - Mantenido por Oracle, enfocado en aplicaciones empresariales críticas.
    - Muy utilizado en entornos corporativos por su integración con Oracle Database y soporte robusto.
- **WebSphere**:
    - Servidor de IBM orientado a aplicaciones empresariales grandes.
    - Ideal para escenarios de alta disponibilidad y escalabilidad.

3.2 Gestión de microservicios y contenedores

Con la tendencia hacia microservicios, se usan herramientas adicionales para desplegar aplicaciones más modulares.

- **Docker**: Contenedores ligeros que empaquetan aplicaciones con todas sus dependencias.
- **Kubernetes**: Orquesta y gestiona múltiples contenedores.
- **Spring Cloud**: Extiende Spring Boot para construir microservicios, con soporte para gestión de configuración y descubrimiento de servicios.

3.3 Herramientas de automatización e integración continua

- **Jenkins**: Automatiza la construcción, pruebas y despliegue de código.
- **GitLab CI/CD**: Sistema de integración y entrega continua embebido en GitLab.
- **Apache Kafka**: Mensajería distribuida, útil en arquitecturas de microservicios.

4 JavaBeans

JavaBeans es una tecnología Java para definir componentes de software reutilizables. Un JavaBean es una clase Java especial que sigue ciertas convenciones para permitir la encapsulación, la manipulación de herramientas de desarrollo y la reutilización de código. Los JavaBeans son particularmente útiles para modularizar el procesamiento de datos y la lógica empresarial en aplicaciones Java SE y Java EE.

4.1 Características de un JavaBean

Para que una clase sea considerada un **JavaBean**, debe cumplir con los siguientes requisitos o convenciones:

1. **Constructor por defecto sin parámetros**
    - La clase debe tener un constructor público sin parámetros para que pueda ser instanciada fácilmente por otras herramientas.
2. **Propiedades accesibles mediante getters y setters**

    Las propiedades privadas de la clase se deben manipular a través de métodos públicos **getters** y **setters**.
    Por ejemplo:

```
public class Person {

   private String name;


   public String getName() {

      return name;

   }


   public void setName(String name) {

      this.name = name;

   }

}
```

3. **Serialización**

    La clase debe implementar la interfaz Serializable para que pueda ser guardada en disco o transmitida a través de la red.

    ```
    public class Person implements java.io.Serializable {}
    ```

4. **Encapsulación y modularidad**
    - Las propiedades son privadas y se exponen mediante métodos públicos, lo que permite mantener un diseño desacoplado.

4.2 Uso de JavaBeans

1. **Manipulación de datos (DTOs o POJOs)**

JavaBeans se utilizan para almacenar datos de forma estructurada. En aplicaciones Java EE, se usan como **Data Transfer Objects (DTOs)** para transferir información entre capas (por ejemplo, entre el backend y la capa de presentación).

Ejemplo de JavaBean simple:

```
public class Product implements java.io.Serializable {
    private String name;
    private double price;

    public Product() {}  // Constructor por defecto

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}
```

2. **Desarrollo Visual en IDEs**

- Los JavaBeans pueden ser manipulados por herramientas de desarrollo visual como **NetBeans** o **Eclipse**. Estas herramientas pueden detectar automáticamente las propiedades del JavaBean y permitir a los desarrolladores configurar su comportamiento sin escribir código adicional.

3. **Uso en JSPs (JavaServer Pages)**

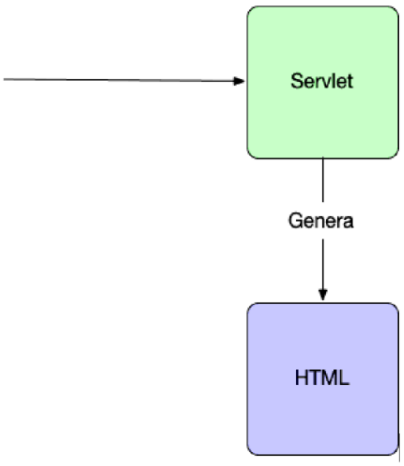
- En aplicaciones web basadas en **JSP**, los JavaBeans se usan para **almacenar y transferir datos** entre el servidor y la interfaz web.
  Ejemplo de uso en JSP:

```
<jsp:useBean id="product" class="Product” scope="session"/>
<jsp:setProperty name="product" property="name" value="Laptop"/>
<jsp:getProperty name="product" property="name"/>
```

5 Servlet

5.1 Introducción al Servlet

Un Servlet es el concepto más básico de componente web a nivel de Java EE. Un **Servlet** es un componente Java que se ejecuta en el servidor para procesar **solicitudes HTTP** y generar **respuestas dinámicas**. Es una de las tecnologías base en el desarrollo de aplicaciones web Java y constituye el núcleo de frameworks más avanzados como **JSP (JavaServer Pages)** y **Spring MVC**. En esencia se trata de una clase que genera una página HTML.



Los servlets permiten que un servidor web maneje solicitudes desde el cliente (generalmente navegadores web) y generen respuestas en diferentes formatos, como HTML o JSON. Aunque los servlets pueden responder a cualquier tipo de solicitudes, estos son utilizados comúnmente para extender las aplicaciones alojadas por servidores web, de tal manera que pueden ser vistos como applets* de Java que se ejecutan en servidores en vez de navegadores web. Este tipo de servlets son la contraparte Java de otras tecnologías de contenido dinámico Web, como PHP y ASP.NET.
Aunque es una tecnología bastante antigua y en gran medida desfasada, sigue siendo un componente clave en el desarrollo de aplicaciones web con Java dado que son la base sobre la que se construyen tecnologías más avanzadas como JSP o Spring MVC.

** Componente de una aplicación que se ejecuta en el contexto de otro programa, por ejemplo, en un navegador web.*

5.2 Ciclo de vida de un Servlet

Uno de los conceptos más importantes que debemos comprender para manejar bien un Servlet es su ciclo de vida. El ciclo de vida de un framework o aplicación, es el conjunto de fases o estados desde que se inicializa el proceso hasta que este se destruye. En el caso del Servlet, el ciclo de vida es gestionado por el **contenedor web** (por ejemplo, Apache Tomcat). Involucra los siguientes pasos:

1. **Carga y creación del Servlet**
    - El contenedor web **carga** la clase del servlet la primera vez que recibe una solicitud (o al iniciar la aplicación si está configurado en modo load-on-startup).
    - Se crea una única instancia del servlet para atender múltiples solicitudes simultáneamente (**singleton**).
2. **Inicialización** (init())
    - El método init() se ejecuta una vez, al momento de la creación del servlet, para realizar tareas de inicialización (como la configuración de recursos).
3. **Procesamiento de solicitudes** (service())
    - El contenedor llama a service() cada vez que recibe una solicitud. Este método delega la llamada a los métodos específicos de **HTTP** (doGet(), doPost(), doPut(), etc.).
4. **Destrucción** (destroy())
    - Cuando el contenedor deja de necesitar el servlet (por ejemplo, al apagar el servidor), llama a destroy() para liberar recursos.

5.3 Métodos principales de un Servlet

- **init(ServletConfig config):** Este método se ejecuta una vez para inicializar el servlet.

```
@Override
public void init() throws ServletException {
        System.out.println("Servlet inicializado");
}
```

- **service(HttpServletRequest req, HttpServletResponse res):** Gestiona todas las solicitudes HTTP y delega la llamada a los métodos específicos (doGet(), doPost(), etc.).
- **doGet(HttpServletRequest req, HttpServletResponse res):** Se llama cuando la solicitud es de tipo **GET**.

```
@Override
protected void doGet (HttpServletRequest req,
HttpServletResponse res) throws ServletException, IOException {
```

```
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<h1>Hello from doGet!</h1>");
    }
```

- **doPost(HttpServletRequest req, HttpServletResponse res):** Se usa para manejar solicitudes **POST**, normalmente enviadas desde formularios HTML.

```
@Override
protected void doPost(HttpServletRequest req,
HttpServletResponse res) throws ServletException, IOException {
    String name = req.getParameter("name");
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<h1>Welcome, " + name + "!</h1>");
}
```

- **destroy():** Este método se ejecuta al finalizar el ciclo de vida del servlet y libera recursos.

```
@Override
public void destroy() {
    System.out.println("Servlet destruido");
}
```

5.4 Configuración de un Servlet

1. **Usando anotaciones:** Desde Java EE 6, se pueden registrar servlets con anotaciones en lugar del archivo web.xml.

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
@WebServlet(name = "HelloServlet", urlPatterns = {"/hello"})
public class HelloServlet extends HttpServlet {
    // Implementación de métodos doGet/doPost
}
```

2. **Usando web.xml:** También puedes registrar el servlet en el archivo web.xml de la aplicación.

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>com.example.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

5.5 Cómo los Servlets Procesan las Solicitudes HTTP

Los servlets utilizan las interfaces HttpServletRequest y HttpServletResponse para manejar solicitudes y generar respuestas.

- **HttpServletRequest:** Contiene toda la información de la solicitud, como parámetros, encabezados y datos del cuerpo.
- **HttpServletResponse:** Permite enviar respuestas HTTP con contenido dinámico y establecer códigos de estado y encabezados HTTP.

5.6 Manejo de Parámetros

1. **Parámetros desde la URL (Query Params):** Si accedemos a una URL como http://localhost:8080/hello?name=Eustaquio , podemos capturar el parámetro name así:

```
String name = request.getParameter("name");
```

2. **Parámetros desde un formulario HTML**
   Formulario HTML:

```
<form action="hello" method="POST">
    <input type="text" name="name" placeholder="Enter your name">
    <button type="submit">Submit</button>
</form>
```

   Servlet:

```
@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    String name = request.getParameter("name");
    response.setContentType("text/html");
    response.getWriter().println("<h1>Hello, " + name +
    "!</h1>");
}
```

5.7 Redirección y Reenvío de Solicitudes

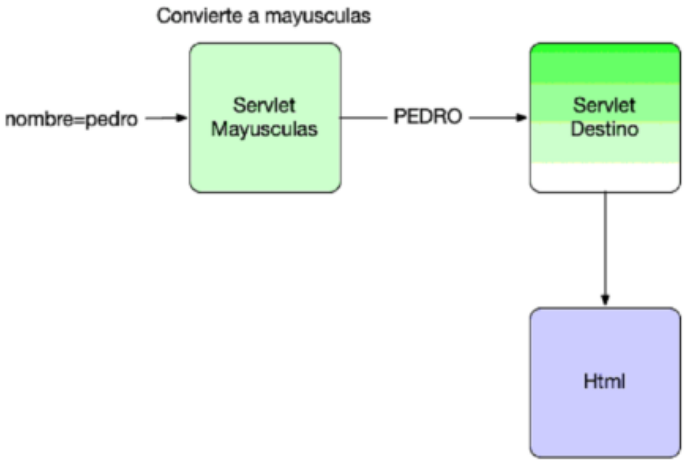
- **Redirección:** Envía al cliente a otra URL.

```
response.sendRedirect("https://www.example.com");
```

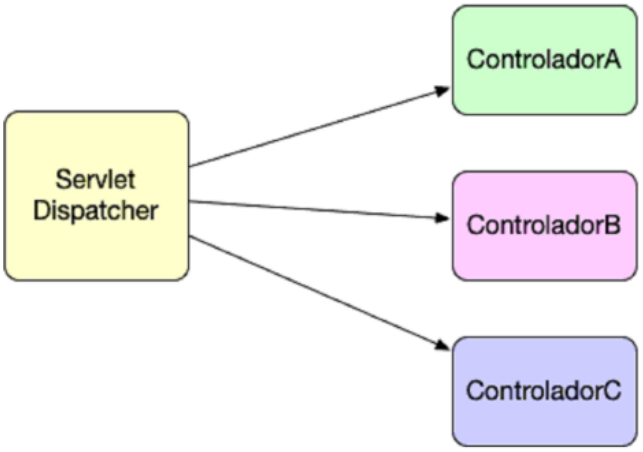
- **Reenvío interno:** Pasa la solicitud a otro Servlet o recurso dentro de la misma aplicación.

```
RequestDispatcher dispatcher = request.
                                    getRequestDispatcher(
                                        "/otherServlet"
```

```
                                                    );
dispatcher.forward(request, response);
```



- **FrontController**: Puede parecer poco importante este sistema de delegación pero es la base de todos los frameworks modernos web de Java ya que por ejemplo Spring MVC usa un Servlet Principal o FrontController para recibir todas las peticiones web y redirigirlas a cada una de los controladores que tiene .Este Servlet se denomina DispacherServlet a nivel de Spring.



5.8 Práctica

5.8.1 Requisitos previos

1. NetBeans IDE instalado
2. Apache Tomcat configurado como servidor en NetBeans
3. JDK instalado

5.8.2 Paso a paso

- **Paso 1: Crear proyecto Web en NetBeans**
  1. Abre NetBeans.
  2. Ve a **File → New Project**.

3. Selecciona **Java with Maven** → **Web Application** y haz click en **Next**.
4. Asigna un nombre al proyecto (por ejemplo, ServletExample) y selecciona **Apache Tomcat** como servidor.
5. Haz click en **Finish**.

- **Paso 2: Agregar el Servlet al Proyecto**
  1. Haz click derecho sobre el proyecto en el panel **Projects**.
  2. Selecciona **New** → **Servlet**.
  3. Asigna un nombre al Servlet, por ejemplo, HelloServlet, y haz click en **Next**.
  4. Define la **URL Mapping** como /hello y haz click en **Finish**.
- **Paso 3: Editar el código del Servlet**

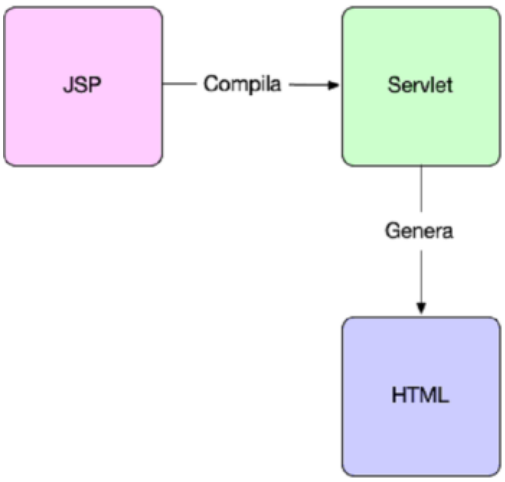
```java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = {"/hello"})
public class HelloServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head><title>Hello Servlet</title></head>");
        out.println("<body>");
        out.println("<h1>Hello, World!</h1>");
        out.println("<p>Este es un ejemplo de servlet usando NetBeans y Apache.</p>");
        out.println("</body>");
        out.println("</html>");
    }

}
```

- **Paso 4: Configurar el Servidor y Ejecutar el Proyecto**
  1. Haz click derecho sobre el proyecto y selecciona **Run**.
  2. NetBeans iniciará el servidor **Apache Tomcat** y desplegará tu aplicación.
  3. Abre tu navegador y accede a la URL: http://localhost:8080/ServletExample/hello

6 JSP

6.1 Introducción a JSP

JavaServer Pages (JSP) es una tecnología de Java para el desarrollo de **aplicaciones web dinámicas**. Permite crear páginas web con contenido dinámico mezclando **código HTML, CSS, JavaScript** y **Java**. JSP es parte de la especificación **Java EE** y se compila en **servlets** por el contenedor web, facilitando la generación de contenido dinámico en respuesta a solicitudes HTTP.



6.2 Características Principales de JSP

1. **Combinación de HTML y Java**: Permite insertar fragmentos de código Java directamente dentro de páginas HTML.
2. **Compilación en Servlets**: Cada página JSP se convierte en un servlet al ser desplegada en el servidor.
3. **Separación de Presentación y Lógica de Negocio**: Facilita la separación del diseño (HTML) de la lógica empresarial.
4. **Integración con Servlets**: JSP es ideal para el desarrollo en combinación con **Servlets**, donde estos manejan la lógica y JSP genera la vista.
5. **Soporte de etiquetas personalizadas**: A través de **JSTL** (JSP Standard Tag Library) y **Tag Libraries**, se puede extender JSP sin necesidad de mucho código Java.

6.3 Ciclo de Vida de una Página JSP

1. **Traducción**: El contenedor traduce la página JSP en un **servlet Java**.
2. **Compilación**: El servlet generado se compila en **bytecode**.
3. **Carga e Inicialización**: El servlet JSP se carga y se inicializa en el servidor.
4. **Procesamiento de solicitudes**: El servlet responde a las solicitudes HTTP.

5. **Destrucción**: Cuando el servidor se detiene o la aplicación se cierra, el servlet JSP es destruido.

6.4 Estructura Básica

Un archivo JSP tiene la extensión .jsp y puede contener:

- **Código HTML**.
- **Scriptlets** de Java.
- **Expresiones JSP** para mostrar datos.
- **Declaraciones** para definir métodos o variables.
- **Directivas JSP** para configurar la página.

Ejemplo

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %> <!DOCTYPE html>
<html>
    <head>
        <title>Hello JSP</title>
    </head>
    <body>
        <h1>Welcome to JSP!</h1>
         <% // Scriptlet: Bloque de código Java
            String name = "John Doe";
            out.println("<p>Hello, " + name + "!</p>");
        %>
        <p>Current date and time: <%= new java.util.Date() %></p>
    </body>
</html>
```

6.5 Elementos de JSP

1. **Directivas JSP:** Configuran cómo se comporta la página JSP.

   ```
   <%@ page import="java.util.*" %>
   <%@ include file="header.jsp" %>
   ```

   <%@ page %>: Configura la página (por ejemplo, codificación, imports).
   <%@ include %>: Incluye archivos JSP o HTML.
   <%@ taglib %>: Declara bibliotecas de etiquetas.

2. **Scriptlets (<% … %>):** Permiten escribir bloques de código Java dentro de HTML.

   ```
   <%
       String user = "Alice";
       out.println("<p>User: " + user + "</p>");
   %>
   ```

3. **Expresiones JSP (<%= … %>):** Muestran directamente el valor de una expresión en la página HTML.

```
<p>2 + 2 = <%= 2 + 2 %></p>
```

4. **Declaraciones JSP (<%! … %>):** Se usan para declarar variables o métodos que estarán disponibles para toda la página JSP.

```
<%!
   int sum(int a, int b) {
      return a + b;
   }
%>
<p>Sum: <%= sum(5, 3) %></p>
```

5. **Expression Language (${ … }):** Permite acceder de manera sencilla a objetos, datos y valores. Se aconseja su uso por encima de los Scriptlets.

```
<p>User name: ${user.name}</p>
```

6. **JSTL:** Es una biblioteca de etiquetas estándar que reemplaza los scriptlets y permite manipular datos con etiquetas XML-friendly.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
        <ul>
                <c:forEach var="item" items="${itemsList}">
                        <li>${item}</li>
                </c:forEach>
        </ul>
```

6.6 Manejo de Parámetros con JSP

Se pueden capturar parámetros enviados desde un formulario HTML o URL mediante request.getParameter().

Formulario HTML:

```
<form action="welcome.jsp" method="GET">
   <input type="text" name="username" placeholder="Enter your
name">
   <button type="submit">Submit</button>
</form>
```

JSP que procesa la solicitud:

```
<%
        String username = request.getParameter("username");
%>
<p>Hello, <%= username %>!</p>
```

6.7 Manejo/Inclusión de Archivos JSP

1. **Inclusión en tiempo de compilación (<%@ include %>):** El contenido del archivo se incluye al compilar la página.
   ```
   <%@ include file="header.jsp" %>
   ```
2. **Inclusión en tiempo de ejecución (<jsp:include />):** El contenido se incluye cada vez que la página es solicitada.
   ```
   <jsp:include page="footer.jsp" />
   ```

6.8 Manejo de Errores en JSP

Puedes definir páginas de error personalizadas en el archivo web.xml.

Configuración en web.xml:

```
<error-page>
  <error-code>404</error-code>
  <location>/error404.jsp</location>
</error-page>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/error.jsp</location>
</error-page>
```

Página de error personalizada (error.jsp):

```
<h1>Oops! Something went wrong.</h1>
<p>Please try again later.</p>
```

6.9 Práctica

6.9.1 Objetivo

Crear una aplicación sencilla en NetBeans que utilice JSP para mostrar un mensaje de bienvenida personalizado basado en un parámetro ingresado por el usuario.

6.9.2 Requisitos previos

1. NetBeans IDE instalado
2. Apache Tomcat configurado como servidor en NetBeans
3. JDK instalado

6.9.3 Paso a paso

- **Paso 1: Configuración inicial**
  1. **Abre NetBeans** y selecciona:
     File → New Project.

2. En el asistente de creación, selecciona:
    - **Categories** Java with Maven (Java with Ant es otra alternativa)
    - **Projects**: **Web Application**.
    - Haz click en **Next**.
3. **Nombre del Proyecto**:
   Ingresa un nombre como "**EjemploJSP**".
    - Ubicación: Elige una carpeta donde guardar el proyecto.
    - Haz click en **Next**.
4. **Servidor**:
    - Selecciona **Apache Tomcat** (debe estar instalado y configurado).
    - Versión: **Java EE 8 o superior**.
5. **Configuración del Framework**:
    - No necesitas seleccionar ningún framework adicional para este ejemplo.
    - Haz click en **Finish**.

- **Paso 2: Editar la página de inicio para solicitar los datos**
    1. Reemplaza el contenido del archivo "**index.html**" por el siguiente código

```
<!DOCTYPE html>
<html lang="en">
        <head>
                <meta charset="UTF-8">
                        <title>Formulario de Bienvenida</title>
                </head>
        <body>
                <h1>Bienvenido a la aplicación JSP</h1>
                <form action="bienvenida.jsp" method="GET">
                        <label for="nombre">Introduce tu
                nombre:</label>
                        <input type="text" id="nombre"
                name="nombre" required>
                        <button type="submit">Enviar</button>
                </form>
        </body>
</html>
```

- **Paso 3: Crear la Página JSP para mostrar el mensaje**
    1. En el panel **Projects**, haz click derecho sobre **Web Pages** → **New** → **JSP**.
    2. Asignar "**bienvenida.jsp**" como nombre de archivo.
    3. Reemplaza el contenido del archivo **bienvenida.jsp** con este código:

```
<%@ page contentType="text/html; charset=UTF-8"
language="java" %>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Página de Bienvenida</title>
</head>
    <body>
        <h1>Página de Bienvenida</h1>
        <% // Capturamos el parámetro 'nombre'
        enviado desde el formulario
        String nombre =
    request.getParameter("nombre");
        // Validamos si el nombre no es nulo ni vacío
            if (nombre != null &&
        !nombre.trim().isEmpty()) { %>
            <p>¡Hola, <strong><%= nombre %>
            </strong>! Bienvenido a nuestra
            aplicación JSP.</p>
        <% } else { %>
            <p>No ingresaste ningún nombre. Por
            favor, regresa al <a href="index.html">
            formulario </a> e intenta
            nuevamente.</p>
        <% } %>
    </body>
</html>
```

- **Paso 4: Ejecutar el proyecto**
    1. Haz click derecho sobre el nombre del proyecto y selecciona **Properties → Run.** Verifica que en el campo **Server** esté seleccionado “**Apache Tomcat**” y que el **Context Path** sea “**/EjemploJSP**”.
    2. Haz click derecho sobre el nombre del proyecto y selecciona **Run** (o presiona F6)

7 JSTL

7.1 Introducción a JSTL

JSTL (JavaServer Pages Standard Tag Library) es una biblioteca de etiquetas estándar para JSP que facilita la escritura de código. JSTL le permite realizar tareas comunes como iteración, operaciones condicionales, procesamiento de expresiones

y manipulación de datos directamente utilizando etiquetas en lugar de utilizar grandes cantidades de código Java embebido en páginas **JSP**.

JSTL mejora la claridad y la capacidad de mantenimiento del código JSP. , promoviendo una mayor separación entre la lógica empresarial y la presentación. Esto es similar a cómo funcionan los marcos **MVC** (Modelo-Vista-Controlador), donde intentan almacenar la lógica dentro de etiquetas en lugar de incrustar código Java.

7.2 Tipos de etiqueta

JSTL tiene varias librerías que se agrupan en función de su propósito. Las principales librerías son:

- **Core Tags (c:)**: Condiciones, iteraciones, y manejo de expresiones.
- **XML Tags (x:)**: Procesamiento de XML.
- **Formatting Tags (fmt:)**: Formateo de números, fechas y mensajes internacionales.
- **SQL Tags (sql:)**: Consultas y actualización de bases de datos.
- **Functions Tags (fn:)**: Funciones para manipulación de cadenas.

7.3 Tabla de etiquetas

Etiqueta	Uso	Librería	Ejemplo
<c:out>	Escapa y muestra el valor de una variable o expresión.	**Core**	**<c:out value="${user.name}" />**
<c:if>	Renderiza un bloque de código si se cumple una condición.	**Core**	**<c:if test="${user.age >= 18}">Adult</c:if>**
<c:choose>	Anida condiciones tipo if-else usando <c:when> y <c:otherwise>.	**Core**	**<c:choose><c:when test="${value == 1}">One</c:when><c:otherwise>Not One</c:otherwise></c:choose>**
<c:when>	Parte del bloque <c:choose> para definir condiciones.	**Core**	**<c:when test="${user.role == 'admin'}">Admin</c:when>**
<c:otherwise>	Se ejecuta si ninguna condición previa es verdadera dentro de <c:choose>.	**Core**	**<c:otherwise>Guest</c:otherwise>**
<c:forEach>	Itera sobre colecciones, listas o arrays.	**Core**	**<c:forEach var="item" items="${products}">...</c:forEach>**
<c:forTokens>	Itera sobre una cadena de texto separada por delimitadores (tokens).	**Core**	**<c:forTokens var="token" items="a,b,c" delims="," />**
<c:set>	Define una variable de contexto o establece su valor.	**Core**	**<c:set var="name" value="John Doe" />**

<c:remove>	Elimina una variable del contexto.	**Core**	**<c:remove var="name" />**
<c:redirect>	Redirige a una URL especificada.	**Core**	**<c:redirect url="anotherPage.jsp" />**
<c:url>	Construye URLs, manejando adecuadamente parámetros de contexto.	**Core**	**<c:url value="somePage.jsp"><c:param name="id" value="123" /></c:url>**
<c:catch>	Captura excepciones en el bloque de código.	**Core**	**<c:catch var="error"><c:throw/>Error Handling</c:catch>**
<fmt:formatNumber>	Formatea un número según el patrón especificado.	**Formatting**	**<fmt:formatNumber value="${amount}" pattern="#,##0.00" />**
<fmt:bundle>	Define un recurso de mensajes para la localización.	**Formatting**	**<fmt:bundle basename="messages" />**
<fmt:setBundle>	Establece un nuevo conjunto de recursos.	**Formatting**	**<fmt:setBundle basename="messages" />**
<fmt:setLocale>	Establece la configuración regional para la página.	**Formatting**	**<fmt:setLocale value="es_ES" />**
<fmt:setTimeZone>	Establece la zona horaria para la formateo de fechas.	**Formatting**	**<fmt:setTimeZone value="GMT" />**
<fmt:message>	Obtiene un mensaje localizado desde un archivo de recursos.	**Formatting**	**<fmt:message key="welcome.message" />**
<fmt:requestEncoding>	Establece la codificación de caracteres para la solicitud.	**Formatting**	**<fmt:requestEncoding value="UTF-8" />**
<fmt:setTimeZone>	Establece la zona horaria para el formato de fechas.	**Formatting**	**<fmt:setTimeZone value="GMT" />**
<sql:param>	Declara un parámetro para usar en las consultas SQL.	**SQL**	**<sql:param name="id" value="1"/>**
<sql:dateParam>	Se utiliza para pasar parámetros de tipo fecha en consultas SQL.	**SQL**	**<sql:dateParam name="startDate" value="${date}" />**
<sql:setDataSource>	Configura la fuente de datos para las operaciones SQL.	**SQL**	**<sql:setDataSource var="ds" driver="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/mydb" user="root" password="password"/>**
<sql:transaction>	Define una transacción SQL.	**SQL**	**<sql:transaction><sql:update>...</sql:update></sql:transaction>**
<x:transform>	Aplica una transformación XSLT a un documento XML.	**XML**	**<x:transform xml="${xmlDoc}" xslt="stylesheet.xsl" />**
<x:forEach>	Itera sobre nodos en un documento XML.	**XML**	**<x:forEach var="item" select="$doc/items/item" />**
<x:set>	Establece una variable en el contexto XML.	**XML**	**<x:set var="myVar" select="$doc/item" />**
<x:param>	Establece un parámetro para pasar a una transformación XSLT.	**XML**	**<x:param name="paramName" select="$value" />**
<x:if>	Condición para renderizar un bloque de	**XML**	**<x:if**

	código si se cumple.		**test="$condition">...</x:if>**
<fn:length>	Devuelve la longitud de una cadena o colección.	**Functions**	**${fn:length("Hello")}**
<fn:endsWith>	Devuelve true si una cadena termina con una subcadena específica.	**Functions**	**${fn:endsWith(name, "Doe")}**
<fn:indexOf>	Devuelve la posición de la primera aparición de una subcadena en una cadena.	**Functions**	**${fn:indexOf(name, "o")}**
<fn:split>	Divide una cadena en un arreglo usando un delimitador.	**Functions**	**${fn:split(name, ",")}**
<fn:replace>	Reemplaza partes de una cadena con otra.	**Functions**	**${fn:replace(name, "John", "Jane")}**
<fn:substring>	Devuelve una subcadena a partir de una cadena original.	**Functions**	**${fn:substring(name, 0, 4)}**
<fn:toLowerCase>	Convierte una cadena a minúsculas.	**Functions**	**${fn:toLowerCase(name)}**
<fn:toUpperCase>	Convierte una cadena a mayúsculas.	**Functions**	**${fn:toUpperCase(name)}**
<fn:trim>	Elimina espacios en blanco al principio y al final de una cadena.	**Functions**	**${fn:trim(name)}**
<fn:join>	Une elementos de un arreglo en una cadena, utilizando un delimitador.	**Functions**	**${fn:join(array, ", ")}**
<fn:containsIgnoreCase>	Devuelve true si una cadena contiene otra subcadena, ignorando el caso.	**Functions**	**${fn:containsIgnoreCase(name, "john")}**

7.4 Práctica

7.4.1 Objetivo

Crear una aplicación sencilla en la que mostraremos una serie de productos añadidos a una lista. Para ello crearemos la clase de entidad **Producto**, el servlet **ProductoServlet** y un **JSP** que usará las etiquetas **JSTL** para verificar si la lista contiene datos y, en caso de ser así, rellenar una tabla con dichos datos.

7.4.2 Requisitos Previos

1. NetBeans IDE instalado
2. Apache Tomcat configurado como servidor en NetBeans
3. JDK instalado
4. Asegúrate de que tu proyecto tiene el **JSTL** añadido como dependencia.
    - Si usas **Maven**, añade esto a tu pom.xml:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
```

```
</dependency>
```

- Si no usas Maven, descarga la librería JSTL (jstl.jar) y colócala en la carpeta **WEB-INF/lib** de tu proyecto.

7.4.3 Paso a paso:

- **Paso 1: Configuración inicial**
  1. **Abre NetBeans** y selecciona:
     File → New Project.
  2. En el asistente de creación, selecciona:
     - **Categories:** Java with Maven (Java with Ant es otra alternativa)
     - **Projects**: **Web Application**.
     - Haz click en **Next**.
  3. **Nombre del Proyecto**:
     Ingresa un nombre como "**EjemploJSTL**".
     - Ubicación: Elige una carpeta donde guardar el proyecto.
     - Haz click en **Next**.
  4. **Servidor**:
     - Selecciona **Apache Tomcat** (debe estar instalado y configurado).
     - Versión: **Java EE 8 o superior**.
  5. **Configuración del Framework**:
     - No necesitas seleccionar ningún framework adicional para este ejemplo.
     - Haz click en **Finish**.
- **Paso 2: Crear la clase de entidad**
  1. En el panel **Projects**, haz clic derecho sobre **Web Pages →** **New → Java Class**.
  2. Asignar como nombre "**Producto**"
  3. Indicar un paquete como ubicación en "**Package**"
  4. Haz click en **Finish**
  5. Añade este código al archivo:

```
public class Product {
    private int id;
    private String name;
    private double price;

    public Product(int id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public int getId() { return id; }
    public String getName() { return name; }
```

```
    public double getPrice() { return price; }
}
```

- **Paso 3: Crear el Servlet que manejará la lista de productos.**
  1. Haz click derecho sobre el proyecto en el panel **Projects**.
  2. Selecciona **New** → **Servlet**.
  3. Asigna como nombre “ProductoServlet”, y haz click en **Next**.
  4. Define la **URL Mapping** como /product y haz click en **Finish**.
  5. Añade este código al archivo:

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ProductoServlet extends HttpServlet {
   @Override
        protected void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      List<Product> products = new ArrayList<>();
      products.add(new Product(1, "Laptop", 800.0));
      products.add(new Product(2, "Smartphone", 500.0));
      products.add(new Product(3, "Tablet", 300.0));

      request.setAttribute("products", products);
                        RequestDispatcher dispatcher =
request.getRequestDispatcher("productos.jsp");
      dispatcher.forward(request, response);
   }
}
```

- **Paso 4: Crear el JSP con JSTL**
  1. En el panel **Projects**, haz click derecho sobre **Web Pages** → **New** → **JSP**.
  2. Asignar “**showProducts.jsp**” como nombre de archivo.
  3. Reemplaza el contenido del archivo **showProducts.jsp** con este código:

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

<html>
<head>
   <title>Lista de Productos</title>
```

```
</head>
<body>
   <h1>Lista de Productos</h1>

   <c:if test="${empty products}">
      <p>No hay productos disponibles.</p>
   </c:if>

   <c:if test="${not empty products}">
      <table border="1">
         <tr>
            <th>ID</th>
            <th>Nombre</th>
            <th>Precio</th>
         </tr>
         <c:forEach var="product" items="${products}">
            <tr>
               <td><c:out value="${product.id}" /></td>
               <td><c:out value="${product.name}" /></td>
               <td><c:out value="${product.price}" /></td>
            </tr>
         </c:forEach>
      </table>
   </c:if>
</body>
</html>
```

- **Paso 5: Ejecutar el código**
    1. Realizar las verificaciones pertinentes de que todo esta en funcionamiento y bien especificado.
    2. Haz click derecho sobre el nombre del proyecto y selecciona **Run** (o presiona F6).

8 JSF

8.1 Introducción a JSF

JavaServer Faces (JSF) es una tecnología y framework para aplicaciones Java basadas en web que simplifica el desarrollo de interfaces de usuario en aplicaciones Java EE. JSF usa JavaServer Pages (JSP) como la tecnología que permite hacer el despliegue de las páginas, pero también se puede acomodar a otras tecnologías

como XUL (acrónimo de XML-based User-interface Language, lenguaje basado en XML para la interfaz de usuario).

JSF proporciona un conjunto de **componentes reutilizables** de interfaz gráfica y facilita la vinculación entre el frontend y el backend.

JSF sigue el paradigma **MVC (Modelo-Vista-Controlador)**, separando la presentación de la lógica empresarial y ofreciendo herramientas para gestionar los estados de los componentes entre peticiones HTTP.

8.2 Características de JSF

1. **Componentes Reutilizables**: JSF ofrece una biblioteca de componentes UI que se pueden extender o personalizar.
2. **Soporte de Gestión de Estados**: Cada componente de la vista tiene su estado, que JSF administra a lo largo del ciclo de vida de las solicitudes.
3. **Vinculación con Beans de Backing**: Utiliza **managed beans** (JavaBeans controlados por el framework) para conectar la lógica empresarial y la interfaz de usuario.
4. **Integración con HTML/CSS/JavaScript**: Facilita la creación de páginas usando XHTML que se integran fácilmente con estilos y scripts.
5. **Validación de Datos**: Soporta validaciones en el lado del servidor y del cliente, además de la gestión de errores.
6. **Internacionalización (i18n)**: Permite la localización y soporte para múltiples idiomas.

8.3 Arquitectura

JSF utiliza la arquitectura **Modelo-Vista-Controlador (MVC)**:

1. **Modelo**: El modelo suele estar compuesto por **managed beans** y objetos que contienen la lógica empresarial.
2. **Vista**: Páginas **XHTML** que definen la interfaz gráfica utilizando etiquetas JSF.
3. **Controlador**: JSF actúa como controlador, gestionando peticiones HTTP, manejando el ciclo de vida de los componentes y delegando al modelo y la vista según sea necesario.

8.4 Ciclo de vida

1. **Restore View**: JSF reconstruye la vista de la página si es una solicitud posterior, o crea una nueva si es la primera vez que se accede.
2. **Apply Request Values**: Asigna los valores de los campos del formulario a los componentes correspondientes.

3. **Process Validations**: Valida los datos introducidos según las reglas definidas.
4. **Update Model Values**: Actualiza los valores del modelo (managed beans) con los datos del formulario.
5. **Invoke Application**: Ejecuta la lógica empresarial o acciones asociadas al evento.
6. **Render Response**: Renderiza la vista con los datos actualizados y la envía al cliente.

8.5 Ventajas

1. Ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos de los componentes o manejar elementos UI como objetos con estado en el servidor.
2. Permite construir aplicaciones Web que implementan una separación entre el comportamiento y la presentación tradicionalmente ofrecidas por arquitectura UI del lado del cliente. JSF se hace fácil de usar al aislar al desarrollador del API de Servlet.
3. La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas.
4. La tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JavaServer Faces se han creado directamente sobre el API JavaServlet. Esto permite hacer algunas cosas: usar otra tecnología de presentación junto a JSP, crear componentes propios personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.
5. JavaServer Faces ofrece una gran cantidad de componentes open source para las funcionalidades que se necesiten. Los componentes Tomahawk de MyFaces y ADFFaces de Oracle son un ejemplo. Además, también existe una gran cantidad de herramientas para el desarrollo IDE en JSF al ser el estándar de JAVA.
6. La tecnología JavaServer Faces proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos. Además, ofrece una rápida adaptación para nuevos desarrolladores.

8.6 Configuración central (faces-config.xml)

El archivo **faces-config.xml** es un archivo **de configuración XML** usado en aplicaciones JSF (JavaServer Faces) para definir varios aspectos del comportamiento de la aplicación, como las reglas de **navegación**, los **managed beans**, los **conversores**, **validadores**, y otras configuraciones globales. Si bien algunas configuraciones pueden hacerse mediante anotaciones (por ejemplo, en Java EE 7 y superiores), **faces-config.xml** sigue siendo útil para definir rutas complejas de navegación o para usar en aplicaciones más tradicionales.

8.6.1 Ubicación del archivo

El archivo debe colocarse en la carpeta **WEB-INF** dentro del proyecto web. La ruta completa sería: **src/main/webapp/WEB-INF/faces-config.xml**.

8.6.2 Elementos del JSF

1. **<faces-config>**: Es el **elemento raíz** del archivo. Define el espacio de nombres XML y la versión de JSF utilizada.
    - **Atributo version**: Define la versión del esquema JSF (por ejemplo, 2.3 para la versión más reciente de JSF).
    - **Espacio de nombres XML**: Permite validar el archivo contra el esquema JSF.

```
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_3.xsd"
version="2.3">

    <!-- Resto de las especificaciones –>

</faces-config>
```

2. **<managed-bean>**: Un **managed bean** es un componente de Java utilizado para conectar la interfaz gráfica (página XHTML) con la lógica de negocio. Estos beans pueden declararse directamente en este archivo.
    - **<managed-bean-name>**: Define el nombre con el que el bean se referencia en las páginas JSF.
    - **<managed-bean-class>**: Especifica la clase Java que representa al bean.
    - **<managed-bean-scope>**: Define el **alcance** del bean:
        - **request**: Dura por una solicitud HTTP.
        - **session**: Persiste durante la sesión del usuario.
        - **application**: Comparte una instancia para toda la aplicación.
        - **view**: Dura solo durante la vista actual.

```
<managed-bean>
        <managed-bean-name>myBean</managed-bean-name>
        <managed-bean-class>
                com.example.myBean
        </managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

3. **<navigation-rule>**: Las reglas de navegación definen cómo se redirige entre diferentes vistas en la aplicación. Estas reglas permiten que una página (o **outcome** del bean) determine a qué vista redirigir.
    - **<from-view-id>**: Especifica la página de origen (por ejemplo, login.xhtml).
    - **<navigation-case>**: Define un caso específico de navegación.
        - **<from-outcome>**: Define el **outcome** (resultado) retornado desde el bean. Este resultado determina a qué vista se debe redirigir.
        - **<to-view-id>**: Especifica la página de destino (por ejemplo, home.xhtml).

```
<navigation-rule>

        <from-view-id>/login.xhtml</from-view-id>

        <navigation-case>

                <from-outcome>home</from-outcome>

                <to-view-id>/home.xhtml</to-view-id>

        </navigation-case>

</navigation-rule>
```

4. **<converter>**: Los **conversores** transforman datos entre su representación en la interfaz de usuario y el formato adecuado para el modelo de negocio.
    - **<converter-id>**: Identificador único del conversor.
    - **<converter-class>**: Clase Java que implementa la lógica del conversor.

```
<converter>
        <converter-id>myConverter</converter-id>
        <converter-class>
                com.example.MyConverter
        </converter-class>
</converter>
```

5. **<validator>**: Los **validadores** permiten comprobar la validez de los datos introducidos por el usuario.
    - **<validator-id>**: Identificador único del validador.
    - **<validator-class>**: Clase Java que contiene la lógica de validación.

```
<validator>
        <validator-id>emailValidator</validator-id>
        <validator-class>
                com.example.EmailValidator
        </validator-class>
</validator>
```

6. **<locale-config>**: Permite definir la **configuración regional** predeterminada y las opciones de localización disponibles para la aplicación.
    - **<default-locale>**: Idioma predeterminado (por ejemplo, es para español).
    - **<supported-locale>**: Idiomas adicionales soportados.

```
<locale-config>
        <default-locale>es</default-locale>
        <supported-locale>en</supported-locale>
        <supported-locale>fr</supported-locale>
</locale-config>
```

7. **<resource-bundle>**: Permite asociar un **bundle de recursos** (archivo de propiedades) que contiene textos localizados para la interfaz de usuario.
    - **<base-name>**: Nombre del archivo de propiedades (por ejemplo, messages.properties).
    - **<var>**: Nombre con el que se hace referencia en las páginas XHTML.

8.7 Managed Bean VS CDI Bean

8.7.1 Managed Bean: Un **Managed Bean** (también llamado **JSF Managed Bean**) es una clase Java utilizada en aplicaciones **JSF (JavaServer Faces)** para manejar la lógica de negocio y enlazar con las páginas **XHTML**. Estos beans siguen las reglas del ciclo de vida de JSF y pueden declararse en el archivo **faces-config.xml** o mediante anotaciones. Sus características son:

- **Anotación principal:** @ManagedBean
- **Declaración opcional:** Se pueden declarar en faces-config.xml o mediante anotaciones.
- **Alcance:** Puede tener alcances como request, session, application o view.

- **Uso principal:** Se utiliza principalmente en aplicaciones basadas en JSF para enlazar la vista y la lógica.

8.7.2 CDI Bean: Un **CDI Bean** (**Context and Dependency Injection Bean**) es un componente más flexible y moderno que forma parte de Jakarta EE (antes Java EE). CDI proporciona un mecanismo estándar de inyección de dependencias y es mucho más potente que los Managed Beans tradicionales. Sus características son:

- **Anotación principal:** @Named (para exponer el bean a la vista) y @Inject (para inyectar dependencias).
- **Alcances más flexibles:** Usan anotaciones como @RequestScoped, @SessionScoped o @ApplicationScoped.
- **Inyección de dependencias:** CDI permite inyectar beans fácilmente mediante la anotación @Inject.
- **Uso más generalizado:** Pueden usarse tanto en **JSF** como en otros contextos de la aplicación.

8.7.3 Tabla de Anotaciones de Alcance

Anotación	Descripción	Caso de Uso Principal
@RequestScoped	Vive durante una única petición HTTP.	Formularios simples, autenticación rápida.
@SessionScoped	Vive durante toda la sesión del usuario.	Login, carrito de compras.
@ApplicationScoped	Vive durante toda la aplicación.	Configuración global, parámetros compartidos.
@ViewScoped	Vive mientras el usuario permanezca en la misma vista.	Formularios multi-paso, peticiones AJAX.
@Dependent	Depende del ciclo de vida del bean que lo inyecta.	Servicios auxiliares sin estado.
@ConversationScoped	Controla su ciclo de vida con un flujo explícito.	Flujos complejos como compras o reservas.

8.7.4 Tabla de comparación Managed Bean y CDI Bean

Característica	Managed Bean (JSF)	CDI Bean
Anotación principal	**@ManagedBean**	**@Named** (para exponer el bean a la vista)

Alcances	**@RequestScoped, @SessionScoped, @ViewScoped**	Más flexible: **@RequestScoped, @SessionScoped, @Dependent,** etc.
Inyección de dependencias	No soportan **@Inject** nativamente	Soportan inyección de dependencias mediante **@Inject**
Configuración	Opcionalmente en **faces-config.xml**	No necesita configuración en XML
Uso principal	En aplicaciones basadas en JSF	En JSF, REST, EJB, y otros contextos
Disponibilidad	Solo en JSF	Usado en todo Jakarta EE

8.8 Práctica

8.8.1 Objetivo

Crear un formulario sencillo de inicio de sesión utilizando la tecnología JSF. Usaremos un managed bean para almacenar los datos del usuario.

8.8.2 Requisitos Previos

1. NetBeans IDE instalado
2. Apache Tomcat configurado como servidor en NetBeans
3. JDK instalado

8.8.3 Paso a paso:

- **Paso 1: Configuración inicial**
    1. **Abre NetBeans** y selecciona:
       File → New Project
    2. En el asistente de creación, selecciona:
        - **Categories** Java with Maven (Java with Ant es otra alternativa)
        - **Projects**: **Web Application**.
        - Haz click en **Next**.
    3. **Nombre del Proyecto**:
       Ingresa un nombre como “**LoginJSF**”.
        - Ubicación: Elige una carpeta donde guardar el proyecto.
        - Haz click en **Next**.
    4. **Servidor**:
        - Selecciona **Apache Tomcat** (debe estar instalado y configurado).
        - Versión: **Java EE 8 o superior**.
    5. **Configuración del Framework**:
        - Selecciona **JavaServer Faces** como framework.

- Añade la dependencia al pom.xml:

```
<dependencies>
    <dependency>
        <groupId>org.glassfish</groupId>
        <artifactId>jakarta.faces</artifactId>
        <version>4.0.0</version>
    </dependency>
</dependencies>
```

6. **Dependencias JSF**: NetBeans añade automáticamente las bibliotecas necesarias. En caso contrario es necesario incluir **Mojarra** (la implementación de referencia de JSF) en las dependencias del proyecto.

- **Paso 2: Crear un Managed Bean**

  Crea una clase normal de Java, click derecho encima de src (source packages) y selecciona **New → Java Class**. Añádela a un paquete nombrado apropiadamente (por ejemplo, com.example.beans). Para que nuestra clase se consideré un **Managed Bean** tenemos que añadir la anotación **@ManagedBean** y la anotación de alcance.

```
package com.example.beans;


import jakarta.faces.bean.ManagedBean;
import jakarta.faces.bean.SessionScoped;
import java.io.Serializable;


@ManagedBean(name = "loginBean")
@SessionScoped
public class LoginBean implements Serializable {
   private String username;
```

```
    private String password;

    private String message;


    // Getters y setters

    public String getUsername() { return username; }

    public void setUsername(String username) { this.username =
username; }


    public String getPassword() { return password; }

    public void setPassword(String password) { this.password =
password; }


    public String getMessage() { return message; }


    // Lógica de autenticación

    public String login() {

        if ("admin".equals(username) && "1234".equals(password)) {

            message = "Login exitoso.";

            return "home";  // Navegar a home.xhtml

        } else {

            message = "Usuario o contraseña incorrectos.";

            return "login";  // Permanecer en login.xhtml

        }

    }

}
```

Aviso: *Asegúrate de hacer los "import" adecuadamente o dará error de compilación.*

- **Paso 3: Crear las páginas XHTML**

  Crea dos páginas XHTML (Haz click derecho en la carpeta **Web Pages** y selecciona **New** → **XHTML**), les asignaremos como nombre "**login.xhtml**" y "**home.xhtml**" . Sustituye el código de ambas páginas por los siguientes:

  login.xhtml:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<h:head>
   <title>Login Page</title>
</h:head>
<h:body>
   <h:form>
      <h3>Inicio de Sesión</h3>
      <h:outputLabel value="Usuario: " for="username"/>
      <h:inputText id="username"
value="#{loginBean.username}" required="true"/><br/>

      <h:outputLabel value="Contraseña: "
for="password"/>
      <h:inputSecret id="password"
value="#{loginBean.password}" required="true"/><br/>

      <h:commandButton value="Login"
action="#{loginBean.login}"/>
```

```
        <h:outputText value="#{loginBean.message}"
style="color:red"/>

    </h:form>

</h:body>

</html>
```

home.xhtml:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"

      xmlns:h="http://java.sun.com/jsf/html">

<h:head>

    <title>Home</title>

</h:head>

<h:body>

    <h2>Bienvenido, #{loginBean.username}!</h2>

    <h:commandLink value="Cerrar Sesión" action="login"/>

</h:body>

</html>
```

- **Paso 4: Configurar la Navegación**

  Crear el archivo **faces-config.xml** (click derecho encima de la carpeta **WEB-INF** y seleccionamos **New→ JSF Faces Configuration**). Añade el siguiente código:

```
<?xml version="1.0" encoding="UTF-8"?>

<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"


xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
                xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_3.xsd"

                version="2.3">

    <navigation-rule>

        <from-view-id>/login.xhtml</from-view-id>

        <navigation-case>

            <from-outcome>home</from-outcome>

            <to-view-id>/home.xhtml</to-view-id>

        </navigation-case>

    </navigation-rule>

</faces-config>
```

- **Paso 5: Ejecutar la Aplicación**
    1. Haz clic derecho en el proyecto y selecciona "Run".
    2. Accede a http://localhost:8080/LoginJSF/faces/login.xhtml

9 JPA

9.1 Introducción a JPA

Java Persistence API (JPA) es la especificación oficial de **persistencia de datos** en aplicaciones Java EE/Jakarta EE y SE. Su propósito es **mapear objetos Java (POJOs) a tablas en bases de datos relacionales**, simplificando la interacción entre la lógica del negocio y los datos persistentes.

JPA abstrae los detalles del acceso a la base de datos mediante **ORM (Object-Relational Mapping)**, permitiendo al desarrollador enfocarse más en la lógica del negocio sin preocuparse por sentencias SQL complicadas.

9.2 Características

1. **ORM (Mapeo Objeto-Relacional):**
    - Los objetos Java se mapean a tablas en la base de datos.
    - Las instancias de clase se convierten en registros de la base de datos y viceversa.

2. **Independencia del Proveedor:**
    - JPA es solo una especificación, lo que permite utilizar diferentes **implementaciones** (como Hibernate, EclipseLink, OpenJPA).
3. **Consultas con JPQL (Java Persistence Query Language):**
    - JPQL es un lenguaje de consultas orientado a objetos similar a SQL, pero opera sobre entidades en lugar de tablas.
4. **Administración de Transacciones:**
    - Las transacciones se pueden gestionar automáticamente o de forma manual para asegurar la consistencia de los datos.

9.3 Elementos fundamentales

1. Entidades

Una **entidad** en JPA es una clase Java que representa una tabla en la base de datos. Cada instancia de esta clase corresponde a una **fila** en la tabla.

Ejemplo:

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.Table;


@Entity

@Table(name = "usuarios")

public class Usuario {


    @Id

    private Long id;

    private String nombre;

    private String email;


    // Getters y Setters

    public Long getId() {
```

```
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

- La anotación **@Entity** marca la clase como una entidad gestionada por JPA.
- **@Table(name = "usuarios")** indica la tabla de la base de datos que esta entidad representa.
- **@Id** especifica la columna que será la **clave primaria**.

2. Persistencia (EntityManager)

EntityManager es el principal componente de JPA que se encarga de realizar las operaciones de persistencia, como **guardar, actualizar, eliminar y consultar entidades**. Cada **EntityManager** gestiona un conjunto de entidades.

Ejemplo:

```
import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;


public class UsuarioDAO {


   private EntityManagerFactory emf =
Persistence.createEntityManagerFactory("miUnidadDePersistencia");


   public void guardarUsuario(Usuario usuario) {

      EntityManager em = emf.createEntityManager();

      em.getTransaction().begin();  // Inicia la transacción

      em.persist(usuario);         // Persiste la entidad en la base de datos

      em.getTransaction().commit(); // Finaliza la transacción

      em.close();

   }

}
```

3. Unidad de Persistencia (persistence.xml)

El archivo **persistence.xml** se utiliza para configurar los detalles de la conexión a la base de datos y los parámetros de JPA. Se encuentra en la carpeta **META-INF** del proyecto.

Ejemplo:

```
<persistence xmlns="http://jakarta.ee/xml/ns/persistence"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
            xsi:schemaLocation="http://jakarta.ee/xml/ns/persistence

http://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"

            version="3.0">


   <persistence-unit name="miUnidadDePersistencia"
transaction-type="RESOURCE_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

      <class>com.example.Usuario</class>

      <properties>

         <property name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/miDB" />

         <property name="jakarta.persistence.jdbc.user" value="root" />

         <property name="jakarta.persistence.jdbc.password"
value="1234" />

         <property name="jakarta.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />

         <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect" />

         <property name="hibernate.hbm2ddl.auto" value="update" />

      </properties>

   </persistence-unit>

</persistence>
```

- **<persistence-unit>** define una unidad de persistencia.
- **provider** especifica la implementación de JPA (en este caso, Hibernate).
- **properties** contiene los detalles de conexión a la base de datos.

4. JPQL (Java Persistence Query Language)

JPQL permite realizar consultas sobre entidades, similar a SQL pero orientado a objetos.

Ejemplo:

```
public List<Usuario> obtenerUsuarios() {
   EntityManager em = emf.createEntityManager();
   List<Usuario> usuarios = em.createQuery("SELECT u FROM Usuario u", Usuario.class).getResultList();
   em.close();
   return usuarios;
}
```

9.4 Tabla de Anotaciones JPA

Anotación	Ámbito de Uso	Descripción
@Entity	Type	Marca una clase como entidad JPA que será mapeada a una tabla en la base de datos.
@Table	Type	Especifica la tabla asociada a la entidad. Sin ella, se usa el nombre de la clase.
@Id	Field / Method	Define el campo o método como la clave primaria de la entidad.
@GeneratedValue	Field / Method	Configura la generación automática de valores para la clave primaria.
@Column	Field / Method	Especifica propiedades de la columna asociada (nombre, longitud, tipo).
@Basic	Field / Method	Define un mapeo básico sin modificaciones adicionales.
@Lob	Field / Method	Indica que el campo es un Large Object (BLOB/CLOB).
@Transient	Field / Method	Excluye el campo del mapeo a la base de datos.
@Enumerated	Field / Method	Define cómo se almacena un Enum en la base de datos (ORDINAL o STRING).
@Temporal	Field / Method	Define si un Date se almacena como DATE, TIME o TIMESTAMP.
@Embedded	Field / Method	Embebe un objeto dentro de la entidad, tratándolo como parte de esta.
@Embeddable	Type	Marca una clase como embebible, utilizada dentro de otra entidad.
@OneToOne	Field / Method	Define una relación uno a uno entre dos entidades.

@OneToMany	Field / Method	Define una relación uno a muchos.
@ManyToOne	Field / Method	Define una relación muchos a uno.
@ManyToMany	Field / Method	Define una relación muchos a muchos.
@JoinColumn	Field / Method / Parameter	Especifica la columna de unión (clave foránea).
@JoinTable	Field / Method	Define la tabla intermedia para relaciones muchos a muchos.
@MapsId	Field / Method	Indica que un campo comparte el mismo valor que la clave primaria en otra entidad.
@EmbeddedId	Field / Method	Define que la entidad tiene una clave primaria compuesta (objeto embebido).
@IdClass	Type	Permite usar una clase separada como clave primaria compuesta.
@Version	Field / Method	Marca un campo como número de versión para control de concurrencia.
@NamedQuery	Type	Define una consulta JPQL predefinida en la entidad.
@NamedQueries	Type	Agrupa varias anotaciones @NamedQuery.
@Query (de Spring)	Method / Parameter	Define una consulta personalizada en un DAO/Repository.
@Access	Type / Field / Method	Especifica si JPA accede a los atributos o a los métodos.
@Cacheable	Type	Marca una entidad como cacheable.
@SequenceGenerator	Type / Field / Method	Configura un generador de secuencia para la clave primaria.
@TableGenerator	Type / Field / Method	Define un generador basado en tablas para claves primarias.
@SqlResultSetMapping	Type	Mapea resultados de una consulta SQL nativa a una clase.
@NamedNativeQuery	Type	Define una consulta nativa SQL predefinida.
@PrePersist	Method	Callback que se ejecuta antes de insertar la entidad en la base de datos.
@PostPersist	Method	Callback que se ejecuta después de insertar la entidad.
@PreUpdate	Method	Callback que se ejecuta antes de actualizar la entidad.
@PostUpdate	Method	Callback que se ejecuta después de actualizar la entidad.
@PreRemove	Method	Callback que se ejecuta antes de eliminar la entidad.
@PostRemove	Method	Callback que se ejecuta después de eliminar la entidad.
@PostLoad	Method	Callback que se ejecuta después de cargar la entidad desde la base de datos.

9.5 Tabla de Anotaciones de Validación (javax.validation / Jakarta Validation)

Anotación	Ámbito de Uso	Descripción
@NotNull	Field / Method / Parameter	Valida que el valor no sea nulo.
@Null	Field / Method / Parameter	Valida que el valor sea nulo.
@NotEmpty	Field / Method / Parameter	Valida que una cadena o colección no esté vacía (no puede ser null ni contener longitud 0).
@NotBlank	Field / Method / Parameter	Valida que una cadena no sea nula ni contiene solo espacios en blanco.
@Size	Field / Method / Parameter	Valida que el tamaño de una colección, mapa, array o cadena esté dentro de un rango definido.
@Min	Field / Method / Parameter	Valida que un valor numérico sea mayor o igual a un mínimo definido.
@Max	Field / Method / Parameter	Valida que un valor numérico sea menor o igual a un máximo definido.
@Positive	Field / Method / Parameter	Valida que un valor numérico sea positivo (> 0).
@PositiveOrZero	Field / Method / Parameter	Valida que un valor numérico sea positivo o 0.
@Negative	Field / Method / Parameter	Valida que un valor numérico sea negativo (< 0).
@NegativeOrZero	Field / Method / Parameter	Valida que un valor numérico sea negativo o 0.
@DecimalMin	Field / Method / Parameter	Valida que un valor numérico decimal sea mayor o igual al mínimo especificado.
@DecimalMax	Field / Method / Parameter	Valida que un valor numérico decimal sea menor o igual al máximo especificado.
@Digits	Field / Method / Parameter	Valida que un número tenga una cantidad específica de dígitos enteros y decimales.
@Past	Field / Method / Parameter	Valida que una fecha sea anterior al momento actual.
@PastOrPresent	Field / Method / Parameter	Valida que una fecha sea anterior o igual al momento actual.
@Future	Field / Method / Parameter	Valida que una fecha sea posterior al momento actual.
@FutureOrPresent	Field / Method / Parameter	Valida que una fecha sea posterior o igual al momento actual.

@Email	Field / Method / Parameter	Valida que una cadena tenga un formato de correo electrónico válido.
@Pattern	Field / Method / Parameter	Valida que una cadena coincida con una expresión regular definida.
@AssertTrue	Field / Method / Parameter	Valida que el valor de un booleano sea true.
@AssertFalse	Field / Method / Parameter	Valida que el valor de un booleano sea false.
@CNPJ (Bean Validation BR)	Field / Method / Parameter	Valida un CNPJ (número de identificación fiscal brasileño).
@CPF (Bean Validation BR)	Field / Method / Parameter	Valida un CPF (identificación personal brasileña).
@CreditCardNumber	Field / Method / Parameter	Valida que una cadena represente un número de tarjeta de crédito válido.
@Length (Hibernate Validator)	Field / Method / Parameter	Valida que la longitud de una cadena esté dentro de un rango específico.
@Range (Hibernate Validator)	Field / Method / Parameter	Valida que un valor numérico esté dentro de un rango especificado.
@URL	Field / Method / Parameter	Valida que una cadena sea una URL válida.
@Valid	Field / Method / Parameter	Valida que el objeto referenciado también sea validado según sus propias restricciones.
@ConvertGroup	Field / Method / Parameter	Convierte un grupo de validación en otro durante la validación.
@Constraint	Type / Field / Method / Parameter	Define una restricción personalizada.
@ReportAsSingleViolation	Type	Agrupa varias restricciones pero reporta solo una violación al usuario.

10 EJB

10.1 Introducción a EJB

Enterprise JavaBeans (EJB) es una especificación del **Java EE/Jakarta EE** que define un modelo para desarrollar componentes de software modulares, transaccionales y distribuidos que se ejecutan en un entorno empresarial. Estos componentes están diseñados para manejar tareas complejas, como la **gestión de transacciones, seguridad, concurrencia y servicios remotos**, permitiendo que los desarrolladores se enfoquen en la lógica de negocio sin preocuparse demasiado por los aspectos de infraestructura.

La principal diferencia con los **JavaBeans** convencionales es que si pueden utilizarse en entornos de objetos distribuidos al soportar nativamente la invocación remota RMI.

10.2 Casos de uso

EJB es especialmente útil en aplicaciones que requieren procesamiento en múltiples capas, como:

- **Sistemas bancarios**: con transacciones complejas y concurrentes.
- **Aplicaciones de e-commerce**: que manejan grandes volúmenes de datos y usuarios.
- **Servicios empresariales distribuidos**: que se comunican entre múltiples sistemas.

EJB ofrece servicios integrados como:

- **Transacciones distribuidas** (usando JTA).
- **Seguridad declarativa** y programática.
- **Soporte para concurrencia y multithreading**.
- **Mensajería asíncrona** mediante JMS.
- **Escalabilidad** mediante balanceo de carga y agrupación de instancias.

10.3 Tipos de EJB

1. Session Beans

Los **Session Beans** encapsulan la lógica de negocio y están asociados a un cliente específico. Se dividen en:

- **Stateless Session Beans** (SLSB):
  No mantiene estado entre invocaciones. Son ideales para operaciones independientes, como cálculos o servicios rápidos.
  - **Ejemplo**: Validación de credenciales de usuario.
- **Stateful Session Beans** (SFSB):
  Mantienen el estado entre las llamadas del cliente durante toda la sesión. Son útiles para procesos de larga duración, como un carrito de compras.
  - **Ejemplo**: Gestión de una compra en e-commerce.
- **Singleton Session Beans**:
  Hay solo una instancia en toda la aplicación. Se utiliza para datos globales o servicios que deben ser únicos (como caché o configuración).
  - **Ejemplo**: Servicio de caché compartido.

Stateless	Stateful
Diferentes servidores entregan diferente información al mismo tiempo	Mismo Servidor procesa todas las peticiones
Ni se mantiene ni se almacenan peticiones previas	Se almacena información de previas peticiones
Las peticiones son independientes y no dependen de resultados anteriores	Peticiones basadas en resultados de otras peticiones
Peticiones al servidor independientes en cada petición, no hace falta ningún estado	Basada en protocolos de internet que requieren de estado

2. Message-Driven Beans (MDB)

Los **Message-Driven Beans** se utilizan para **procesar mensajes asíncronos**. Escuchan una cola de mensajes **JMS (Java Message Service)** y ejecutan lógica en respuesta a esos mensajes.

- **Uso típico**: Procesamiento de pedidos en segundo plano.

10.4 Ciclo de vida de los Session Beans

1. Stateless Session Bean - Lifecycle

Un **Stateless Session Bean** (SLSB) **no mantiene estado entre llamadas** de cliente. Su ciclo de vida es más sencillo que el de los Stateful Beans.

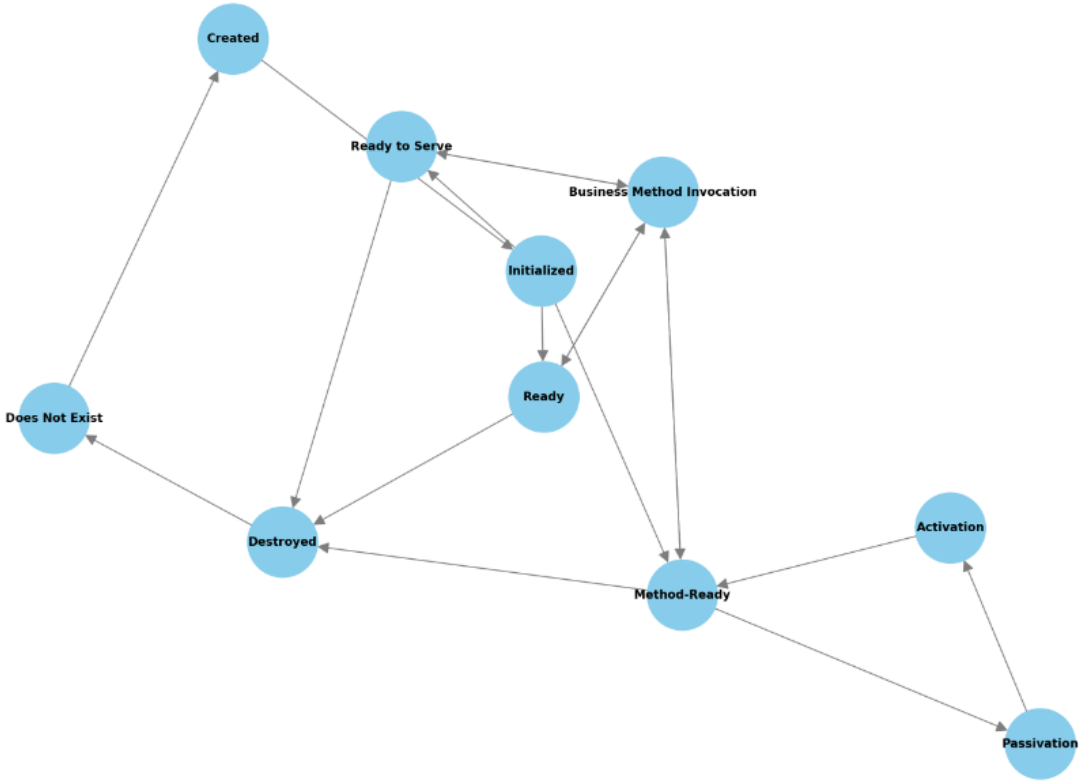
Fases del ciclo de vida:

Does Not Exist → Created → Initialized → Ready to Serve ↔ Business Method Invocation → Destroyed → Does Not Exist

1. **Does Not Exist**:
    - El bean aún no ha sido creado.
2. **Created (Instantiation)**:

- El contenedor crea una instancia del bean mediante el constructor por defecto (sin parámetros).

3. **Initialized (Post-Construct State)**:
   - El contenedor invoca el método **@PostConstruct**, permitiendo realizar tareas de inicialización.
4. **Ready to Serve (Business Method Invocation)**:
   - El bean está listo para procesar las solicitudes de los clientes.
   - Cada vez que un cliente invoca un método, el contenedor asigna una instancia del bean desde un **pool**.
5. **Destroyed (Pre-Destroy State)**:
   - Antes de destruir la instancia, el contenedor llama al método **@PreDestroy** para liberar recursos.
6. **Does Not Exist**:
   - La instancia es eliminada y devuelta al pool o destruida si ya no es necesaria.



2. Stateful Session Bean - Lifecycle

Un **Stateful Session Bean (SFSB) mantiene el estado** específico del cliente entre las invocaciones. El ciclo de vida es más complejo, ya que las instancias están ligadas a la sesión del cliente.

Fases del ciclo de vida:

Does Not Exist → Created → Initialized → Method-Ready ↔ Passivation ↔ Activation → Destroyed → Does Not Exist

1. **Does Not Exist**:
   - No existe ninguna instancia del bean.
2. **Created (Instantiation)**:
   - El contenedor crea una nueva instancia para cada cliente que necesita un SFSB.
3. **Initialized (Post-Construct State)**:
   - Después de la creación, se invoca **@PostConstruct** para inicializar recursos.
4. **Method-Ready (Business Method Invocation)**:
   - El bean está listo para que el cliente invoque métodos de negocio. El estado se mantiene entre invocaciones.
5. **Passivation (Passivated State)** (Opcional):
   - Si el bean no se usa por un tiempo, el contenedor puede dejarlo pasivo (serializar su estado) para liberar recursos. El contenedor invoca **@PrePassivate** antes de dejarlo pasivo.
6. **Activation (Activated State)**:
   - Cuando el cliente vuelve a utilizar el bean, el contenedor lo **activa** (lo deserializa) y ejecuta **@PostActivate**.
7. **Destroyed (Pre-Destroy State)**:
   - Si la sesión termina o expira, el contenedor destruye la instancia invocando **@PreDestroy**.

3. Singleton Session Bean - Lifecycle

Un **Singleton Session Bean** garantiza que **solo haya una instancia** del bean en toda la aplicación, compartida por todos los clientes.

Fases del ciclo de vida:

Does Not Exist → Created → Initialized → Ready ↔ Business Method Invocation → Destroyed → Does Not Exist

1. **Does Not Exist**:
   - No hay ninguna instancia del bean.
2. **Created (Instantiation)**:
   - El contenedor crea una única instancia al inicio de la aplicación (a menos que el bean esté configurado con **@Startup**).
3. **Initialized (Post-Construct State)**:
   - Después de la creación, se ejecuta **@PostConstruct** para realizar cualquier inicialización.
4. **Ready (Business Method Invocation)**:

- El bean está disponible para que cualquier cliente invoque sus métodos.

5. **Destroyed (Pre-Destroy State)**:
   - Al finalizar la aplicación, el contenedor llama a **@PreDestroy** para liberar recursos y luego destruye el bean.

4. Tabla anotaciones utilizadas en el ciclo de vida

Anotación	Descripción	Ámbito	Aplicable a	Ejemplo de Uso
@PostConstruct	Método invocado después de la creación del bean para inicialización de recursos.	Method	Stateless, Stateful, Singleton	Inicializar conexiones a BD o servicios externos.
@PreDestroy	Método invocado antes de la destrucción del bean para liberar recursos.	Method	Stateless, Stateful, Singleton	Cerrar conexiones o liberar memoria.
@PrePassivate	Método invocado antes de la pasivación del Stateful bean.	Method	Stateful	Guardar temporalmente el estado del bean.
@PostActivate	Método invocado después de la activación del Stateful bean.	Method	Stateful	Restaurar recursos al deserializar el bean.
@Startup	Fuerza que el Singleton bean se inicie al arrancar la aplicación.	Type	Singleton	Servicios que deben estar disponibles al inicio (e.g., cachés).
@Lock	Controla el nivel de concurrencia en los Singleton beans.	Method / Type	Singleton	Asegura exclusión mutua o acceso concurrente.
@AccessTimeout	Define un límite de tiempo para adquirir un bloqueo en un Singleton bean.	Method	Singleton	Previene bloqueos indefinidos en situaciones de alta concurrencia.

@Remove	Marca un método que finaliza la instancia de un Stateful bean.	Method	Stateful	Indica al contenedor que el bean ya no es necesario.
@StatefulTimeout	Especifica un límite de tiempo para que un Stateful bean permanezca inactivo antes de ser destruido.	Type	Stateful	Útil para liberar recursos automáticamente si el cliente deja de usar el bean.
@Singleton	Define un Singleton Session Bean.	Type	Singleton	Asegura que solo haya una instancia del bean.
@Stateless	Define un Stateless Session Bean.	Type	Stateless	Útil para operaciones que no requieren estado entre llamadas.
@Stateful	Define un Stateful Session Bean.	Type	Stateful	Ideal para procesos con estado como sesiones de usuario.

11 JMS

11.1 Introducción a JMS

Java Message Service (JMS) es una API de Java que permite crear, enviar, recibir y leer mensajes entre aplicaciones de software de forma asíncrona y desacoplada. Es un componente clave de la plataforma Java EE (Enterprise Edition) y está diseñado para soportar la comunicación en sistemas distribuidos, facilitando el intercambio de mensajes entre diferentes aplicaciones o módulos.

Los sistemas y aplicaciones JMS son necesarios en entornos empresariales que requieren el intercambio de información entre entidades heterogéneas. Debería funcionar independientemente de otros y debería ser fácilmente extensible. JMS permite que las aplicaciones se comuniquen a través de mensajes sin requerir una conexión directa, aumentando así la flexibilidad y escalabilidad del sistema.

11.2 Características

1. **Comunicación Asincrónica**: Las aplicaciones que usan JMS no necesitan esperar una respuesta inmediata después de enviar un mensaje. Los mensajes se entregan cuando el destinatario está disponible para recibirlos.
2. **Desacoplamiento**: El remitente de un mensaje no necesita saber nada sobre el destinatario, y viceversa. Las aplicaciones pueden funcionar de manera independiente, lo que permite una alta modularidad y flexibilidad.

3. **Fiabilidad**: JMS asegura que los mensajes se entreguen de manera fiable a través de mecanismos como la persistencia de mensajes, confirmación de recepción y gestión de errores.
4. **Compatibilidad con Transacciones**: JMS puede trabajar dentro de un contexto transaccional, lo que garantiza que los mensajes se entreguen como parte de una transacción más amplia, o que se deshagan en caso de fallo.
5. **Escalabilidad**: Los sistemas que usan JMS son altamente escalables, ya que permiten la comunicación entre muchas aplicaciones y el procesamiento simultáneo de mensajes en un entorno distribuido.

11.3 Modelos de mensajeria

1. **Point-to-Point (P2P)** o **Cola de Mensajes**:
    - Este modelo sigue el patrón **productor-consumidor**, donde un mensaje es enviado a una cola y un solo receptor lo recibe.
    - El productor envía mensajes a una cola específica, y sólo un consumidor puede recibir el mensaje.
    - Cada mensaje es consumido por un solo receptor, lo que asegura que no haya duplicación de procesamiento.
    - Ejemplo típico: aplicaciones de procesamiento por lotes o sistemas de colas de trabajo.

    Elementos Clave en Point-to-Point:

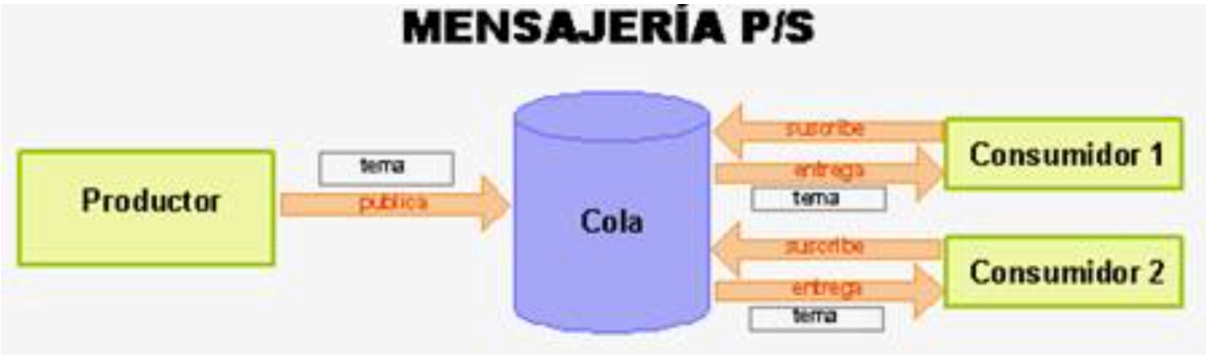
    - **Cola (Queue)**: Contiene los mensajes hasta que son consumidos.
    - **Productor (Sender)**: Envía mensajes a la cola.
    - **Consumidor (Receiver)**: Recibe los mensajes de la cola.

2. **Public/Subscribe (Pub/Sub)** o **Tópicos de Mensajes**:
    - En este modelo, un mensaje enviado a un tópico puede ser recibido por múltiples suscriptores.
    - Los productores publican mensajes en un tópico, y todos los suscriptores que están escuchando ese tópico recibirán el mensaje.
    - Este modelo permite la distribución de un mensaje a varios consumidores.

- Ejemplo típico: sistemas de notificación, actualizaciones en tiempo real (como las cotizaciones en bolsa).

Elementos Clave en Pub/Sub:

- **Tópico (Topic)**: Canal donde los productores publican los mensajes y los suscriptores escuchan.
- **Publicador (Publisher)**: Envía mensajes a un tópico.
- **Suscriptor (Subscriber)**: Escucha mensajes de un tópico.



11.4 Tipo de mensajes

1. **TextMessage**: Mensajes de texto simples. Este es el tipo más utilizado, por ejemplo, para enviar mensajes en formato XML o JSON.
2. **ObjectMessage**: Contiene un objeto Java serializado. Permite enviar objetos completos entre aplicaciones.
3. **BytesMessage**: Un mensaje de datos en formato binario, que es útil para transmitir archivos u otro tipo de datos binarios.
4. **StreamMessage**: Contiene una secuencia de datos primitivos, similar a un flujo de datos (stream).
5. **MapMessage**: Un conjunto de pares clave-valor, donde las claves son String y los valores son tipos de datos primitivos.
6. **Message**: Es el tipo más genérico y sólo contiene un encabezado y propiedades opcionales.

11.5 Arquitectura

JMS sigue una arquitectura basada en un **middleware de mensajería** o **Message-Oriented Middleware (MOM)**, lo que significa que se basa en intermediarios (brokers) que almacenan y distribuyen mensajes. La arquitectura consta de varios componentes:

1. **JMS Provider**: El proveedor de JMS es el sistema que implementa la API de JMS y proporciona la infraestructura necesaria para que los mensajes se

envíen y reciban. Ejemplos de proveedores son Apache ActiveMQ, IBM MQ y RabbitMQ.

2. **JMS Client**: Son las aplicaciones que producen y consumen mensajes. Los clientes de JMS interactúan con el proveedor para enviar y recibir mensajes.
3. **Administered Objects**: Son configuraciones predefinidas que un administrador del sistema crea para simplificar el uso de JMS. Los dos objetos administrados principales son:
    - **ConnectionFactory**: Un objeto que permite a los clientes JMS crear conexiones con el proveedor de mensajería.
    - **Destination**: Define el lugar donde se envían y reciben los mensajes. Puede ser una cola o un tópico.
4. **Messages**: Son los datos que se intercambian entre los clientes JMS.
5. **Connection**: Es la conexión que se establece entre un cliente JMS y el proveedor.
6. **Session**: Representa una sesión entre el cliente y el proveedor, en la que se pueden enviar y recibir mensajes. La sesión también puede ser transaccional, lo que garantiza la integridad en el procesamiento de los mensajes.

11.6 Transacciones

JMS soporta transacciones, lo que significa que un conjunto de mensajes puede ser enviado o recibido como una única operación atómica. En una transacción, JMS asegura que:

1. Los mensajes se entregan en el orden correcto.
2. Si ocurre un error durante la transacción, todos los mensajes pueden ser revertidos (rollback).
3. Los mensajes se confirman cuando el consumidor ha procesado correctamente el mensaje.

11.7 Integración

JMS se integra con otros componentes de Java EE como **Enterprise JavaBeans (EJB)**. Por ejemplo, un **Message-Driven Bean (MDB)** es un tipo especial de EJB que se utiliza para recibir mensajes JMS de manera asíncrona. Los MDB permiten que las aplicaciones Java EE procesen mensajes de manera eficiente sin tener que interactuar directamente con el proveedor de JMS.

11.8 Ejemplo

1. **Productor:**
   ```
   ConnectionFactory connectionFactory = new
   ActiveMQConnectionFactory("tcp://localhost:61616");
   Connection connection = connectionFactory.createConnection();
   ```

```
Session session = connection
                .createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("testQueue");
MessageProducer producer = session.createProducer(queue);

TextMessage message = session.createTextMessage("Hello, JMS!");
producer.send(message);

connection.close();
```

2. Consumidor:

```
ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory("tcp://localhost:61616");
Connection connection = connectionFactory.createConnection();
Session session = connection
                .createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("testQueue");
MessageConsumer consumer = session.createConsumer(queue);

connection.start();

Message message = consumer.receive();
if (message instanceof TextMessage) {
   TextMessage textMessage = (TextMessage) message;
   System.out.println("Received message: " + textMessage.getText());
}

connection.close();
```

12 JNDI

12.1 Introducción a JNDI

JNDI (Java Naming and Directory Interface) permite que las aplicaciones distribuidas busquen servicios de una manera abstracta e independiente de recursos.
El caso de uso más común es configurar un grupo de conexión de base de datos en un servidor de aplicaciones Java EE. Cualquier aplicación que se implemente en ese servidor puede obtener acceso a las conexiones que necesitan utilizando el nombre JNDI Java:comp/env/FooBarPool sin tener que conocer los detalles de la conexión.

12.2 Conceptos Clave de JNDI

1. **Servicios de nombres**: JNDI permite acceder a recursos mediante un **nombre lógico** o alias. Los servicios de nombres asignan estos nombres a objetos o servicios (por ejemplo, una base de datos o una cola de mensajes). Un ejemplo de un servicio de nombres es **DNS** (Sistema de Nombres de Dominio), que convierte nombres de dominio legibles por humanos (por ejemplo, www.ejemplo.com) en direcciones IP.
2. **Servicios de directorio**: Además de asignar nombres, los servicios de directorio pueden almacenar información adicional sobre los recursos. Esta funcionalidad es proporcionada por sistemas como **LDAP** (Lightweight Directory Access Protocol), que almacena jerárquicamente información sobre personas, organizaciones, y otros objetos.
3. **Proveedor de servicio**: JNDI es un marco **agnóstico** del proveedor, lo que significa que puede interactuar con diferentes servicios de nombres y directorio (como LDAP, DNS, NIS, RMI) a través de proveedores que implementen la interfaz JNDI.
4. **Contexto**: El **contexto** en JNDI representa un espacio de nombres en el que los objetos pueden ser vinculados o buscados. Dentro de un espacio de nombres, puedes acceder a diferentes objetos usando su nombre.

12.3 Estructura de JNDI

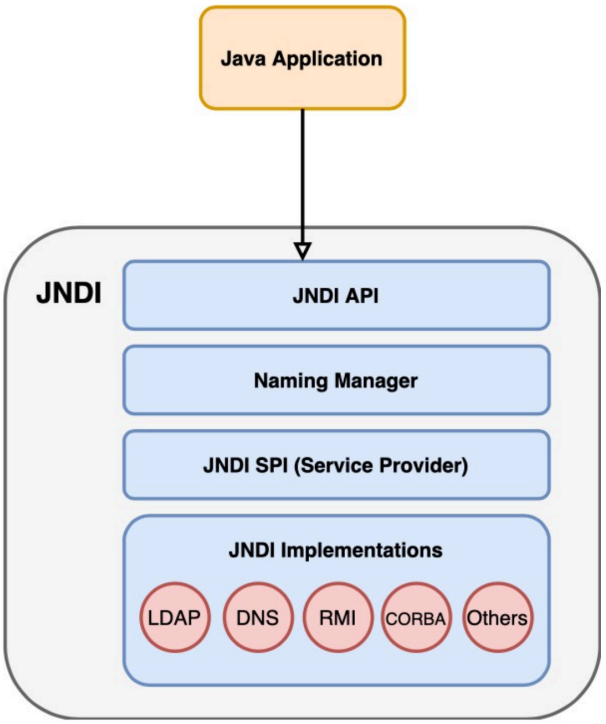
- **InitialContext**: Es el punto de entrada principal para interactuar con JNDI. Proporciona acceso al servicio de nombres y te permite realizar búsquedas y operaciones.
- **Contexto de nombres**: Es un espacio de nombres jerárquico similar a un sistema de archivos donde los recursos se asignan a nombres lógicos.
- **Directorio**: JNDI puede gestionar no solo la asociación de nombres a objetos, sino también la organización jerárquica de estos recursos y la adición de metadatos.

12.4 Arquitectura

La aplicación puede usar la API JNDI para realizar operaciones en el servicio de nombres, luego lo que sucede dentro de ella depende de qué proveedor de servicios (**SPI***) esté conectado. Para declarar uno de los SPI que desea usar en su aplicación, debe especificar una clase que sea parte del SPI específico.

** Bus SPI (Serial Peripheral Interface) es un estándar de comunicaciones.*

JDK incluye cuatro proveedores de servicios: ***LDAP*** **,** ***DNS*** **,** ***RMI*** **y** ***CORBA*** . Sin embargo, puede crear su propio proveedor de servicios para sus servicios personalizados.



12.5 Ejemplo

1. Requisitos: Para usar el JNDI, debe tener las clases JNDI y uno o más proveedores de servicios. El Java 2 SDK, v1.3 incluye tres proveedores de servicios para los siguientes servicios de nombres/directorio:

- Protocolo ligero de acceso a directorios (LDAP)
- Common Object Request Broker Architecture (CORBA) Servicio de nombres de Common Object Services (COS)
- Registro de invocación de método remoto (RMI) de Java

2. Configuración del DataSource en el servidor de aplicaciones: El servidor de aplicaciones (como Tomcat, WildFly o GlassFish) se configura para proporcionar un **DataSource** que representa una conexión a una base de datos. Se le asigna un nombre JNDI, por ejemplo: jdbc/MyDB.

3. Código:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

public class JndiExample {
   public static void main(String[] args) {
      try {
```

```
            // Crear el contexto inicial JNDI
            Context initialContext = new InitialContext();

            // Buscar el DataSource registrado en el servidor
            DataSource ds = (DataSource)
initialContext.lookup("java:/comp/env/jdbc/MyDB");

            // Obtener una conexión de la fuente de datos
            Connection conn = ds.getConnection();

            // Trabajar con la conexión...
            System.out.println("Conexión establecida con éxito.");

            // Cerrar la conexión cuando termines
            conn.close();
        } catch (NamingException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

13 JAAS

13.1 Introducción a JAAS

JAAS (Java Authentication and Authorization Service) es un framework de seguridad integrado en Java que proporciona un mecanismo robusto y flexible para la autenticación y autorización de usuarios en aplicaciones Java. Fue introducido como parte de **Java SE** desde la versión 1.4, con la intención de desacoplar los mecanismos de seguridad de la lógica de negocio de la aplicación, ofreciendo una capa independiente y extensible.

13.2 Propósito

- **Autenticación (Authentication)**: JAAS verifica la identidad de los usuarios que intentan acceder a la aplicación. Esto se logra mediante la integración con diferentes tipos de sistemas de autenticación (por ejemplo, LDAP, bases de datos, archivos de texto, Kerberos, etc.), usando una interfaz unificada.
- **Autorización (Authorization)**: Una vez autenticado un usuario, JAAS controla los permisos que dicho usuario tiene en función de roles o privilegios específicos. Esto asegura que los usuarios solo puedan acceder a los recursos y funciones que están autorizados a usar.

13.3 RBAC

El Control de Acceso Basado en Roles o **RBAC** por sus siglas en inglés, es uno de los paradigmas de seguridad más comunes en las aplicaciones de hoy en día. El principio básico consiste en otorgar a cada usuario únicamente los permisos imprescindibles para desarrollar las funciones asociadas a la posición o rol que cumplen dentro de la organización.

RBAC en el Desarrollo Web: Cuando hablamos de RBAC en el contexto del desarrollo web hay que hacer, eso sí, una distinción clara entre lo que esto significa en el backend y lo que significa en el frontend. En el contexto del backend hablamos propiamente de ese sistema de seguridad en el que un usuario solo podrá acceder a ciertos recursos si está debidamente identificado y tiene otorgados los permisos para acceder a ellos.

Pero en el **contexto del frontend** debido a que el código fuente de la aplicación es accesible por el usuario, no podemos hablar de RBAC como un sistema de seguridad ya que sería relativamente sencillo sobrepasar para todo aquel usuario con un mínimo de conocimiento que supiera dónde mirar. Por ello, en el frontend tenemos que considerarlo más como una forma de mejorar la experiencia del usuario (**UX**) que como un sistema de seguridad.

13.4 Arquitectura

JAAS sigue una arquitectura modular y flexible, lo que permite utilizar diferentes tipos de **LoginModules** para autenticar usuarios en diferentes entornos o mediante diferentes mecanismos.

JAAS utiliza dos conceptos clave:

1. **Subject**: Representa una entidad que intenta acceder a la aplicación (por ejemplo, un usuario). Un **Subject** puede tener múltiples credenciales (como un nombre de usuario, una contraseña, un certificado digital, etc.).
2. **Principal**: Es una identidad asociada a un **Subject** después de la autenticación. Por ejemplo, puede representar el nombre de usuario o algún otro tipo de identificación, como un rol o permiso.

13.5 Componentes

1. **LoginModule**: Es el componente que se encarga del proceso de autenticación en JAAS. Los módulos de inicio de sesión (LoginModules) son la parte fundamental del sistema de autenticación. JAAS soporta varios mecanismos de autenticación, como bases de datos, LDAP, Kerberos, archivos, etc. Un **LoginModule** puede autenticar usuarios utilizando cualquier método específico de la aplicación, y los desarrolladores pueden implementar sus propios **LoginModules** personalizados si fuera necesario.
2. **Subject**: Un **Subject** es una entidad (normalmente un usuario) que ha sido autenticada en la aplicación. El **Subject** contiene información de identidad del usuario (Principal) y cualquier otra credencial necesaria para la autenticación.
3. **Principal**: Es una entidad abstracta que representa una identidad del usuario autenticado. Por ejemplo, podría ser el nombre del usuario o un rol de seguridad.
4. **Policy**: Define el modelo de autorización, que puede estar basado en permisos. Una **Policy** en JAAS se encarga de definir qué roles o identidades tienen acceso a recursos específicos en la aplicación.
5. **AccessControlContext**: Es el contexto que contiene la información de control de acceso. Cuando se ejecuta una operación que necesita autenticación o autorización, JAAS consulta el **AccessControlContext** para verificar si el sujeto tiene los permisos necesarios.

13.6 Ciclo de Autenticación

1. **LoginContext**: El punto de entrada principal para iniciar el proceso de autenticación en JAAS es la clase **LoginContext**. Se encarga de gestionar el proceso de autenticación y comunicar la información entre los **LoginModules** y el **Subject**.
2. **Proceso de Autenticación**:
    - **LoginContext.login()**: Se crea una instancia de **LoginContext** que invoca el método login(). Esto inicia el proceso de autenticación.
    - Los **LoginModules** configurados en el archivo de configuración JAAS son ejecutados en el orden especificado. Cada **LoginModule** trata de autenticar al usuario.
    - Si la autenticación tiene éxito, se almacena un **Subject** con los **Principals** correspondientes.
3. **Autorización**:
    - Una vez que un **Subject** ha sido autenticado, la aplicación puede verificar si el **Subject** tiene los permisos necesarios para realizar una acción. Esto se logra utilizando el marco de seguridad basado en permisos de Java (**SecurityManager** y **AccessController**).
    - Si el usuario tiene los permisos necesarios, la acción solicitada se realiza, de lo contrario, se deniega el acceso.

13.7 Anotaciones

Anotación	Descripción	Ámbito de Uso
@RolesAllowed	Especifica los roles permitidos para acceder a un método o clase.	Type, Method
@PermitAll	Permite que todos los usuarios accedan al método o clase.	Type, Method
@DenyAll	Niega el acceso a todos los usuarios para un método o clase.	Type, Method
@RunAs	Define un rol que el EJB o Servlet debe asumir durante su ejecución para realizar tareas con ese rol.	Type
@DeclareRoles	Declara los roles de seguridad que se pueden usar dentro de la aplicación (usualmente en Servlets).	Type
@LoginConfig	Configura el tipo de autenticación usado en una aplicación web (por ejemplo, FORM, BASIC).	Type (Clase Servlet)
@SecurityDomain (JBoss/WildFly)	Especifica el dominio de seguridad (security domain) que se debe usar para la autenticación.	Type

1. **@RolesAllowed**:

- Restringe el acceso a los métodos o clases a los usuarios que pertenezcan a los roles especificados.
- Es utilizado en aplicaciones Java EE, como EJBs y Servlets.

```
@RolesAllowed({"admin", "user"})
public void secureMethod() {
    // Código accesible solo por usuarios con roles admin o user
}
```

2. **@PermitAll**:
   - Permite el acceso a todos los usuarios a un método o clase sin restricciones.
   - Útil para abrir el acceso a ciertos métodos en aplicaciones seguras.

```
@PermitAll
public void openMethod() {
    // Código accesible por cualquier usuario
}
```

3. **@DenyAll**:
   - Restringe el acceso a todos los usuarios para un método o clase.
   - Puede usarse para bloquear el acceso en ciertas condiciones.

```
@DenyAll
public void restrictedMethod() {
    // Nadie puede acceder a este método
}
```

4. **@RunAs**:
   - Define un rol temporal que el componente (EJB o Servlet) debe asumir durante su ejecución, lo que permite al componente realizar operaciones con privilegios adicionales.

```
@RunAs("admin")
public class AdminComponent {
```

```
    // Este componente actuará como 'admin'
}
```

5. **@DeclareRoles**:
   - Se usa para declarar los roles de seguridad disponibles en la aplicación, generalmente en el contexto de Servlets o aplicaciones web.

```
@DeclareRoles({"admin", "user"})
public class MyServlet extends HttpServlet {
    // Este servlet puede verificar estos roles
}
```

6. **@LoginConfig**:
   - Configura el mecanismo de autenticación que será usado en una aplicación web, como FORM o BASIC authentication.

```
@LoginConfig(authMethod = "FORM", realmName = "myRealm")
public class MyWebAppConfig {
    // Configuración de autenticación
}
```

7. **@SecurityDomain** (Específico de JBoss/WildFly):
   - Anotación propia de los servidores de aplicaciones JBoss/WildFly que especifica el dominio de seguridad que debe ser utilizado por el EJB para la autenticación y autorización.

```
@SecurityDomain("myDomain")
public class SecureBean {
    // Bean que utiliza el dominio de seguridad 'myDomain'
}
```

13.8 Ejemplo

1. **Configuración de JAAS**: El primer paso es crear un archivo de configuración que describa los **LoginModules** que se deben usar para la autenticación. Un

archivo típico de configuración de JAAS, con extensión .config o .login, puede parecerse a esto:

```
EjemploJAAS {
   com.example.MyLoginModule required;
};
```

2. **Creación de un LoginModule**: Se puede crear un **LoginModule** personalizado implementando la interfaz javax.security.auth.spi.LoginModule. Esta interfaz define cuatro métodos principales:
   - **initialize()**: Inicializa el LoginModule con la información proporcionada (como el Subject y los callback handlers).
   - **login()**: Realiza el proceso de autenticación.
   - **commit()**: Si la autenticación fue exitosa, se confirma y se guarda la identidad del usuario (el Principal).
   - **abort() y logout()**: Manejan la cancelación o finalización del proceso de autenticación.
   - **LoginModule**:

```
import javax.security.auth.spi.LoginModule;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import java.util.Map;


public class MyLoginModule implements LoginModule {
   private Subject subject;
   private CallbackHandler callbackHandler;
   private String username;
   private boolean authenticationSuccess;
```

```
    @Override
        public void initialize(Subject subject, CallbackHandler
callbackHandler,
                               Map<String, ?> sharedState, Map<String, ?>
options) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
    }

    @Override
    public boolean login() throws LoginException {
        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("Username: ");
        callbacks[1] = new PasswordCallback("Password: ", false);

        try {
            callbackHandler.handle(callbacks);
            String name = ((NameCallback) callbacks[0]).getName();
                               char[] password = ((PasswordCallback)
callbacks[1]).getPassword();

            // Autenticación ficticia
                if ("admin".equals(name) && "password".equals(new
String(password))) {
                username = name;
                authenticationSuccess = true;
```

```
                return true;
            } else {
                throw new LoginException("Invalid credentials");
            }
        } catch (Exception e) {
            throw new LoginException("Error handling callbacks: " +
e.getMessage());
        }
    }

    @Override
    public boolean commit() throws LoginException {
        if (authenticationSuccess) {
            // Crear un Principal para el Subject
            subject.getPrincipals().add(() -> username);
            return true;
        } else {
            return false;
        }
    }

    @Override
    public boolean abort() throws LoginException {
        return false;
    }
```

```
    @Override
    public boolean logout() throws LoginException {
        return false;
    }
}
```

3. **Autenticación en la Aplicación**: Para usar JAAS en tu aplicación, es imprescindible crear un **LoginContext** que inicie el proceso de autenticación:

```
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;


public class Main {
    public static void main(String[] args) {
        try {
            LoginContext loginContext = new
LoginContext("EjemploJAAS");
            loginContext.login(); // Autenticación
            System.out.println("Autenticación exitosa");
        } catch (LoginException e) {
            System.out.println("Error de autenticación: " +
e.getMessage());
        }
    }
}
```

13.9 Integración con Servidores de Aplicaciones

En aplicaciones empresariales, los servidores de aplicaciones como **WildFly**, **Tomcat** o **GlassFish** usan JAAS para la seguridad, integrando la autenticación con sistemas de seguridad como LDAP, Kerberos o bases de datos.

1. **Autenticación Web**: En aplicaciones web, JAAS puede integrarse con el contenedor de servlets para autenticar usuarios que acceden a la aplicación mediante **formularios web** o autenticación basada en certificados.
2. **EJB y JAAS**: Los **Enterprise JavaBeans (EJB)** usan JAAS para manejar la autenticación y autorización de los usuarios que interactúan con los servicios EJB.

14 JAX-WS

14.1 Introducción a JAX-WS

JAX-WS (Java API for XML Web Services) es una API de Java que permite la creación, invocación y consumo de servicios web basados en **SOAP (Simple Object Access Protocol)**. Es una tecnología estándar que facilita la interoperabilidad entre sistemas distribuidos en distintas plataformas y lenguajes, utilizando SOAP sobre HTTP, SMTP u otros protocolos de transporte.

14.2 SOAP

El **SOAP** (protocolo de acceso a objetos simples) es el protocolo de aplicaciones fundacional basado en **XML** y usado para implementar servicios web dentro de una SOA (arquitectura orientada a servicios). El SOAP se transporta principalmente a través de **HTTP** y sistemas de mensajería middleware (**JMS, MQ Series, MSMQ, Tuxedo, TIBCO RV**), pero también se puede ser transportar a través de otros protocolos como **SMTP** (protocolo para transferencia simple de correo) y **FTP** (protocolo de transferencia de archivos).

Los mensajes SOAP suelen incluir los siguientes elementos:

- **El sobre:** el sobre SOAP es el elemento raíz de un mensaje SOAP y es necesario. En esencia, el sobre contiene el mensaje SOAP igual que un sobre tradicional contiene una carta escrita.
- **Encabezado**: el encabezado SOAP es opcional y, cuando está presente, contiene información específica de la aplicación como los detalles de autenticación, direccionamiento y enrutamiento.
- **Cuerpo**: el cuerpo SOAP es necesario y contiene el mensaje de la aplicación que se está transportando, incluidos la operación remota específica que se invoca y los datos (parámetros) que se intercambian.

Los mensajes SOAP suelen ser grandes, ya que deben contener la información que las aplicaciones y los clientes necesitan para analizar los datos que contienen y ejecutar la lógica apropiada. Cuanto más grande es el mensaje, más procesamiento requiere del servidor, lo que aumenta el consumo de recursos y disminuye la capacidad total. El aumento de tamaño también puede tener un efecto adverso sobre el rendimiento de aplicaciones desarrolladas sobre SOAP, ya que se necesitan más recursos de red para transferir los mensajes.

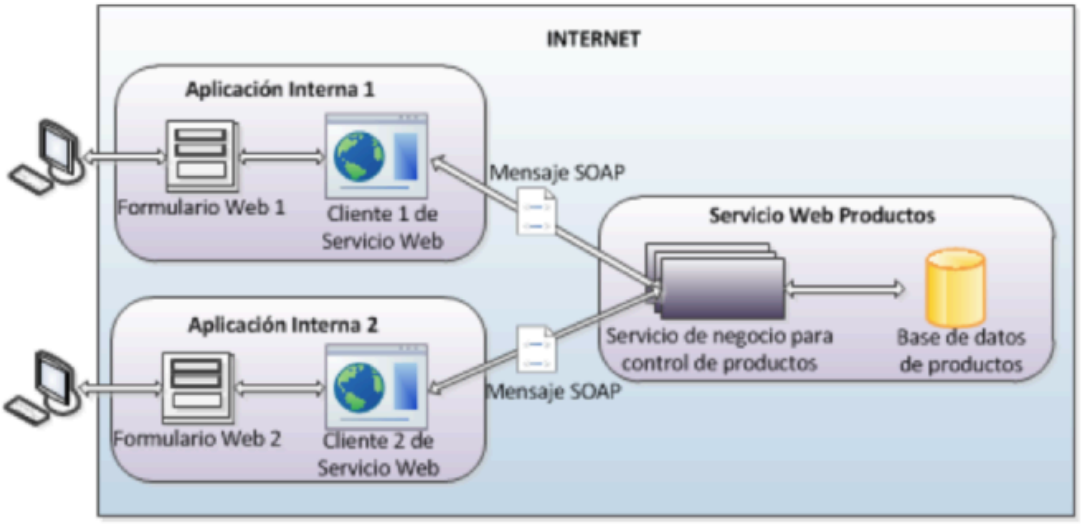
Dado que SOAP está basado en XML, es susceptible a diversos ataques y vulnerabilidades relacionados con XML, y más vulnerable aún a los ataques asociados a su protocolo de capa de transporte, normalmente HTTP.

Los mensajes SOAP se estructuran con etiquetas XML además de que interviene un lenguaje adicional para su definición, **WSDL** (Web Services Description Language) que se usa para generar una interfaz del Servicio Web, misma que será su punto de entrada. A éste se le conoce como Archivo wsdl o Contrato y una vez que el servicio está publicado, éste archivo estará accesible en una red por medio de un url que tiene terminación ?wsdl

Dentro del archivo wsdl se describe todo el servicio web, las operaciones que tiene disponibles, las estructuras de datos de los mensajes de cada operación, entre otras cosas que mencionaré más adelante. Aquí también se utilizan los XSD, que son los archivos que especifican los datos de los mensajes. De igual manera, los explicaré a detalle más adelante.

Para poder invocar un Servicio Web, es necesario crear un Cliente Consumidor que use el archivo wsdl, lo interprete correctamente y con toda la información que éste contiene, sea capaz de conectarse a él, invocarlo e intercambiar los mensajes en el formato esperado.

Los Servicios Web SOAP tienen la propiedad de Interoperabilidad, que significa que pueden ser invocados sin importar la plataforma o el lenguaje que se esté usando, siempre y cuando estos sean capaces de generar un Cliente Consumidor con el mismo estándar del protocolo SOAP, es por eso que la construcción del cliente varía dependiendo de la plataforma y el lenguaje de programación. Así, una aplicación Java puede consumir el mismo Servicio Web que una aplicación .NET, un sistema empaquetado (Siebel, EBS, etc) entre muchos otros.



14.3 Anotaciones

Anotación	Descripción	Ámbito de Uso	Ejemplo
@WebService	Declara una clase como servicio web. Define el endpoint del servicio y puede especificar su nombre y namespace.	Type	**@WebService(name = "CalculadoraService", targetNamespace = "http://servicios.miempresa.com/")**
@WebMethod	Expone un método Java como una operación del servicio web. Se usa para métodos que deben ser accesibles como parte del servicio.	Method	**@WebMethod(operationName = "sumar") public int suma(int a, int b) { return a + b; }**
@WebParam	Define los parámetros de entrada de un método expuesto en el servicio web, especificando su nombre y tipo.	Parameter	**@WebMethod public int suma(@WebParam(name = "a") int num1, @WebParam(name = "b") int num2)**

@WebResult	Especifica el valor de retorno del método expuesto como servicio web, definiendo el nombre del resultado en el mensaje SOAP.	Return	**@WebMethod @WebResult(name = "resultado") public int suma(int a, int b)**
@SOAPBinding	Configura el estilo de mensajes SOAP (RPC o Document) y cómo se estructuran (literal o codificado).	Type, Method	**@SOAPBinding(style = Style.DOCUMENT, use = Use.LITERAL)**
@Oneway	Declara que un método no tiene respuesta, es decir, es unidireccional (equivalente a un método void).	Method	**@WebMethod @Oneway public void enviarNotificacion(String mensaje)**
@HandlerChain	Define la ubicación de un archivo XML que especifica la cadena de manejadores (handlers) para interceptar mensajes SOAP.	Type, Method	**@HandlerChain(file="handlers.xml")**
@XmlSeeAlso	Especifica clases adicionales que deben ser consideradas durante la serialización y deserialización de XML en JAX-WS.	Type	**@XmlSeeAlso({Cliente.class, Pedido.class})**
@WebServiceRef	Inyecta una referencia a un servicio web JAX-WS en un componente o clase.	Field, Method, Parameter	**@WebServiceRef(CalculadoraService.class) private CalculadoraService service;**

@WebServiceClient	Define una clase generada a partir de un WSDL que actúa como cliente para consumir un servicio web.	Type	**Se genera automáticamente al usar wsimport o por herramientas como wsgen.**
@RequestWrapper	Define el nombre y la clase de solicitud del mensaje SOAP entrante para un método de servicio web.	Method	**@RequestWrapper(localName = "SumaRequest", className = "com.miempresa.SumaRequest")**
@ResponseWrapper	Define el nombre y la clase de respuesta del mensaje SOAP saliente para un método de servicio web.	Method	**@ResponseWrapper(localName = "SumaResponse", className = "com.miempresa.SumaResponse")**
@Action	Define las acciones SOAP para los mensajes entrantes y salientes, relacionadas con WS-Addressing.	Method	**@Action(input = "http://servicio/Entrada", output = "http://servicio/Salida")**
@FaultAction	Especifica las acciones SOAP que se ejecutan cuando ocurre una excepción definida por el usuario.	Method	**@FaultAction(className = "com.miempresa.FaultClase", value = "http://servicio/FaultAction")**
@WebFault	Define una clase personalizada de excepción para mapear errores SOAP en excepciones Java.	Type	**@WebFault(name = "ClienteNotFoundException", faultBean = "com.miempresa.ClienteNotFound")**

@MTOM	Activa el envío optimizado de archivos binarios mediante el uso del protocolo MTOM (Message Transmission Optimization Mechanism).	Type	**@MTOM(enabled = true, threshold = 1024)**
@BindingType	Configura el tipo de enlace del servicio web, por ejemplo, para cambiar el protocolo de SOAP a JSON o HTTP.	Type	**@BindingType(value = SOAPBinding.SOAP12HTTP_BINDING)**
@WebEndpoint	Define un punto de acceso alternativo dentro de un servicio web.	Method	**@WebEndpoint(name = "CalculadoraPort")**
@ServiceMode	Especifica si el servicio web opera en modo mensaje o modo de servicio (message vs. payload).	Type	**@ServiceMode(value = Service.Mode.MESSAGE)**
@Addressing	Activa WS-Addressing, que añade metadatos de direccionamiento (como las cabeceras ReplyTo) a los mensajes SOAP.	Type	**@Addressing(enabled = true, required = false)**

14.4 Ejemplo

1. **Creación de la implementación del servicio web**:
   - Se escribe una clase Java que implementa la lógica de negocio del servicio.
   - Se marca esta clase con @WebService, lo que indica que es un servicio web SOAP.

```
@WebService
public class CalculadoraService {
   @WebMethod
   public int suma(int a, int b) {
      return a + b;
   }
}
```

2. **Generación del WSDL**:
   - El contenedor JAX-WS (como GlassFish, Apache TomEE o WildFly) genera automáticamente el archivo WSDL que describe el servicio y sus operaciones.
   - También es posible generar manualmente el WSDL utilizando la herramienta wsgen.
3. **Exposición del servicio**:
   - El servicio web se despliega en un servidor de aplicaciones que soporte JAX-WS.
   - También puede exponerse directamente utilizando un **Endpoint**:

```
public class Publicador {
   public static void main(String[] args) {
      Endpoint.publish("http://localhost:8080/CalculadoraService",
new CalculadoraService());
   }
}
```

4. **Invocación desde un cliente**:

- El cliente genera las clases necesarias para interactuar con el servicio utilizando la herramienta wsimport, que toma el WSDL del servicio y genera las clases Java.
- Luego, el cliente puede invocar los métodos remotos del servicio de la siguiente manera:

```
URL url = new
URL("http://localhost:8080/CalculadoraService?wsdl");
```

```
QName qname = new QName("http://service.example.com/",
"CalculadoraService");

Service service = Service.create(url, qname);

CalculadoraService calc =
service.getPort(CalculadoraService.class);


int resultado = calc.suma(5, 10);

System.out.println("Resultado: " + resultado);
```

15 JAX-RS

15.1 Introducción a JAX-RS

JAX-RS (Java API for RESTful Web Services) es una especificación de Java que proporciona una API para desarrollar servicios web basados en el estilo arquitectónico **REST (Representational State Transfer)**. REST es una arquitectura utilizada para interactuar con recursos de manera sencilla a través del protocolo HTTP, aprovechando los métodos HTTP estándar como GET, POST, PUT, DELETE, etc.

JAX-RS simplifica el desarrollo de estos servicios en aplicaciones Java, proporcionando anotaciones para mapear clases y métodos a solicitudes HTTP y trabajar fácilmente con recursos representados en formatos como JSON o XML.

15.2 API RESTful

REST no es un protocolo ni un estándar, sino más bien un conjunto de límites de arquitectura. Los desarrolladores de las API pueden implementarlo de distintas maneras.

Cuando el cliente envía una solicitud a través de una API de RESTful, ésta transfiere una representación del estado del recurso requerido a quien lo haya solicitado o al extremo. La información se entrega por medio de HTTP en uno de estos formatos: **JSON** (JavaScript Object Notation), **HTML**, **XLT**, **Python**, **PHP** o texto sin formato. JSON es el lenguaje de programación más popular, ya que tanto las máquinas como las personas lo pueden comprender y no depende de ningún lenguaje, a pesar de que su nombre indique lo contrario.

También es necesario tener en cuenta otros aspectos. Los encabezados y los parámetros también son importantes en los métodos HTTP de una solicitud HTTP de la API de RESTful, ya que contienen información de identificación importante con respecto a los metadatos, la autorización, el identificador uniforme de recursos (URI), el almacenamiento en caché, las cookies y otros elementos de la solicitud. Hay encabezados de solicitud y de respuesta, pero cada uno tiene sus propios códigos de estado e información de conexión HTTP.

Para que una API se considere de RESTful, debe cumplir los siguientes criterios:

- Arquitectura cliente-servidor compuesta de clientes, servidores y recursos, con la gestión de solicitudes a través de HTTP.
- Comunicación entre el cliente y el servidor sin estado, lo cual implica que no se almacena la información del cliente entre las solicitudes de GET y que cada una de ellas es independiente y está desconectada del resto.
- Datos que pueden almacenarse en caché y optimizan las interacciones entre el cliente y el servidor.
- Una interfaz uniforme entre los elementos, para que la información se transfiera de forma estandarizada. Para ello deben cumplirse las siguientes condiciones:
    1. Los recursos solicitados deben ser identificables e independientes de las representaciones enviadas al cliente.
    2. El cliente debe poder manipular los recursos a través de la representación que recibe, ya que esta contiene suficiente información para permitirlo.
    3. Los mensajes autodescriptivos que se envíen al cliente deben contener la información necesaria para describir cómo debe procesarla.
    4. Debe contener hipertexto o hipermedios, lo cual significa que cuando el cliente acceda a algún recurso, debe poder utilizar hipervínculos para buscar las demás acciones que se encuentren disponibles en ese momento.
    5. Un sistema en capas que organiza en jerarquías invisibles para el cliente cada uno de los servidores (los encargados de la seguridad, del equilibrio de carga, etc.) que participan en la recuperación dez la información solicitada.
    6. Código disponible según se solicite (opcional), es decir, la capacidad para enviar códigos ejecutables del servidor al cliente cuando se requiera, lo cual amplía las funciones del cliente.

Si bien la API de REST debe cumplir todos estos parámetros, resulta más fácil de usar que un protocolo definido previamente, como SOAP (protocolo simple de acceso a objetos), el cual tiene requisitos específicos, como la mensajería XML y la

seguridad y el cumplimiento integrados de las operaciones, que lo hacen más lento y pesado.

Por el contrario, REST es un conjunto de pautas que pueden implementarse según sea necesario. Por esta razón, las API de REST son más rápidas y ligeras, cuentan con mayor capacidad de ajuste y, por ende, resultan ideales para el Internet de las cosas (IoT) y el desarrollo de aplicaciones para dispositivos móviles.

15.3 Características

1. **Uso de anotaciones**: JAX-RS utiliza un conjunto de anotaciones para definir los servicios RESTful. Estas anotaciones permiten que las clases Java se conviertan en endpoints REST.
2. **Modelado basado en recursos**: En REST, los servicios se modelan como recursos, y las operaciones en estos recursos (CRUD) se mapean a los métodos HTTP estándar.
3. **Compatibilidad con múltiples formatos de datos**: JAX-RS permite manejar datos en diversos formatos como JSON, XML, texto plano, entre otros. La conversión entre objetos Java y estos formatos se realiza mediante los proveedores de contenido (MessageBodyReader y MessageBodyWriter).
4. **Inyección de dependencias**: Utiliza la inyección de dependencias para manejar recursos como parámetros de la solicitud HTTP, contexto de seguridad, URI, etc.

15.4 Anotaciones

Anotación	Descripción	Ámbito de Uso	Ejemplo
@Path	Define la ruta (URI) del recurso.	Class, Method	**@Path("/productos")**
@GET	Mapea un método para manejar solicitudes HTTP GET.	Method	**@GET public List<Product> obtenerProductos()**
@POST	Mapea un método para manejar solicitudes HTTP POST.	Method	**@POST public Response crearProducto(Product producto)**
@PUT	Mapea un método para manejar solicitudes HTTP PUT.	Method	**@PUT @Path("/{id}") public Response actualizarProducto(@PathParam("id") int id, Product producto)**

@DELETE	Mapea un método para manejar solicitudes HTTP DELETE.	Method	**@DELETE @Path("/{id}") public Response eliminarProducto(@PathParam("id") int id)**
@HEAD	Mapea un método para manejar solicitudes HTTP HEAD.	Method	**@HEAD public Response obtenerEncabezados()**
@OPTIONS	Mapea un método para manejar solicitudes HTTP OPTIONS.	Method	**@OPTIONS public Response obtenerOpciones()**
@PATCH	Mapea un método para manejar solicitudes HTTP PATCH.	Method	**@PATCH public Response actualizarParcialProducto(Product producto)**
@PathParam	Inyecta el valor de un parámetro de ruta en el método.	Parameter	**@PathParam("id") int id**
@QueryParam	Inyecta un valor de parámetro de consulta de la URL.	Parameter	**@QueryParam("categoria") String categoria**
@FormParam	Inyecta un valor enviado en un formulario HTML (application/x-www-form-urlencoded).	Parameter	**@FormParam("nombre") String nombre**
@HeaderParam	Inyecta un valor de un encabezado HTTP.	Parameter	**@HeaderParam("User-Agent") String userAgent**
@CookieParam	Inyecta un valor de una cookie HTTP.	Parameter	**@CookieParam("sesionID") String sesionID**
@MatrixParam	Inyecta un valor de un parámetro de matriz de la URI (usado en URLs con segmentos de matriz).	Parameter	**@MatrixParam("color") String color**

@Consumes	Especifica los tipos de contenido que puede aceptar el método.	Class, Method	**@Consumes(MediaType.APPLICATION_JSON)**
@Produces	Especifica los tipos de contenido que puede devolver el método.	Class, Method	**@Produces(MediaType.APPLICATION_JSON)**
@DefaultValue	Especifica un valor por defecto si el parámetro no está presente en la solicitud.	Parameter	**@DefaultValue("10") @QueryParam("limite") int limite**
@Context	Inyecta información de contexto del entorno de ejecución, como el HttpServletRequest o la URI.	Parameter	**@Context HttpServletRequest request**
@Produces	Especifica el formato de respuesta del recurso.	Class, Method	**@Produces(MediaType.TEXT_XML)**
@Consumes	Define el tipo de entrada aceptado por el recurso.	Class, Method	**@Consumes(MediaType.APPLICATION_JSON)**
@BeanParam	Permite agrupar varios parámetros en un único objeto.	Parameter	**@BeanParam Producto producto**
@ApplicationPath	Define la ruta base para la aplicación JAX-RS.	Class	**@ApplicationPath("/api")**
@Provider	Marca una clase como un proveedor de servicios (ej., filtros, excepciones).	Class	**@Provider**
@ExceptionMapper	Mapea excepciones a respuestas HTTP personalizadas.	Class	**@Provider public class MyExceptionMapper implements ExceptionMapper<MyException>**

@Encoded	Evita la decodificación automática de un parámetro de entrada.	Parameter	**@Encoded @QueryParam("url") String url**
@Produces	Define el tipo de medios que puede generar un recurso o método.	Class, Method	**@Produces("application/json")**
@Suspend	Utilizado en el manejo de solicitudes asíncronas.	Method	**public void myMethod(@Suspended AsyncResponse asyncResponse)**
@StreamingOutput	Proporciona una manera de generar una salida en streaming.	Return Type	**public StreamingOutput generarReporte()**
@RequestScoped	Define que el recurso o bean está en el ámbito de la solicitud.	Class	**@RequestScoped public class MyResource {}**

15.5 Ejemplo

1. **URI de recursos**

En JAX-RS, los recursos RESTful se identifican mediante **URI** (Uniform Resource Identifier), y se define la ruta de acceso a esos recursos usando la anotación @Path. Los métodos que manipulan estos recursos están asociados a métodos HTTP específicos.

- La ruta /productos define un recurso de productos, con diferentes métodos HTTP como GET, POST, PUT, y DELETE.
- Las anotaciones @Produces y @Consumes especifican que los datos entrantes y salientes están en formato **JSON**

```
@Path("/productos")
public class ProductoResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Producto> obtenerProductos() {
        // Lógica para obtener y devolver la lista de productos.
    }

    @POST
```

```
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response crearProducto(Producto producto) {
        // Lógica para crear un nuevo producto.
    }

    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response actualizarProducto(@PathParam("id") int id,
Producto producto) {
        // Lógica para actualizar un producto.
    }

    @DELETE
    @Path("/{id}")
    public Response eliminarProducto(@PathParam("id") int id) {
        // Lógica para eliminar un producto.
    }
}
```

2. **Inyección de parámetros**

JAX-RS permite acceder a diferentes partes de la solicitud HTTP mediante inyección de dependencias con anotaciones como @PathParam, @QueryParam, @HeaderParam, y @FormParam.

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Producto obtenerProductoPorId(@PathParam("id") int id) {
    // Lógica para obtener el producto por ID.
}
```

3. **Formato de datos**

JAX-RS admite diferentes formatos de datos como **JSON**, **XML**, **Texto Plano**, y otros mediante las anotaciones @Produces y @Consumes.

- @Consumes: Define el tipo de datos que el servicio web puede recibir (por ejemplo, application/json).
- @Produces: Define el tipo de datos que el servicio web puede devolver (por ejemplo, application/xml o application/json).

Por ejemplo, si un método está diseñado para recibir y devolver datos en formato JSON:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response crearProducto(Producto producto) {
   // Lógica para crear el producto.
}
```

4. **Excepciones y Manejo de Errores**

JAX-RS también proporciona mecanismos para manejar errores y excepciones en aplicaciones RESTful, utilizando clases como Response y excepciones personalizadas.

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response obtenerProductoPorId(@PathParam("id") int id) {
   Producto producto = productoService.find(id);
   if (producto == null) {
      return
Response.status(Response.Status.NOT_FOUND).entity("Producto no
encontrado").build();
   }
   return Response.ok(producto).build();
}
```

5. **Integración con CDI (Contexts and Dependency Injection)**

JAX-RS se integra perfectamente con CDI, lo que permite inyectar servicios, controladores o cualquier otro bean manejado dentro de las clases de recursos.

El ciclo de vida de los recursos en JAX-RS es manejado por el contenedor. Los recursos pueden tener diferentes ámbitos de vida:

- **Por solicitud (RequestScoped)**: Un nuevo recurso se crea para cada solicitud HTTP.
- **Singleton**: El recurso es un único objeto compartido entre todas las solicitudes, utilizando @Singleton.

```
@Path("/productos")
@RequestScoped
public class ProductoResource {

   @Inject
```

```
    private ProductoService productoService;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Producto> obtenerProductos() {
        return productoService.obtenerTodos();
    }
}
```