

JSF PASO A PASO

Entorno: Netbeans

Requisitos: Tener un servidor de aplicaciones configurado (WildFly, Glassfish, Tomcat...)

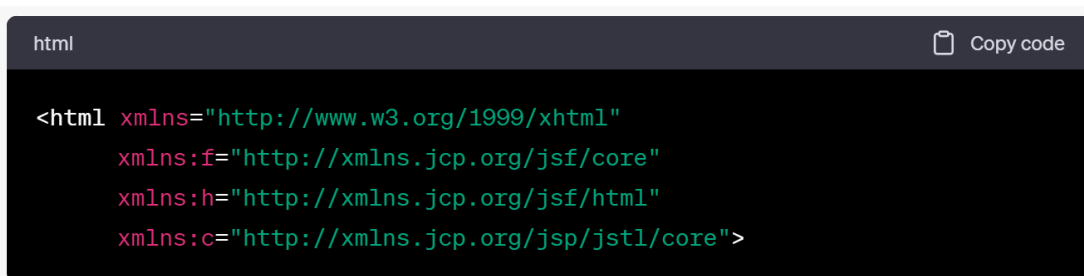
Descripción del ejemplo: Vamos a desarrollar una página que pide los datos de un jugador de fútbol. Una vez que hemos introducido los datos correspondientes (nombre, posición, equipo, edad y si el jugador está activo), al hacer clic en el botón iremos a otra página web que nos mostrará en una tabla todos los jugadores que coincidan con el equipo indicado. En caso de no rellenar alguno de los campos, la página se refrescará mostrándonos un mensaje de alerta.

Paso 1: Crear un nuevo proyecto web en NetBeans

- Abre NetBeans y selecciona "**Archivo**" en la barra de menú.
- Luego, elige "**Nuevo Proyecto**" y selecciona "**Java Web**" y "**Aplicación web**" en las categorías.
- Haz clic en "**Siguiente**" y proporciona un nombre para tu proyecto web.
- Selecciona el servidor que desees utilizar (por ejemplo, GlassFish o Apache Tomcat) y haz clic en "**Siguiente**".
- En la página "**Configuración del servidor**", deja los valores predeterminados y haz clic en "**Siguiente**".
- En la página "**Frameworks y tecnologías**", selecciona "**JavaServer Faces**" y haz clic en "**Finalizar**".

Paso 2: Crear una página XHTML para ingresar los detalles del jugador

- Abre el archivo "**index.xhtml**" en el editor de NetBeans.
- Agrega las siguientes declaraciones de espacio de nombres (**xmlns:f** y **xmlns:c**) en la etiqueta `<html>`:

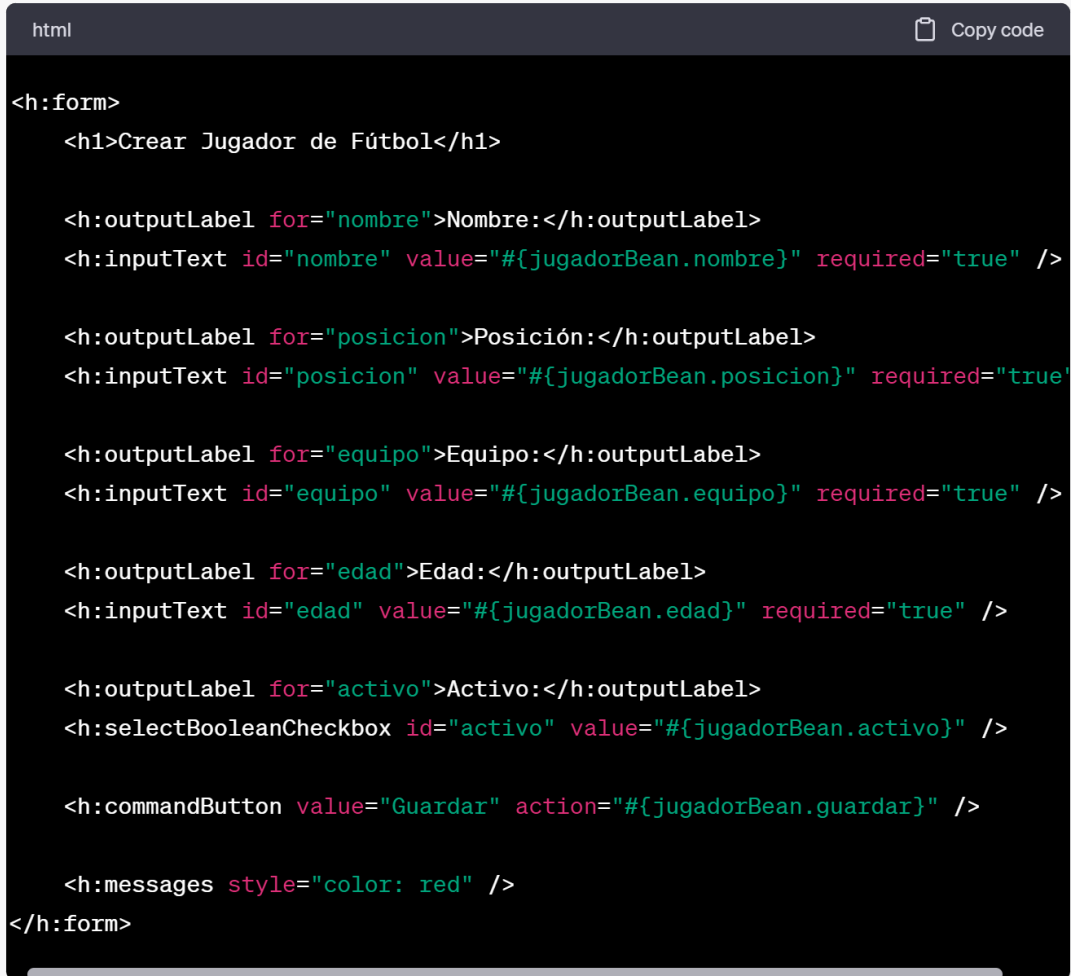


```
html Copy code  
  
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:f="http://xmlns.jcp.org/jsf/core"  
      xmlns:h="http://xmlns.jcp.org/jsf/html"  
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
```

Estas declaraciones permiten el uso de las bibliotecas de etiquetas JSF y JSTL (JavaServer Pages)

Standard Tag Library) que facilitan la implementación de funcionalidades y permiten añadir estructuras lógicas.

- A continuación, crea un formulario para ingresar los detalles del jugador:

A screenshot of a code editor with a dark theme. The editor shows HTML code for a form titled 'Crear Jugador de Fútbol'. The code includes labels and input fields for 'Nombre', 'Posición', 'Equipo', and 'Edad', each with a corresponding input text field. There is also a boolean checkbox for 'Activo' and a 'Guardar' button. The code uses JSP tags and EL expressions like `#{jugadorBean.nombre}`. A messages tag is at the bottom. The editor has a 'Copy code' button in the top right corner.

```
html
Copy code

<h:form>
  <h1>Crear Jugador de Fútbol</h1>

  <h:outputLabel for="nombre">Nombre:</h:outputLabel>
  <h:inputText id="nombre" value="#{jugadorBean.nombre}" required="true" />

  <h:outputLabel for="posicion">Posición:</h:outputLabel>
  <h:inputText id="posicion" value="#{jugadorBean.posicion}" required="true" />

  <h:outputLabel for="equipo">Equipo:</h:outputLabel>
  <h:inputText id="equipo" value="#{jugadorBean.equipo}" required="true" />

  <h:outputLabel for="edad">Edad:</h:outputLabel>
  <h:inputText id="edad" value="#{jugadorBean.edad}" required="true" />

  <h:outputLabel for="activo">Activo:</h:outputLabel>
  <h:selectBooleanCheckbox id="activo" value="#{jugadorBean.activo}" />

  <h:commandButton value="Guardar" action="#{jugadorBean.guardar}" />

  <h:messages style="color: red" />
</h:form>
```

En el ejemplo de código, podemos observar que en los atributos `value` ponemos cosas como; **`#{jugadorBean.nombre}`**. Esto se llama **Expression Language** o **EL**, es un lenguaje de scripting que se utiliza en tecnologías web basadas en Java, como **JavaServer Faces (JSF)**, para acceder y manipular datos en tiempo de ejecución. En este caso estamos utilizando la **EL** para acceder al atributo `nombre` del objeto `jugadorBean`. Aquí hay una explicación de cómo funciona:

`#{}`: La EL se coloca entre llaves y comienza con `#{}`. Esto indica que se está utilizando una expresión EL.

`jugadorBean`: Es una referencia al nombre del CDI Bean que estamos utilizando. En este caso, estamos accediendo a las propiedades del CDI Bean llamado `jugadorBean`. (Al final de este "tutorial paso a paso" os dejo información detallada de los CDI Bean).

`nombre`: Es el atributo del CDI Bean al que queremos acceder. En este caso, estamos accediendo al atributo `nombre` del objeto `jugadorBean`.

La EL se evalúa en tiempo de ejecución y puede ser utilizada en diferentes contextos, como en las páginas XHTML, en los atributos de los componentes JSF y en los controladores de respaldo (como el CDI Bean en nuestro caso). Permite vincular de manera dinámica los datos y las acciones de la interfaz de usuario con el código subyacente.

La EL también proporciona características adicionales, como el acceso a colecciones, la invocación de métodos y operaciones aritméticas y lógicas básicas. Esto permite realizar cálculos y manipulaciones de datos directamente en las páginas XHTML o en las expresiones del CDI Bean sin la necesidad de escribir código Java adicional.

En resumen, la Expresión de Lenguaje (EL) es un lenguaje de scripting utilizado en tecnologías web basadas en Java, como JSF, para acceder y manipular datos en tiempo de ejecución. Permite una integración fácil y poderosa entre la interfaz de usuario y el código subyacente, facilitando la vinculación dinámica de datos y acciones.

En este ejemplo, utilizamos `<h:outputLabel>` para mostrar etiquetas de texto, `<h:inputText>` para crear campos de entrada de texto y `<h:selectBooleanCheckbox>` para crear un checkbox. Además, hemos utilizado `required="true"` para hacer que todos los campos sean obligatorios. El botón "Guardar" ejecutará el método `guardar()` en el CDI Bean `jugadorBean`. `<h:messages>` se utiliza para mostrar mensajes de validación en rojo.

Paso 3: Configurar el archivo faces-config.xml para la navegación

- Haz clic derecho en la carpeta "**Web pages/WEB-INFO**" en el proyecto y selecciona "**New**", selecciona la opción "**Other**". En la ventana que se abre busca la carpeta "**JavaServer Faces**" y selecciona la opción "**JSF Faces Configuration**".
- El nombre por defecto es "**faces-config.xml**", haz clic en "**Finalizar**".
- Abre el archivo "**faces-config.xml**" y agrega la siguiente configuración para establecer las rutas de navegación:

```
xml Copy code

<?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://
  version="2.2">
  <navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/jugadores.xhtml</to-view-id>
      <redirect />
    </navigation-case>
  </navigation-rule>
</faces-config>
```

En este ejemplo, hemos configurado una regla de navegación que indica que cuando el resultado de la acción **"guardar"** en el formulario sea **"success"**, se redirigirá a la página **"jugadores.xhtml"**.
Desglose de etiquetas:

<navigation-rule> Bloque donde se establece la regla de navegación.

<from-view-id> Origen. Desde donde afecta la regla de navegación

<navigation-case> Bloque donde especificamos el destino y el alias

<from-outcome> Alias. Es el nombre que pasaremos desde un método de Java en formato String para redirigir a la página de destino.

<to-view-id> Destino. La página en la que finalizaremos cuando se cumpla la regla de navegación.

Paso 4: Crear la página XHTML para mostrar los jugadores por equipo

- Crea un nuevo archivo XHTML llamado "jugadores.xhtml".
- Agrega las declaraciones de espacio de nombres y el código siguiente:

```
html Copy code

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">

  <h:body>
    <h1>Jugadores por Equipo</h1>

    <h:dataTable value="#{jugadorBean.jugadoresPorEquipo}" var="jugador"
      <h:column>
        <f:facet name="header">Nombre</f:facet>
        #{jugador.nombre}
      </h:column>
      <h:column>
        <f:facet name="header">Posición</f:facet>
        #{jugador.posicion}
      </h:column>
      <h:column>
        <f:facet name="header">Equipo</f:facet>
        #{jugador.equipo}
      </h:column>
      <h:column>
        <f:facet name="header">Edad</f:facet>
        #{jugador.edad}
      </h:column>
      <h:column>
        <f:facet name="header">Activo</f:facet>
        #{jugador.activo ? 'Sí' : 'No'}
      </h:column>
    </h:dataTable>
  </h:body>
</html>
```

En este ejemplo, utilizamos **<h:dataTable>** para mostrar los jugadores del equipo seleccionado en forma de tabla. Cada columna representa un atributo del jugador.

value="#{jugadorBean.jugadoresPorEquipo}" hace referencia a una lista de jugadores filtrados por equipo en el CDI Bean jugadorBean. *Nota: La lista debe tener jugadores añadidos previamente para que se muestre algo en la tabla*

Paso 5: Actualizar el CDI Bean para manejar la navegación y la lógica de validación

- Actualiza el CDI Bean jugadorBean con el siguiente código:

```
import javax.inject.Named;

import javax.enterprise.context.RequestScoped;

import java.io.IOException;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

import javax.faces.application.FacesMessage;

import javax.faces.context.ExternalContext;

import javax.faces.context.FacesContext;

@Named("jugadorBean")

@RequestScoped

public class JugadorBean implements Serializable {

    private String nombre;

    private String posicion;

    private String equipo;

    private int edad;

    private boolean activo;

    private List<Jugador> jugadores;

    public void guardar() throws IOException {

        if (nombre == null || nombre.isEmpty() || posicion == null || posicion.isEmpty()

            || equipo == null || equipo.isEmpty() || edad <= 0) {

            FacesContext context = FacesContext.getCurrentInstance();

            context.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR,

                "Error", "Completa todos los campos"));

            ExternalContext ec = context.getExternalContext();

            ec.redirect(ec.getRequestContextPath() + "/index.xhtml");
```

```

    } else {

        // Lógica para guardar el jugador en la base de datos o realizar otras acciones

        // Añade el jugador a la lista de jugadores

        if (jugadores == null) {

            jugadores = new ArrayList<>();

        }

        jugadores.add(new Jugador(nombre, posicion, equipo, edad, activo));

        ExternalContext ec = FacesContext.getCurrentInstance().getExternalContext();

        ec.redirect(ec.getRequestContextPath() + "/jugadores.xhtml");

    }

}

// Getters y setters para los atributos

public List<Jugador> getJugadoresPorEquipo() {

    List<Jugador> jugadoresPorEquipo = new ArrayList<>();

    if (jugadores != null) {

        for (Jugador jugador : jugadores) {

            if (jugador.getEquipo().equals(equipo)) {

                jugadoresPorEquipo.add(jugador);

            }

        }

    }

    return jugadoresPorEquipo;

}

}

```

En este ejemplo, hemos agregado lógica adicional al método `guardar()`. Si algún campo no está completo, se muestra un mensaje de error y se redirige a la página inicial utilizando **ExternalContext**. Si todos los campos están completos, el jugador se guarda en la lista de jugadores y se redirige a la página "**jugadores.xhtml**". El método **getJugadoresPorEquipo()** filtra la lista de jugadores por equipo para su visualización en la página "**jugadores.xhtml**". Una alternativa más sencilla para navegar en lugar de usar `ExternalContext` es hacer que el método

devuelva String devolviendo el alias hacía donde queremos ir; string vacío ("") para refrescar el index o el alias ("succes" en este caso) para ir a "jugadores.xhtml". El método guardar() con esta alternativa quedaría así:

```
public String guardar() throws IOException {  
    if (nombre == null || nombre.isEmpty() || posicion == null || posicion.isEmpty()  
        || equipo == null || equipo.isEmpty() || edad <= 0) {  
        FacesContext context = FacesContext.getCurrentInstance();  
        context.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR,  
            "Error", "Completa todos los campos"));  
        return "";  
    } else {  
        // Lógica para guardar el jugador en la base de datos o realizar otras acciones  
        // Añade el jugador a la lista de jugadores  
        if (jugadores == null) {  
            jugadores = new ArrayList<>();  
        }  
        jugadores.add(new Jugador(nombre, posicion, equipo, edad, activo));  
        return "succes";  
    }  
}
```

TEORÍA SOBRE LOS CDI BEAN

Un CDI Bean (Contexts and Dependency Injection Bean) es un componente de Java que se utiliza en el contexto de la tecnología de inyección de dependencias (CDI) en Java EE (Enterprise Edition). Los CDI Beans son una parte integral del enfoque de desarrollo basado en componentes de Java EE.

En términos sencillos, un CDI Bean es una clase Java gestionada por el contenedor CDI que puede ser inyectada en otras clases y utilizada para mantener y compartir datos y lógica de negocio en una aplicación. Los CDI Beans ofrecen un enfoque de programación orientada a objetos para la construcción de componentes reutilizables y la gestión de la dependencia entre ellos.

Aquí hay algunos conceptos clave relacionados con los CDI Beans:

- Ciclo de vida del CDI Bean: Un CDI Bean puede tener diferentes ciclos de vida, como `@RequestScoped`, `@SessionScoped`, `@ApplicationScoped`, `@ConversationScoped`, entre otros. Estos ámbitos definen la duración y el alcance del CDI Bean en el contexto de la aplicación.
- Inyección de dependencias: Los CDI Beans pueden ser inyectados en otras clases utilizando la anotación `@Inject`. Esto permite que las dependencias requeridas por una clase se resuelvan automáticamente por el contenedor CDI, evitando la necesidad de crear instancias manualmente.
- Productores de CDI Bean: Un CDI Bean también puede actuar como un productor de otros CDI Beans. Esto permite crear instancias de CDI Beans personalizados con configuraciones específicas o realizar operaciones adicionales antes de inyectarlos en otras clases.
- Calidad de dependencia: Los CDI Beans también pueden utilizar la anotación `@Qualifier` para diferenciar diferentes implementaciones de una misma interfaz o clase. Esto es útil cuando se tiene más de una implementación disponible y se necesita especificar cuál debe ser inyectada en una determinada clase.

En resumen, un CDI Bean en Java EE es una clase gestionada por el contenedor CDI que puede ser inyectada en otras clases para compartir datos y lógica de negocio. Proporciona un mecanismo eficiente para la gestión de dependencias y la creación de componentes reutilizables en una aplicación Java EE.