



CONVERTIDOR DE MONEDA.

Implementado con Servlet ,JSP y Gson.

GUIA DE DESARROLLO E IMPLEMENTACIÓN

En el siguiente informe se mostrara el paso a pasa y la estructura para la creación de una aplicación web, está fue desarrollada con una conexión a una base de datos, con dependencias y servlet.

Jenniffer Helena Alvarez Puello
Juan Camilo Tejada Porto

Universidad de Cartagena
Facultad de ingeniería
Programa de Ingeniería de Sistemas
Cartagena 2017

Tabla de contenido

Servlet Convertidor jsp	2
CONVERTIDOR SERVLET JSP	3
1. MODELO.	3
1.1 base de datos	3
1.2 Lógica.....	5
2. CONTROLADOR.....	6
3. VISTAS.	8
3.1. Vista Index.html	8
3.2. Vista cambio.jsp.....	9
4. DEPENDENCIAS.	12

Servlet Convertidor jsp

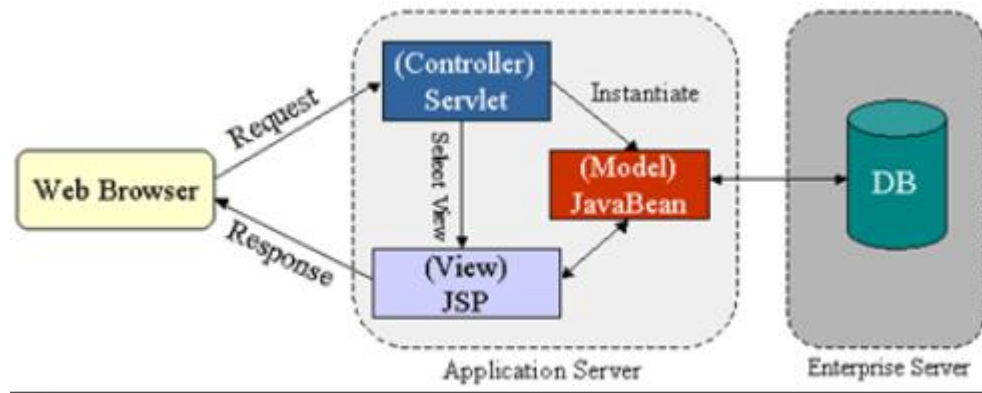


Ilustración 1

Para la elaboración del Convertidor servlet, debemos contextualizar, En la *ilustración 1* podemos observar la estructura básica del proyecto en el cual se trabaja con el patrón arquitectónico MVC, donde desarrollaremos una las vistas del cliente en JSP, Los modelos en código JAVA, la base de datos en MySQL y finalmente el controlador será desarrollado por ser servlet, de igual forma el proyecto se desarrolló con la librería Gson.

Un Servlet es un objeto java que pertenece a una clase que extiende de `javax.servlet.http.HttpServlet`, son pequeños programas escritos en Java que admiten peticiones a través del protocolo HTTP. Los servlets reciben peticiones desde un navegador web, las procesan y devuelven una respuesta al navegador, normalmente en HTML.

Gson es una librería open-source la cual nos permite convertir nuestros objetos Java en JSON o viceversa. JSON para los que no sepan es un formato para intercambiar información (al igual que XML) pero se basa en una estructura de pares clave-valor, este formato se ha popularizado debido a que es más ligero que el XML y es de fácil lectura a la vista de las personas.

Para el desarrollo de la aplicación se utilizó la herramienta **MAVEN**, la cual cuando definimos las dependencias de Maven, este sistema se encargará de ubicar las librerías que deseamos utilizar en Maven Central, el cual es un repositorio que contiene cientos de librerías constantemente actualizadas por sus creadores.

CONVERTIDOR SERVLET JSP

Lo primero es crear el código, la base de datos y el diseño de las vistas para el cliente, se crea un proyecto **Java Application** con la siguiente estructura:

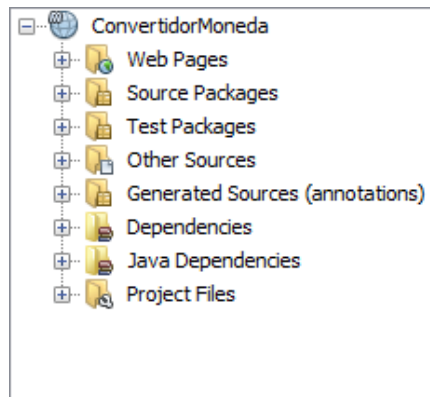


Ilustración 2

El recorrido del proyecto lo daremos empezando desde la base de datos, hasta llegar a la interacción del cliente con la aplicación.

Web Pages: se colocaran los archivos HTML, JSP, JavaScript, CSS, Imágenes, iconos y Multimedia

META-INF: contiene información XML sobre la estructura del proyecto (no debemos tocarla)

WEB-INF: contiene un archivo XML en el cual se registran y describen los Servlet y otros detalles importantes del sitio web, como es el caso de la página de inicio, o el HOME o Welcom

Source Package: Contendrá los archivos de código fuente .java, para el caso de los Servlet y otras clases, como por ejemplo los Java Bean, la clase conexión a bd y las clases utilitarias.

Librerías: contiene las librerías o bibliotecas de clases.jar que necesite el proyecto para funcionar, como es el caso de la biblioteca de clases que actúan como Driver o Manejador de conexiones a MySQL.

1. MODELO.

1.1 base de datos

Se crea una base de datos en MySQL, la cual llamamos “convertidor”, y dentro de esta tenemos una tabla llamada moneda, a la cual realizamos la conexión con el modelo: Nos situamos en moneda.java ubicada en el paquete modelo.

Primero importamos todas las librerías, que nos brinda **Javax.persistence**, las cuales nos permitirán hacer las conexiones, y de igual forma se empiezan a llenar los campos correspondientes a la tabla con la cual se procederá hacer la conexión

```
package modelo;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;

@Entity
@Table(name = "monedas", catalog = "convertidor", schema = "")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Moneda.findAll", query = "SELECT m FROM Moneda m")
    , @NamedQuery(name = "Moneda.findByNombre", query = "SELECT m FROM Moneda m WHERE m.nombre = :nombre")
    , @NamedQuery(name = "Moneda.findByValorDolares", query = "SELECT m FROM Moneda m WHERE m.valorDolares = :valorDolares")
})
```




Ilustración 3

Y definimos de igual forma las los query/Consultas para realizar las búsquedas. En este caso hemos definido una clase moneda y unos 1. NamedQuery denominado “moneda.findAll” que nos busca a todas las monedas, 2. NamedQuery denominado “moneda.findByNombre” que nos busca a las monedas por el nombre, 3. NamedQuery denominado “moneda.findByValorDolares” que nos busca a todas las monedas por el valor del dolar.

```
public class Moneda implements Serializable {

    private static final long serialVersionUID = 1L;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 3)
    @Column(name = "nombre", nullable = false, length = 3)
    private String nombre;
    @Basic(optional = false)
    @NotNull
    @Column(name = "valor_dolares", nullable = false)
    private float valorDolares;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id", nullable = false)
    private Integer id;

    public Moneda() {
    }

    public Moneda(Integer id) {
        this.id = id;
    }
}
```

```

public Moneda(Integer id, String nombre, float valorDolares) {
    this.id = id;
    this.nombre = nombre;
    this.valorDolares = valorDolares;
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public float getValorDolares() {
    return valorDolares;
}

public void setValorDolares(float valorDolares) {
    this.valorDolares = valorDolares;
}

public Integer getId() {
    return id;
}

```



```

@Override
public int hashCode() {
    int hash = 0;
    hash += (id != null ? id.hashCode() : 0);
    return hash;
}

@Override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the id fields are not set
    if (!(object instanceof Moneda)) {
        return false;
    }
    Moneda other = (Moneda) object;
    if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
        return false;
    }
    return true;
}

@Override
public String toString() {
    return "modelo.Moneda[ id=" + id + " ]";
}
}

```

1.2 Lógica

Entre el mismo paquete del modelo encontramos la lógica, encargada de hacer el funcionamiento del cambio de valor de la moneda.

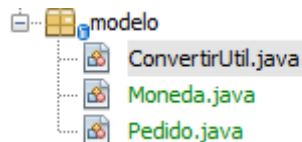


Ilustración 4

```

import java.util.List;

public class ConvertirUtil {

    private List<Moneda> monedas;

    public String convertir(String actual, String objetivo, String valor){
        if(actual.equals("USD")){
            return FromUSD(valor,objetivo);
        }else if(!actual.equals("USD") && !objetivo.equals("USD")){
            return toUSD(valor,actual);
        }else if(!actual.equals("USD") && !objetivo.equals("USD")){
            return FromTo(valor, objetivo, actual);
        }else{
            return valor;
        }
    }

    private String FromUSD(String valor, String objetivo) {
        float actual = Float.parseFloat(valor);
        float result=actual;
        for(int i =0; i<monedas.size();i++){
            if (monedas.get(i).getNombre().equals(objetivo)){
                result=monedas.get(i).getValorDolares()*actual;
            }
        }
        return Float.toString(result);
    }

    private String FromTo(String valor, String objetivo, String actual) {
        valor=toUSD(valor,actual);
        return FromUSD(valor,objetivo);
    }

    public List<Moneda> getMonedas() {
        return monedas;
    }

    public void setMonedas(List<Moneda> monedas) {
        this.monedas = monedas;
    }
}

```

En esta clase nos encargamos de realizar una lista de tipo moneda, que fue la clase que se explicó anteriormente en el cual en la instancia por medio del constructor se le pasan los valores correspondientes.

2. CONTROLADOR.

Por medio del controlador nos encargamos de realizar las instancias del modelo, seleccionar las vistas del JSP y de recibir las peticiones del cliente.

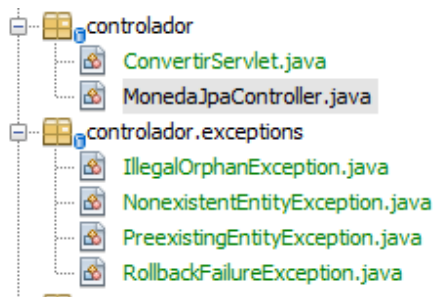


Ilustración 5

```

public class ConvertirServlet extends HttpServlet {

    @PersistenceContext(unitName = "com.seminario_ConvertidorMoneda_war_1.0-SNAPSHOTPU")
    private EntityManager em;
    @Resource
    private javax.transaction.UserTransaction utx;
    private EntityManagerFactory emf = Persistence.createEntityManagerFactory("com.seminario_ConvertidorMoneda_war_1.0");
    private MonedaJpaController js=new MonedaJpaController(utx, emf);
    private ConvertirUtil forex = new ConvertirUtil();
    Gson gson = new Gson();
}

```

Primero definimos todas las variables y de igual forma tenemos nuestro objeto Gson.

El seguimiento de sesión es un mecanismo que los servlets utilizan para mantener el estado sobre la serie de peticiones desde un mismo usuario (esto es, peticiones originadas desde el mismo navegador) durante un periodo de tiempo.

Las sesiones son compartidas por los servlets a los que accede el cliente. Esto es conveniente para aplicaciones compuestas por varios servlets.

```

ring ac=request.getParameter("accion");
ring pc=request.getParameter("pc");
System.out.println(pc);
System.out.println(ac);
(ac!=null){
    if(ac.equals("convertir")){
        forex.setMonedas(js.findMonedaEntities());
        String result = forex.convertir(request.getParameter("actual"), request.getParameter("objetivo"), request.getParameter("valor"));
        request.getSession().setAttribute("monedas.result", result);
        response.sendRedirect("cambio.jsp");
    }
    if(ac.equals("inicio") && ac!=null){
        request.getSession().setAttribute("monedas.lista", js.findMonedaEntities());
        response.sendRedirect("cambio.jsp");
    }
    if(ac.equals("limpiar") && ac!=null){
        request.getSession().setAttribute("monedas.result", "");
        response.sendRedirect("cambio.jsp");
    }
}

```

La parte en la cual respondemos a las peticiones solicitadas por la aplicación de escritorio es la siguiente, en ella recibimos y deserializamos el objeto pedido para convertir el valor solicitado a una moneda objetivo y responder con un String a cliente que contiene el resultado de la operación de conversión.

```

if(pc!=null){
    forex.setMonedas(js.findMonedaEntities());
    String json=request.getParameter("pc");
    System.out.println(json);
    Pedido nuevo= gson.fromJson(json, Pedido.class);
    String result = gson.toJson(forex.convertir(nuevo.getActual(), nuevo.getObjetivo(), nuevo.getValor()));

    System.out.println(result);
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    PrintWriter salida = response.getWriter();
    salida.print(result);
    salida.close();
}

```

Y la parte donde se utiliza el Gson que nos provee un mecanismo de realizar esto de forma sencilla y que no implique muchos cambios a nivel de nuestro código, para poder llevar esto a cabo deberemos crear un clase que se encargue ya sea de la

serialización o deserialización de un objeto en particular (se recomienda crear un solo objeto que haga ambas operaciones).

3. VISTAS.

Las vistas están desarrolladas en JSP, estas con las encargadas de la interacción con el cliente directamente y de mostrar los resultados requeridos.

Con JSP podemos crear aplicaciones web que se ejecuten en variados servidores web, de múltiples plataformas, ya que Java es en esencia un lenguaje multiplataforma. Las páginas JSP están compuestas de código HTML/XML mezclado con etiquetas especiales para programar scripts de servidor en sintaxis Java. Por tanto, las JSP podremos escribirlas con nuestro editor HTML/XML habitual.

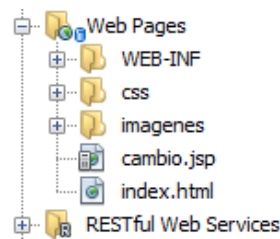


Ilustración 6

Al situarnos en la carpeta de **Web Pages**, podemos ver toda la parte web del proyecto, como ya se dijo que el desarrollo de estos se realizaron en JSP, pero en index se realizó con HTML:

3.1. Vista Index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="css/stylo.css">
    <title>Convertidor</title>
  </head>
  <body>
    <header>
      <div class="ancho">
        <div class="logo">
          <p><a href="index.html">Convertidor</a></p>
        </div>
      </div>
    </header>
    <div id="envoltura">
      <div id="contenedor">
        <div id="cabecera">
        </div>
        <div id="cuerpo">
          <form id="form-login" action="/ConvertidorMoneda/moneda?accion=inicio" method="post" autocomplete="c">
            <div id="resultado">
              <p><label>BIENVENIDO </label></p>
              <p><input type="submit" id="submit" name="limpiar" value="Empezar" class="boton"></p>
            </div>
          </form>
        </div>
      </div>
    </div>
  </body>
</html>
```

Ilustración 7

En index.HTML encontramos la página principal, desarrollado en código HTML, y CSS en la cual estará un botón nos dirigimos al JSP llamado cambio, que será el encargado de mostrar la vista para que el usuario interactúe y realice el cambio pedido.

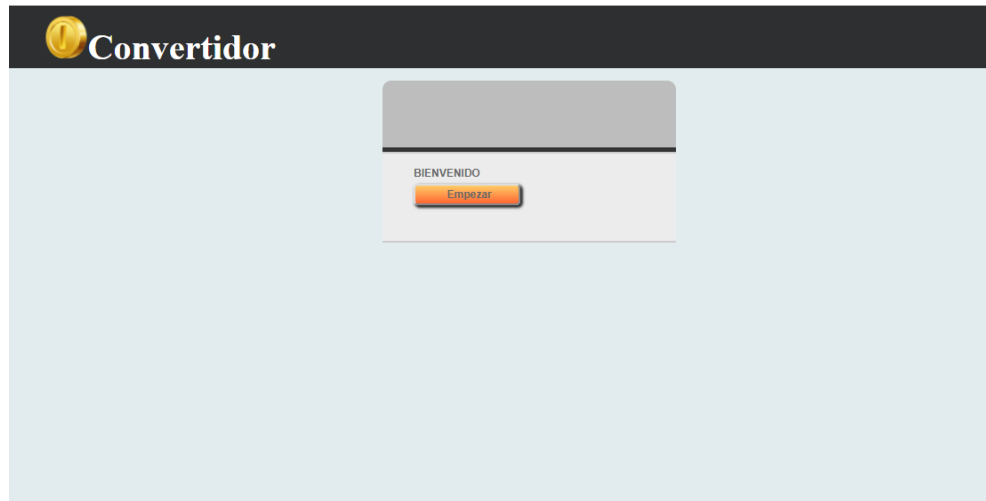


Ilustración 8

3.2. Vista cambio.jsp

```
<%@page import="java.util.List"%>
<%@page import="modelo.Moneda"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%
    String result="";
    if(request.getSession().getAttribute("monedas.result")!=null){
        result= (String) request.getSession().getAttribute("monedas.result");
    }
    if(session.getAttribute("monedas.lista")==null){
        getServletContext().getRequestDispatcher("/index.html").forward(request, response);
    }
    List<Moneda> monedas = (List<Moneda>)session.getAttribute("monedas.lista");
%>
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<link rel="stylesheet" href="css/stylo.css">
<title>Convertidor</title>
</head>
<body>
<header>
    <div class="ancho">
        <div class="logo">
            <p><a href="index.html">Convertidor</a></p>
        </div>
    </div>
```

```

</header>
<div id="envoltura">
  <div id="contenedor">
    <div id="cabecera">
    </div>
    <div id="cuerpo">
      <form id="form-login" action="/ConvertidorMoneda/moneda?accion=convertir" method="post" autocomplete="off">
        <p><label>VALOR:</label></p>
        <p><input name="valor" type="text" id="usuario" placeholder="Ingresa valor" autofocus="" required=""></p>
        <div>
          <p>
            <label for="Moneda">Convertir de </label>
            <select name="actual" class="moneda">
              <%
                for(int e=0;e<monedas.size();e++){
              <%
                <option value="<%=monedas.get(e).getNombre() %"><%=monedas.get(e).getNombre() %"></option>
              <%
                }
              <%
            </select>
          </p>
        </div>
      </form>
    </div>
    <div>
      <p>
        <label for="Moneda">Convertir a: </label>
        <select name="objetivo" class="moneda">
          <%
            for(int e=0;e<monedas.size();e++){
          <%
            <option value="<%=monedas.get(e).getNombre() %"><%=monedas.get(e).getNombre() %"></option>
          <%
            }
          <%
        </select>
      </p>
    </div>
    <div id="bot">
      <input type="submit" id="submit" name="submit" value="Convertir" class="boton"></div>
      <form action="/ConvertidorMoneda/moneda?accion=limpiar" method="post">
        <div id="bot">
          <input type="submit" id="submit" name="limpiar" value="Limpiar" class="boton"></div>
        </form>
      <div id="resultado">
        <label for="Moneda">RESULTADO </label>
        <input type="text" name="nombredelacaja" disabled value="<%=result %"></div>
    </div>
  </div>
</div>

```

En cambio.jsp, se ve la interacción de la vista con el controlador, y de igual forma la respuesta que del controlador es manda; a continuación una explicación de las linead de código:

- Primero realizamos las importaciones por medio de las cuales será validado y mostrados las integraciones, todo en conjunto con el controlador.

```

<%@page import="java.util.List"%>
<%@page import="modelo.Moneda"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>

```

Ilustración 9

- Luego con código java hacemos utilización del servlet, para respectiva conexión con la tabla a implementar, se valida que haya una sesión iniciada, con la base de datos, luego con la tabla y esos datos extraídos son archivado en un arrayLits.

```

<%
String result="";
if(request.getSession().getAttribute("monedas.result")!=null ){
    result= (String) request.getSession().getAttribute("monedas.result");
}
if(session.getAttribute("monedas.lista")==null){
    getServletContext().getRequestDispatcher("/index.html").forward(request, response);
}
List<Moneda> monedas = (List<Moneda>) session.getAttribute("monedas.lista");
%>

```

Ilustración 10

4. Las partes seleccionadas con → en la imagen, hacen referencias a código java, encargado de hacer validaciones.
4. En la **ilustración 10** vemos el resultado final de la página cambio.jsp.

Ilustración 11

Debemos tener que presente que todas estas vistas están diseñadas de la mano con **CSS**, el lenguaje utilizado para describir la presentación de documentos HTML o XML, esto incluye varios lenguajes basados en XML como son XHTML o SVG. CSS describe como debe ser renderizado el elemento.

4. DEPENDENCIAS.

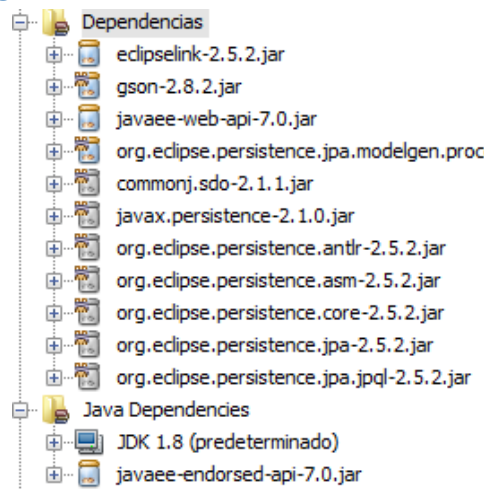


Ilustración 12

En el código de una aplicación con OOP (Programación Orientada a Objetos) tenemos una posible separación del código en dos partes, una en la que creamos los objetos y otra en la que los usamos. Existen patrones como las factorías que tratan esa parte, pero la inyección de dependencias va un poco más allá. Lo que dice es que los objetos nunca deben construir aquellos otros objetos que necesitan para funcionar. Esa parte de creación de los objetos se debe hacer en otro lugar diferente a la inicialización de un objeto. Estas son usadas en el controlador, con la parre del Gson es una librería open-source la cual nos permite convertir nuestros objetos Java en JSON o viceversa. JSON para los que no sepan es un formato para intercambiar información (al igual que XML) pero se basa en una estructura de pares clave-valor, este formato se ha popularizado debido a que es más ligero que el XML y es de fácil lectura a la vista de las personas.

Al igual que la mayoría de las librerías existentes tanto a nivel de generación de XML como de JSON, GSON nos provee la posibilidad de cambiar el nombre a nuestros atributos para cuando se genere el documento o cuando a partir de un documento se quieran cargar los objetos Java.

Para el desarrollo de esta aplicación se utilizó la herramienta **MAVEN**, la cual se utiliza en la gestión y construcción de software. Posee la capacidad de realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. Es decir, hace posible la creación de software con dependencias incluidas dentro de la estructura del JAR. Es necesario definir todas las dependencias del proyecto (librerías externas utilizadas) en un fichero propio de todo proyecto Maven, el POM (Project Object Model). Este es un archivo en formato XML que contiene todo lo necesario para que a la hora de generar el fichero ejecutable de nuestra aplicación este contenga todo lo que necesita para su ejecución en su interior.

Bibliografía

blog.teraswap. (s.f.). Obtenido de <http://www.blog.teraswap.com/gson-introduccion/>

losteatinos. (31 de 9 de 2017). Obtenido de <http://www.losteatinos.es/servlets/servlet.html>