Universidad Nacional Autónoma de Nicaragua.

UNAN-LEÓN.

Tesis para Optar al Título de Ingeniero en Sistemas de Información

Prácticas de laboratorio para el desarrollo de aplicaciones web con Java utilizando Zkoss para la interfaz de usuario e Hibernate para el acceso a base de datos.

Elaborado por:

Br. Deivy Onelli Hebberth Tower.

Br. Nitzard Mardiell Luna Zavala.

Br. Allan José Medina Gutiérrez.

Tutor:

Msc. Denis Leopoldo Espinoza Hernández.

León, Septiembre del 2014.

A Dios que me ha guiado en todo momento y me dio la fuerza y sabiduría para salir siempre adelante permitiendo así finalizar otra meta más con éxito.

A mis padres:

Mi mamá Piedad de Carmen Gutiérrez Carrión, por haberme apoyado siempre, por sus consejos, sus valores y la motivación que me ha permitido ser una persona de bien. Gracias Mamá.

Mi papá Simón Benito Medina Hernández, que siempre ha estado presente cuando lo he necesitado y sé que está muy orgulloso de mí por este gran logro que he cumplido. Gracias Papá.

A mi tutor Denis Espinoza, gracias por compartir conmigo sus conocimientos, por sus consejos, y por haber confiado en nosotros.

A mis compañeros, Nitzard M. Luna y Deivy Onelli Hebberth, por darme la oportunidad de trabajar con ellos en la realización de esta tesis, solo les puedo decir Gracias.

A María Ángeles Meza Dávila, por todo el apoyo que me brindo durante todo el tiempo que estuvimos juntos. Gracias.

A todos los que de una u otra forma me han acompañado a lo largo de estos dos años, y colaboraron en el desarrollo de este proyecto.

Br. Allan José Medina Gutiérrez.

A Dios por darnos la sabiduría, entendimiento para dar un paso más en nuestra vida y poder de esta manera alcanzar el éxito.

A mi madre Juana Tower por estar en cada momento y apoyarme en mi educación y ser la parte fundamental en mi vida, a mis abuelitos que han sido como unos padres para mi desde que empecé a formarme como profesional y a mi padre que me apoyo en estos últimos años, a mi hermano mayor que me ha brindado su apoyo en los momentos más importantes de mi vida y a mis demás familiares que de una manera u otra me han apoyado.

Br. Deivy Onelli Hebberth Tower.

Agradezco a Dios, que es el motor fundamental, que nos llena de salud, sabiduría y de vida, en el caminar de nuestra vida.

A mi familia, por ser ellos los que me impulsan en mi vida día a día, a salir adelante, con todo su apoyo sin condiciones. Gracias a ellos estoy culminando una meta más.

A mi compañero de tesis, Allan Medina y Deivy Hebberth por estar ahí hombro a hombro a la realización de este proyecto.

A Denis Espinoza, aparte de ser nuestro tutor, lo considero un amigo que nos acompañó en este periodo de nuestras vidas, dando de su tiempo, apoyo y consejos para que alcanzáramos nuestro proyecto.

Br. Nitzard M. Luna.

Contenido

Capítulo I: Aspectos Introductorios

1.1 Introducción.

1.2 Antecedentes.

1.3 Planteamiento del Problema.

1.4 Justificación

1.5 Objetivos.

1.1 Introducción

La Web 2.0 es la que representa la evolución de las aplicaciones tradicionales hacia aplicaciones web enfocadas al usuario final. El Web 2.0 es una actitud y no precisamente una tecnología. Se trata de aplicaciones que generen colaboración y dan servicios que reemplacen a las aplicaciones de escritorio.

Las Rich Internet Applications (RIA), son aplicaciones web que tienen la mayoría de las características de las aplicaciones de escritorio tradicionales. Estas aplicaciones utilizan un navegador web estandarizado para ejecutarse y por medio de complementos o una máquina virtual se agregan las características adicionales. Las RIA surgen como una combinación de las ventajas que ofrecen las aplicaciones web y las aplicaciones tradicionales.

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems y publicado en el año 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles, juegos, contenido basado en web y software de empresa.

Dentro del mundo java se encuentra varias plataformas como lo es Java Enterprises Edition la cual es una plataforma para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. Permite utilizar arquitecturas de N capas distribuidas y se apoya ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones y La plataforma Java EE está definida por una especificación. Java EE tiene varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc. y define cómo coordinarlos.

A continuación se muestra una pequeña lista en la cual se puede apreciar el lugar en el que se encuentra el lenguaje Java para la programación Web. Esto fue tomado de un artículo publicado por Stephen O'grady el 22 de Enero de 2014 en redmonk.com (Primer y única industria enfocada al análisis de firmas de desarrollo).

1. JavaScript (+1)
2. Java (-1)
3. PHP
4. C# (+2)
5. Python (-1)

Además de la lista mostrada anteriormente, de acuerdo a la clasificación de IEEE Spectrum (emblemática revista y sitio web del IEEE, organización profesional más grande del mundo dedicada a la ingeniería y las ciencias aplicadas) tenemos otra lista de lenguajes de programación para la Web, publicada por Kanishk Kunal el 11 de Julio de 2014.

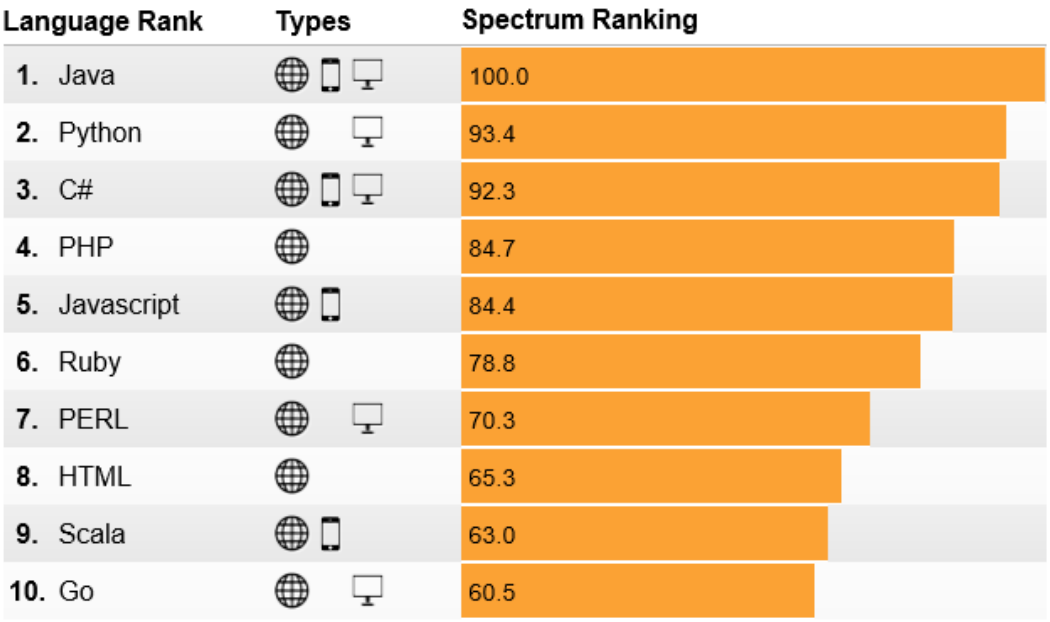


Figura 1: Clasificación de lenguajes de programación Web 2014. **Fuente:** http://superdevresources.com/top-programming-languages-2014/

Como se pudo notar Java esta entre los primeros lugares de las listas de clasificación. ZKoss (ZK) es un framework de interfaz de usuario que permite crear increíbles aplicaciones móviles y web sin tener que aprender JavaScript o AJAX. Ofrece un excelente rendimiento en el desarrollo de aplicaciones empresariales robustas, pero a su vez ágiles de desarrollar.

Además ofrece un entorno de desarrollo rápido, y una línea de aprendizaje prácticamente plana. ZK soporta todos los principales patrones de desarrollo, tales como MVC (Model View Controller) y

MVVM (Model View View Model) y junto con características propias del framework satisfice las necesidades antes mencionadas y puede ayudar a mejorar la cultura organizacional de desarrollo web.

ZK se creó gracias a una comunidad de desarrolladores (Potix Corporation) que se han propuesto que la implementación de interfaces de usuario (GUI) en Ajax sea mucho más fácil y cómoda de implementar y desarrollar, fue lanzado y presentado a la comunidad java en el 2006, es un framework de aplicaciones web en Ajax de código abierto que permite crear interfaces de usuario completas para aplicaciones web sin usar javaScript.

De todo lo anteriormente mencionado fue que se decidió hacer este trabajo, ya que este nuevo framework es muy sencillo de entender y no hay que tener mucho conocimiento sobre la programación web, de esa manera será muy útil para los estudiantes de la carrera de ingeniería en sistema y telemática, además el presente documento está hecho en base a pequeños proyectos guiados para que el estudiante pueda enriquecer su conocimiento con respecto a la programación web con Zkoss.

1.2 Antecedentes

En el departamento de computación se imparten diferentes asignaturas de programación web, entre estas, Diseño de páginas web donde se enseña a programar páginas sencillas con bloc notas y notepad ++**;** programación orientada a la web I, enseña al estudiante las tecnologías básicas con HTML, CSS, JAVASCRIPT y PHP. Y programación orientada a la web II, enseña sobre la tecnología ASP.NET en la cual podemos aprender el lenguaje C# para desarrollar páginas web dinámicas con la herramienta Visual Studio.

Anteriormente en el departamento no se han realizado investigaciones que aborden el estudio sistemático de esta tecnología, sin embargo, es importante citar trabajos que han fomentado el aprendizaje de tecnologías web como lo es:

El trabajo monográfico de la Msc. Valeria Medina el cual se basa en el Desarrollo de una Aplicación Web con ASP .NET 2.0, basada en un modelo de tres capas, dicha tesis se centra en la creación de un portal web que será implementado en el componente curricular Programación Orientada a la Web II el cual podrá servir como modelo de referencia para el diseño de cualquier aplicación dinámica de este tipo.

1.3 Planteamiento del problema

En el departamento de computación, el estudio de nuevos framework como Zkoss es necesario para que docentes y estudiantes se sientan familiarizados con las nuevas tecnologías. Sin embargo, la ausencia de un documento que de manera secuencial e incremental aborde los contenidos teóricos y prácticos, dificulta el aprendizaje de esta nueva tecnología.

Es importante destacar que en el Plan de Estudio 2011 de la carrera de Ingeniería en Sistemas el componente Programación Orientada a la Web II abordará el desarrollo de aplicaciones en Java y al ser un contenido nuevo, los docentes no cuentan con un documento base para impartir la clase y la asignación de prácticas de laboratorios.

De todo lo anterior nos surgen las siguientes interrogantes:

¿Cómo podría desarrollarse un plan de prácticas de laboratorio relacionadas con ZKOSS para el Departamento de Computación de la UNAN-León?

¿Qué secuencia de temas debe tener el documento de manera que permita a los docentes la exposición ordenada y gradual de los contenidos a los estudiantes?

¿Cuál es la metodología que debe seguirse en la elaboración de las prácticas de laboratorio para que el estudiante se sienta familiarizado al momento de su uso?

1.4 Justificación

Ya que el departamento de Computación de la UNAN-León desea incorporar la tecnología Java para el desarrollo de aplicaciones web, se vuelve una necesidad la creación de un documento que permita a los estudiantes abordar de manera eficiente, ordenada y secuencial el seguimiento y el aprendizaje de esta tecnología de una manera ágil, por tal motivo, se ha decidido emplear ZKOSS como framework a ser utilizado en este documento.

1.4.1 Originalidad

No existen trabajos y documentos que contienen prácticas a fines de esta nueva tecnología; por este motivo el presente trabajo ayudará de manera significativa a entender el desarrollo de aplicaciones web con Zkoss. Además será una opción que podrá ser utilizado en cualquier curso que se desee utilizar Java para el desarrollo web.

1.4.2 Alcance

Este documento será de gran apoyo tanto para docentes y como para estudiantes.

Para los docentes

- Tendrán un documento que les permita asignar prácticas de laboratorio guiadas a los estudiantes con una secuencialidad y un nivel de complejidad uniforme al avance de los cursos.

Para los estudiantes

- Tendrán la oportunidad de conocer esta nueva tecnología y de esta manera desarrollar sus propias aplicaciones web de forma ágil y profesional.

1.4.3 Producto

El producto a entregar, es un documento que tendrá prácticas de laboratorio. Este documento presentará las características siguientes:

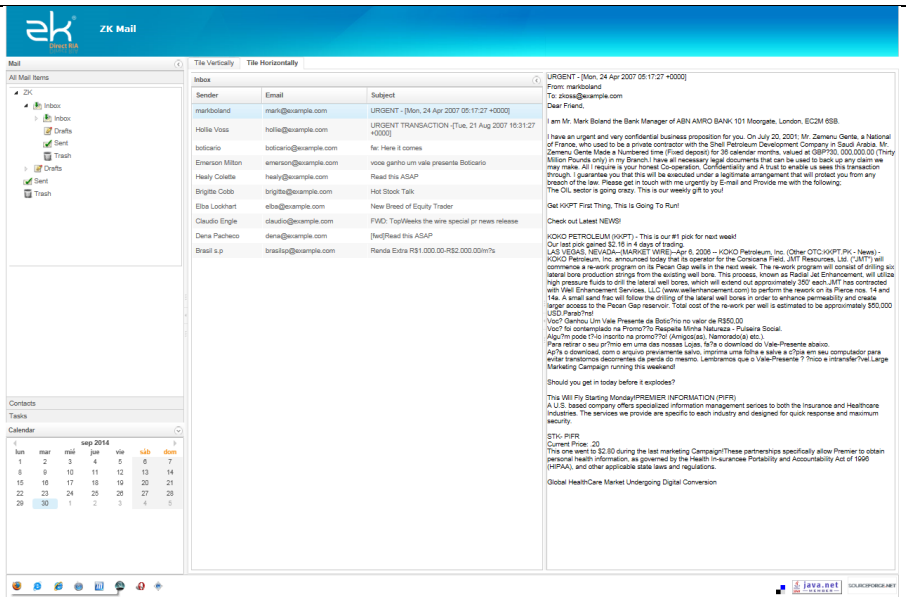
- **Guiado:** Describirá paso a paso la forma en la que se deberá realizar cada ejercicios.
- **Ilustrado:** Explicará de forma breve a través de imágenes la secuencia que se debe seguir para la creación de los ejercicios.

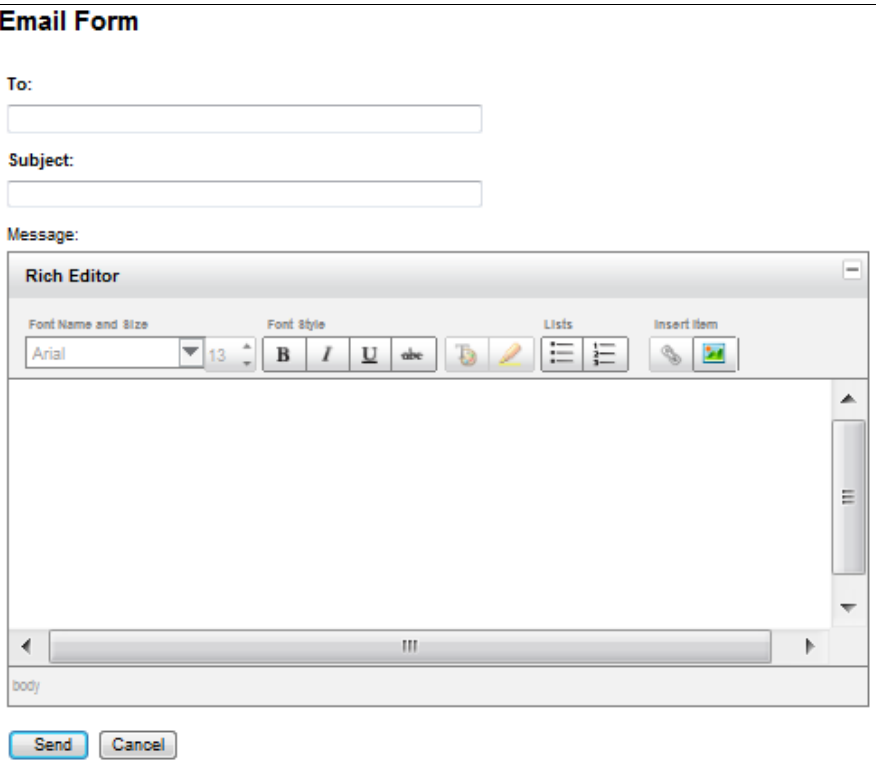
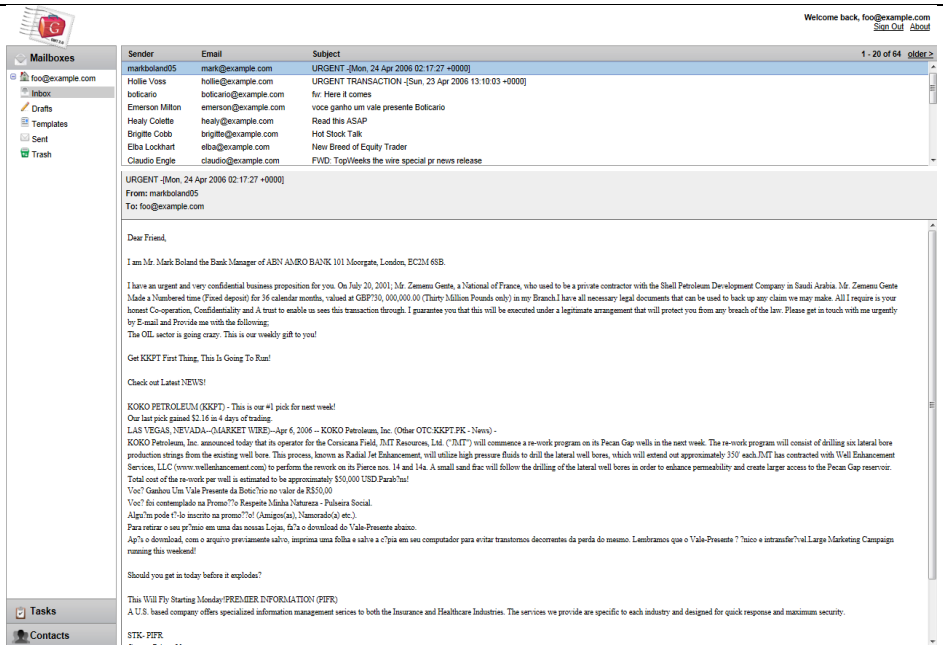
- **Familiar:** Los ejercicios y la documentación serán elaboradas en un lenguaje que los estudiantes puedan comprenderla fácilmente.

1.4.4 Framework de Graphical user interface (GUI) Java Enterprise Edition (EE)

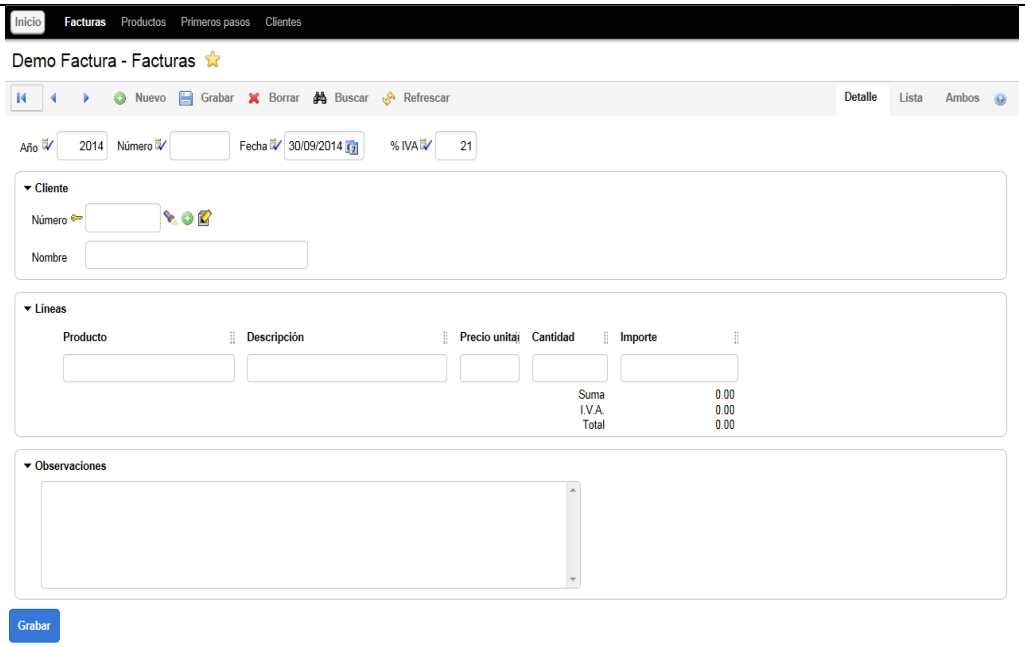
Una de las cosas que se busca en la parte visual es encontrar algún framework que tenga una buena interfaz visual y permita hacerlo de manera rápida y flexible, sin entrar en los detalles de HTML, CSS y Javascript, con un código legible y fácil de entender, para que el estudiante pudiese aplicar sus conocimientos de programación del entorno de escritorio a un entorno web.

Hacemos comparación del aspecto visual de algunas Framework Web Java:

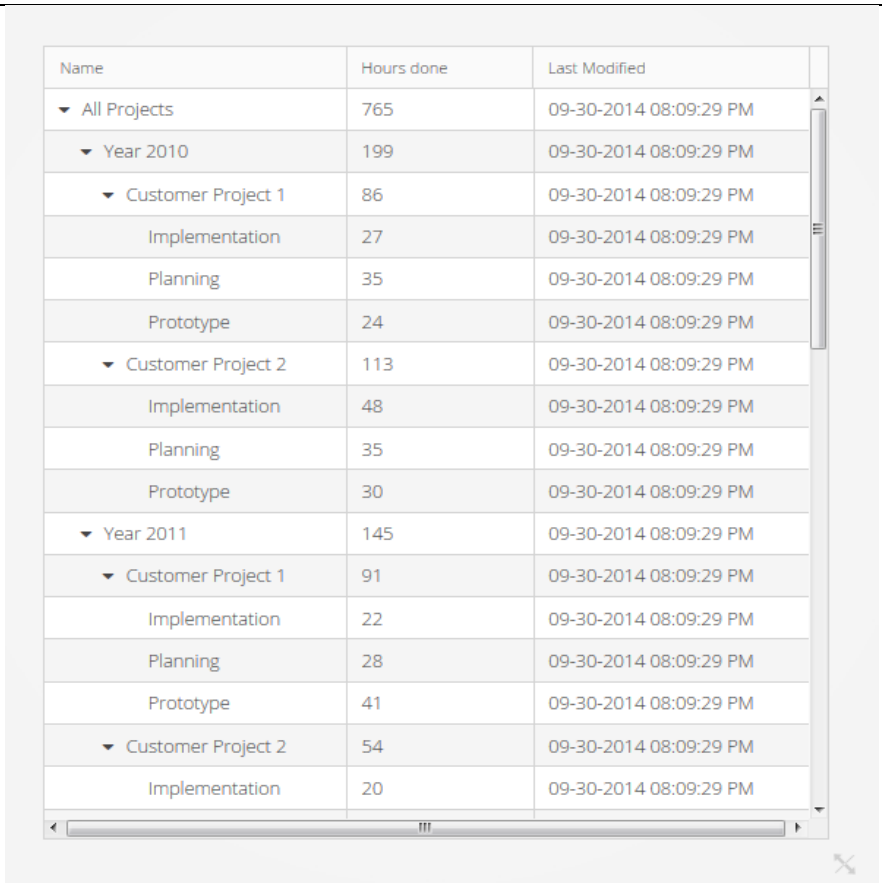
Zkoss	

Apache Click	
GWT (Google Web Toolkit)	

Openxava



Vaadin



Apache Wicket	

1.4.5 ¿Por qué Optar por Zkoss?

ZK es el framework Ajax líder en las empresas y de manera más fácil de construir grandes aplicaciones web modernas de Java.

ZK es de confianza y se estableció con una sólida trayectoria como el marco de elección en todas las industrias. Decenas de miles de desarrolladores usan ZK para construir sus sistemas de misión crítica incluyendo sistemas de varios millones de dólares que sirven a millones de usuarios y decenas de miles de sesiones simultáneas en el ámbito internacional.

Con más de 1.500.000 de descargas, ZK faculta a una amplia variedad de empresas e instituciones, que van desde pequeñas a grandes en múltiples industrias.

Estas son algunas de las empresas que adoptaron ZK para alimentar su infraestructura:

- Ebay
- Sony
- DreamWorks
- Toyota
- Samsung

A demás del soporte de los navegadores:

ZK 6,5 Y 7		
NAVEGADOR	**ZK 6.5**	**ZK 7**
Internet Explorer 6	✔	
Internet Explorer 7	✔	
Internet Explorer 8 +	✔	✔
Firefox	✔	✔
Cromo	✔	✔
Safari	✔	✔
iOS Safari [1]	✔	✔
Navegador de Android [1, 5]	✔	✔
Opera	✔	✔

1.5 Objetivos

1.5.1 Objetivo General

Elaborar prácticas de laboratorio para el desarrollo de aplicaciones web con Java utilizando la tecnología ZKOSS para la interfaz de usuario e Hibernate para el acceso a bases de datos.

1.5.2 Objetivos Específicos

Definir la secuencia de temas a ser abordado en el documento de manera que permita a los docentes la exposición ordenada y gradual de los contenidos a los estudiantes.

Desarrollar un conjunto de prácticas de laboratorio que sigan una misma metodología que permitan al estudiante una fácil y rápida apropiación de los conocimientos.

Capítulo II: Diseño Metodológico

2.1 Detalles de Prácticas y calendarización de prácticas.

2.2 Formato de Prácticas.

2.1 Detalles de Prácticas y calendarización de prácticas.

Temas	Duración Teoría	Prácticas de laboratorio.	Objetivo	Duración Prácticas
Introducción a ZKOSS	2 horas	**Instalación del entorno.**	Dar a conocer que es zkoss y que herramientas se necesitan para trabajar con este framework.	2 horas
Validación y Navegación	6 horas	**Controles y manejo de eventos.**	Familiarizarse con el framework realizando pequeños proyectos utilizando los controles básicos y eventos de una forma sencilla.	4 horas
MVC	4 horas	**Manipulando la entrada de usuario y emplear el patrón MVC**	Preparar a los estudiantes para que aprenda a desarrollar aplicaciones empleando el patrón MVC.	6 horas
Acceso a bases de datos con Hibernate	4 horas	**Accediendo a Base de datos con Hibernate**	En esta práctica se construye una aplicación para administrar una base de datos con 4 operaciones básicas, crear (Create), Leer (Read), actualizar (Update), y borrar (Delete) (CRUD por sus iniciales en inglés).	4 horas
Acceso a varias tablas de bases de datos con Hibernate	4 horas	**Acceso a varias tablas de bases de datos con Hibernate.**	Aprende a utilizar Hibernate para el manejo de varias tablas relacionadas con una aplicación de facturación de productos.	6 horas
Autenticación.	4 horas	**Autenticación.**	En esta práctica, se restringe el acceso a las páginas a través del sistema de autenticación que ZKOSS nos ofrece.	2 horas
MVVM	2 horas	**Poniendo en**	Aprender a utilizar el patrón MVVM como	6 horas

		práctica el patrón MVVM	alternativa para el patrón MVC y aclarar los posibles escenarios en que cada uno de ellos es más favorable.	
Total Horas Teóricas:	**26**		**Total Horas Prácticas:**	**30**

2.2 Formato de Prácticas.

Número de la practica

Nombre de la practica

Descripción breve

Una breve descripción de lo que abordara la práctica.

Objetivo:

El fin a llegar con la realización de la práctica que se tratara de realizar.

Duración:

Tiempo estimado que tiene el alumno para realizar dicha práctica.

Introducción

Una breve reseña del tema a tratar y de lo que se trabajara en la práctica.

Desarrollo

Procedimientos, funcionalidad, códigos fuentes, capturas de pantallas, etc. Para que se lleve a cabo la práctica.

Ejercicios propuestos

Ejercicios prácticos para profundizar y practicar el tema estudiado.

Referencia

Libros, documentos y enlaces web donde se consultó para el desarrollo de la práctica. Y la profundización del tema tratado.

Capítulo III: Aspectos Teóricos

3.1 Introducción a ZKOSS.

3.2 Validación y Navegación.

3.3 Patrón MVC (Modelo Vista Controlador).

3.4 Acceso a Bases de Datos con Hibernate.

3.5 Autenticación.

3.6 Patrón MVVM (Modelo Vista Vista Modelo).

3.1 Introducción a ZKOSS.

3.1.1 ¿Qué es Java?

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles, juegos, contenido basado en web y software de empresa.

3.1.2 Java está en todas partes

- El 97% de los escritorios empresariales ejecutan Java.
- El 89% de los escritorios (o computadoras) en Estados Unidos ejecutan Java.
- 9 millones de desarrolladores de Java en todo el mundo.
- La primera opción para los desarrolladores.
- La primera plataforma de desarrollo.
- 3 mil millones de teléfonos móviles ejecutan Java.
- El 100% de los reproductores de Blu-ray incluyen Java.
- 5 mil millones de Java Cards en uso.
- 125 millones de dispositivos de televisión ejecutan Java.
- 5 principales fabricantes de equipos originales utilizan Java ME.

3.1.3 Java Enterprises Edition

Es una plataforma de programación para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java. Permite utilizar arquitecturas de N capas distribuidas y se apoya ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones.

La plataforma Java EE está definida por una especificación. Java EE tiene varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc y define cómo coordinarlos.

3.1.4 Servidores para la ejecución de aplicaciones Java EE

Algunos servidores que podemos utilizar:

Como consecuencia del éxito del lenguaje de programación Java, el término servidor de aplicaciones usualmente hace referencia a un servidor de aplicaciones Java EE. Entre los servidores de aplicación Java EE privativos más conocidos se encuentran WebLogic de Oracle (antes BEA Systems) y WebSphere de IBM.

Entre los servidores de aplicaciones libres se encuentran:

Figura 2: Servidores de aplicaciones libre.

Mucha gente confunde **TOMCAT** como un servidor de aplicaciones; sin embargo, es solamente un contenedor de servlets.

3.1.5 ¿Qué es ZKOSS?

ZK es un marco de interfaz de usuario que permite crear increíbles aplicaciones móviles y web sin tener que aprender JavaScript o AJAX. ZKoss nos ofrece las siguientes características:

- El diseño y los componentes Responsive.
- Resolución optimizada automáticamente para la web, móviles y tabletas.
- Sólo necesita escribir una vez con uno de los componentes y un conjunto de código base.
- Excelente experiencia de usuario a través de escritorios y dispositivos.

3.1.5.1 Ventajas

- ✓ Zul permite a los no expertos diseñar eficientemente interfaces de usuario.
- ✓ No es necesario que el desarrollador tenga conocimientos de Ajax o JavaScript.
- ✓ Modelo basado en componentes intuitivo dirigido por eventos.
- ✓ Permite centrar toda la lógica de programación en el servidor.

3.1.5.2 Desventajas

× No es apropiado para aplicaciones con alto grado de interacción.

× Videojuegos de acción.

× Aplicaciones basadas en gráficos vectoriales o tridimensionales

× Programas de edición fotográfica o de video.

3.1.5.3 Diferencias con XUL

≠ Aunque soporta muchos de los componentes XUL, ZK los renderiza en etiquetas HTML.

≠ Esto significa que no cuenta con el poder del motor de renderización XUL Gecko el cual está empotrado en Mozilla Firefox.

≠ ZK es compatible con navegadores que no soportan XUL, tales como Internet Explorer.

≠ ZK mantiene su propio look and feel entre los navegadores. Es independiente del look and feel del motor de renderización XUL de Gecko.

≠ ZK no soporta todos los atributos de XUL e introduce algunas extensiones propietarias.

3.1.5.4 El flujo de ejecución de ZK

➢ Cuando un usuario escribe una dirección URL o hace clic en un hipervínculo en el navegador, se envía una solicitud al servidor Web. Si la URL solicitada tiene el patrón de las URL's ZK, se invocará al ZK Loader para servir a esta petición.

➢ ZK Loader carga la página especificada y la interpreta para crear los componentes.

➢ Después de interpretar toda la página el ZK Loader muestra el resultado en una página HTML. La página HTML se envía de vuelta al navegador acompañado del ZK Client Engine.

➢ ZK Client Engine se ubica en el navegador para detectar cualquier evento provocado por la actividad del usuario (mover el ratón, el cambio de un valor) y una vez detectado se lo notifica ZK AU Engine con el envío de una ZK Request.

3.1.5.5 Herramienta de la capa de presentación

ZK está diseñado para ser tan liviano como sea posible, ya que se centra únicamente en la capa de presentación.

No se requiere ninguna otra tecnología de servidor, y todo el middleware, tales como JDBC, Hibernate, Java Mail, EJB y JMS, funciona adecuadamente con ZK. Por lo tanto, se pueden crear aplicaciones web con tecnologías conocidas sin aprender otras nuevas.

3.1.5.6 Manejado por eventos y centrado en el servidor

Todos los componentes se crean en el lado del servidor. La forma en que los componentes se comunican entre sí es por eventos.

En resumen, ZK refleja la sencillez de desarrollo de aplicaciones de escritorio en el desarrollo de aplicaciones web y ofrece a los usuarios una mayor interactividad y mayor capacidad de respuesta.

3.1.5.7 GUI basada en componentes

ZK es una interfaz gráfica de usuario basada en componentes GUI. ZK proporciona componentes basados en XUL y basado en XHTML, y proporciona el lenguaje de marcado ZUML para el diseño de interfaces de usuario.

Es similar al modelo de programación encontrado en las aplicaciones basadas en GUI de escritorio.

Ejemplo:

Interfaz gráfica:

```
<?page title="Pagina de ejemplo Tema 2" contentType="text/html;charset=UTF-8"?>
<zk>
<window title="Elevar un número a la potencia 2" border="normal" width="200px"
apply="org.zkoss.tema2.tema2">
<grid >          <columns>
                        <column label="" width="min"/>
                        <column label=""/>
                </columns>
                <rows>                  <row>
                                <label value="Numero: "></label>
                                <textbox id="numero" width="100px" />
                        </row>
```

```
                        <row>
                                <button id="btCalcular" label="Calcular" />
                                <label id="resultado" value="" />
                        </row>
                </rows>
        </grid>
</window>
</zk>
```

Nos quedara como se ve en la **Figura 3**.

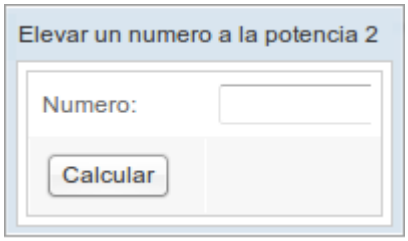


Figura 3: Captura de la aplicación.

Código del controlador:

```
public class tema2 extends SelectorComposer<Component>{
        @Wire
        Textbox numero;
        @Wire

        Label resultado;

        @Listen("onClick = #btCalcular")
        public void fnCalcular() {
                int num = Integer.parseInt(numero.getText());
                int resul = num * num;
                resultado.setValue(String.valueOf(resul));
        }
}
```

3.2 Validación y navegación

3.2.1 Validación de la entrada de usuario

Hay dos formas de validar el valor introducido por el usuario: utilizando constraint o lanzar la excepción **WrongValueException.**

Un elemento de entrada puede estar asociado con una restricción (**constraint**) para validar el valor introducido por un usuario. Si no es suficiente, se puede implementar su propia limitación, o lanzar WrongValueException. Para utilizar la restricción predeterminada, puede especificar una lista de condiciones en InputElement.setConstraint (String), como no positivo y no vacía.

Ejemplo:

```
<textbox id="fullName" constraint="no empty: Por favor escriba su nombre completo"
width="200px"/>

<intbox format="####-####" constraint="no negative,no zero" width="200px" />
```

Para especificar una expresión regular, se debe usar "/" para encerrar la expresión regular de la siguiente manera.

```
<textbox id="email" constraint="/.+@.+\.[a-z]+/: Escriba una direccion de correo valido"
width="200px"/>
```

Observe que la declaración anterior es XML, así que no use "\\" para especificar una barra invertida. En Java:

```
New Textbox().setConstraint("/.+@.+\\.[a-z]+/");
```

Si desea visualizar un mensaje se puede agregar la restricción seguida de dos puntos y el mensaje que desea mostrar cuando falla.

```
<textbox id="email" constraint="/.+@.+\.[a-z]+/: Escriba una direccion de correo valido"
width="200px"/>
```

```
<datebox id="birthday" constraint="no future,no empty" width="200px"/>
```

Observe que el mensaje de error, si se especifica, debe ser el último elemento y comenzar con dos puntos.

3.2.2 Restricciones Predeterminadas:

Condición	**Descripción**
No empty	No está permitido que la caja de texto este vacía.
No future	Fecha en el futuro no es aceptado
No negative	No se permite números negativo.
No past	Fecha en el pasado no se permite
No positive	Números positivo no se permite.
No today	No se permite el día actual
No zero	No se permite que la caja de texto este en cero.
Between yyymmdd and yyymmdd	Que este en medio de yyymmdd y yyymmdd
After yyymmdd	Despues de yyymmdd
Before yyyddmm	Antes de yyymmdd
 End_before End_after After_start After_end . . .	Especifica la posición del cuadro de error. De final a antes. De final a Después. De después a inicio. De después al final.

3.2.3 Validación de la entrada de usuario

Ejemplo:

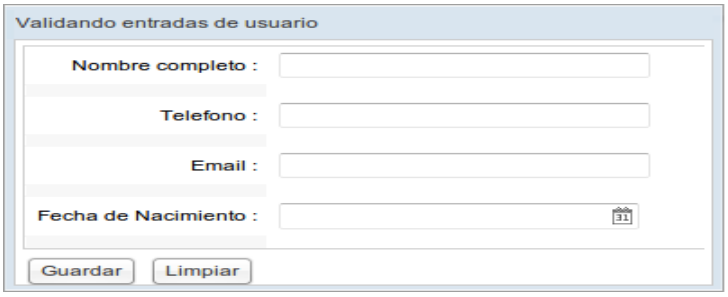


Figura 4: Interfaz de la aplicación validando entradas de usuario.

Se usa **no empty**, para validar que el campo no este vacío.

```
<textbox id="fullName" constraint="no empty: Por favor escriba su nombre completo" width="200px"/>
```

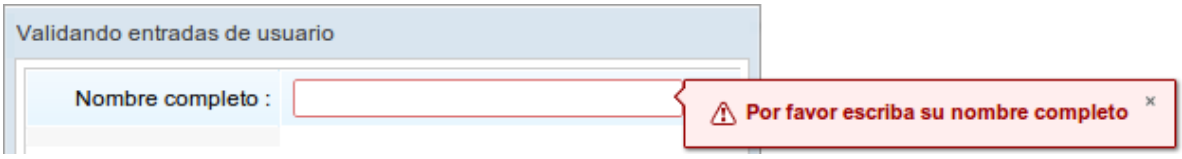


Figura 5: Validar nombre.

```
<intbox format="####-####" constraint="no negative,no zero" width="200px" />
```

no negative, **no zero**, lo ocupamos para validar que los números que se escriban no sean ni negativos, ni que sea 0. Le aplicamos formatos con la etiqueta *format* y le ponemos la máscara como queremos que se vea el número.

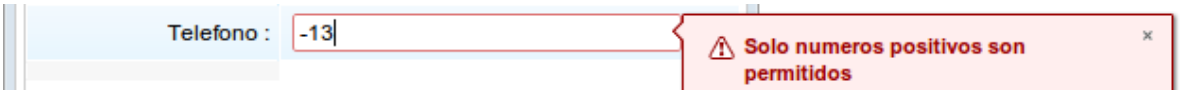


Figura 6: Validar teléfono.

Figura 7: Solo acepta números positivos.

```
<textbox id="email" constraint="/.+@.+\.[a-z]+/: Escriba una direccion de correo valido"
width="200px"/>
```

Validando un correo electrónico para que contenga una estructura valida:

Figura 8: Validar correo.

```
<datebox id="birthday" constraint="no future,no empty" width="200px"/>
```

Ocuparemos **no future** y **no empty**, para validar la fecha de nacimiento del usuario con **no future** esto hará que no podamos tomar una fecha futura.

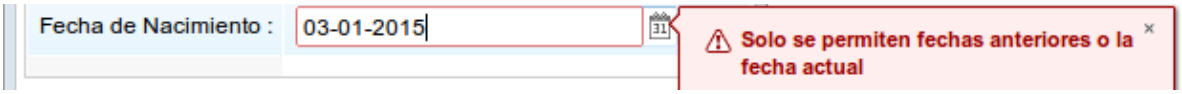


Figura 9: Validar fecha de nacimiento.

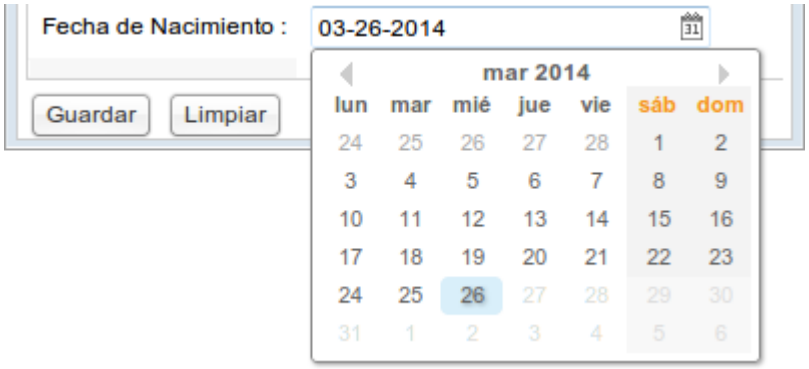


Figura 10: Interfaz para elegir la fecha.

3.2.4 Validación Personalizada

Si quieres una restricción personalizada, se puede implementar la interfaz Constraint y redefinir el método validate. Luego se deberá de especificar la clase que creamos en la propiedad constraint del objeto que deseamos validar.

En el siguiente ejemplo se crea una validación personalizada llamada:
EventNumberConstraint que posteriormente se aplicará a un textbox. Si quieres una restricción personalizada, se puede implementar la interfaz Constraint y redefinir el método validate. Luego se deberá de especificar la clase que creamos en la propiedad constraint del objeto que deseamos validar.

```
public class EventNumberConstraint implements Constraint {
    public void validate(Component comp, Object value) throws WrongValueException {
        if (value != null && (value.intValue() & 1) != 0)
            throw new WrongValueException(comp, "Solamente se permite números regulares,
No "+value);
    }
}
```

Si la validación falla, solo lanzara **WrongValueException** para indicarlo. Se debe tener en cuenta que se tiene que especificar qué componente causa la excepción. A continuación se muestra el uso de la validación anteriormente creada.

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c"?>
<textbox constraint="${c:new(Clase.EventNumberConstraint')}"/>
```

WrongValueException:
Además de lanzar la WrongValueException en Constraint.validate(Component, Object), se puede lanzar WrongValueException en otras situaciones.

Por ejemplo, puede validar el nombre de usuario y contraseña cuando el usuario presiona el botón de inicio de sesión.

3.2.5 Plantillas

En la navegación basada en páginas, los usuarios deben cambiar las páginas con frecuencia. Debemos mantener un diseño de página coherente en toda la aplicación para ayudar a los usuarios a realizar un seguimiento de dónde se encuentren.

Afortunadamente ZK proporciona plantillas para mantener varias páginas con el mismo estilo fácilmente.

Ejemplo de plantilla:

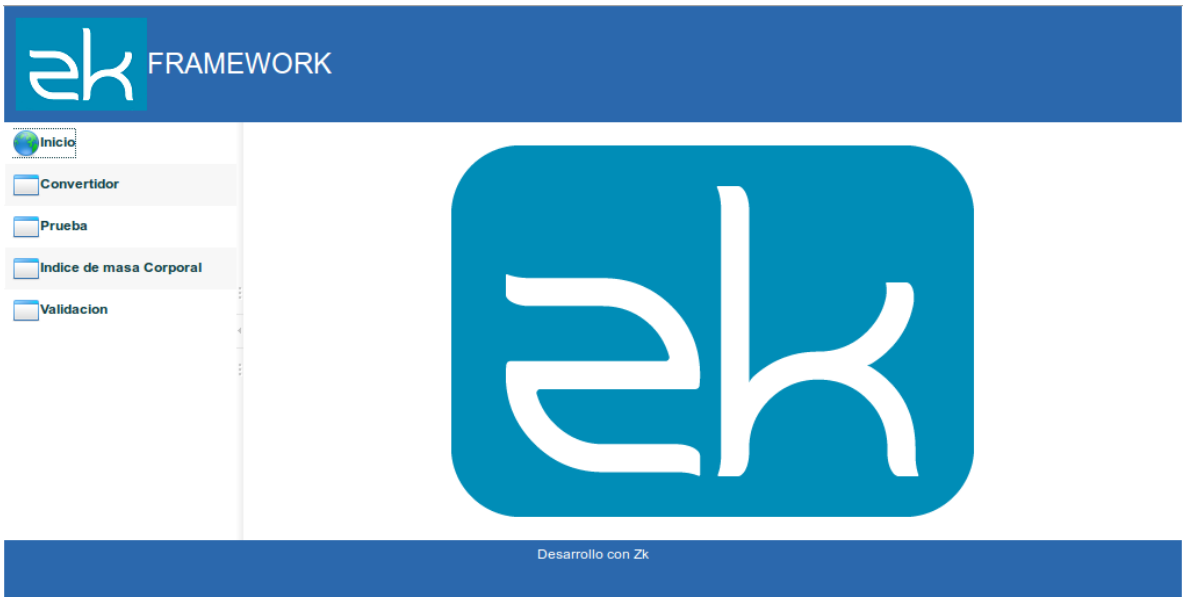


Figura 11: La barra lateral se utiliza para el control de la navegación. Los 4 elementos de menú conducen a diferentes funciones y sólo cambian el contenido de la zona central.

En nuestro ejemplo de aplicación, queremos mantener la cabecera, la barra lateral y el pie de página sin cambios, independientemente de la función que un usuario elige. Sólo la zona central cambia su contenido de acuerdo con la función elegida por los usuarios.

En la navegación basada en páginas, cada función se pone en una página separada y necesitamos tener un estilo coherente. Una forma de hacerlo es copiar la parte duplicada de un zul a otro, pero es difícil de mantener.

Afortunadamente, ZK proporciona una técnica de plantillas que permite definir una plantilla zul y aplicarlo después a varias páginas. Todas las páginas zul que aplican la misma plantilla tienen la misma disposición, por lo que cambiar la plantilla zul puede cambiar el diseño de todas las páginas de una vez.

Los pasos para utilizar plantillas son:

1. Crear una plantilla zul y definir anclas.
2. Aplicar la plantilla en el objetivo zul y definir fragmentos zul para anclajes.

Crear una plantilla no es nada diferente de la creación de un zul normal, pero se debe definir una o más anclajes especificando anotación @insert() en sí. Usted puede dar cualquier nombre para identificar un ancla que se utilizará para insertar un fragmento de zul con el mismo nombre de ancla más tarde.

Ejemplo: Creando una plantilla.

```
<?link rel="stylesheet" type="text/css" href="/estilos.css"?>
<zk>
   <borderlayout hflex="1" vflex="1">
      <north height="100px" border="none" >
         <include src="/cabecera.zul"/>
      </north>
      <west width="200px" border="none" collapsible="true" splittable="true"
minsize="300">
         <include src="/menu.zul"/>
      </west>
      <center id="mainContent" autoscroll="true" border="none" self="@insert(content)">
      </center>
      <south height="50px" border="none">
         <include src="/pie.zul"/>
      </south>
   </borderlayout>
```

```
</zk>
```

Para aplicar la plantilla se hace lo siguiente.

```
<?link rel="stylesheet" type="text/css" href="/estilos.css"?>
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/plantilla.zul"?>
<zk>
        <include self="@define(content)" src="home.zul"/>
</zk>
```

Todas las otras áreas no se han modificado para mantener un constante estilo de diseño entre las funciones. La navegación basada en AJAX, ZK permite implementar otra forma de navegación que sólo actualiza parte necesaria de una página en lugar de volver a cargar la página entera.

La forma más fácil de poner en práctica la navegación basada en AJAX es cambiando el atributo *src* del componente *Include*. Se puede cambiar sólo el contenido parcial de una página en lugar de redirigir a otra página para lograr el propósito de navegación. De esta manera cambian las funciones de navegación, sólo por la sustitución de un grupo de componentes en lugar de toda la página, y por lo tanto, tiene una respuesta más rápida que una navegación basada en página. Pero eso no cambia el URL de un navegador cuando cada tiempo de conmutación cambia a una función diferente.

3.3 El Patrón MVC

El **Modelo Vista Controlador (MVC)** es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación, de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos y características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. Una vez creada una interfaz de usuario, ésta debe responder a la interacción con el Usuario.

A continuación veremos cómo controlar mediante código los componentes de la interfaz de usuario de **manera manual.** La idea es cómo controlar por ti mismo los componentes de la interfaz de usuario Para esto, utilizaremos el patrón de diseño MVC (Modelo-Vista-Controlador).

El **modelo** consta de los datos de la aplicación y las reglas de negocio.

La **vista** cubre la interfaz de usuario. La página zul que contiene los componentes de ZK. La interacción del usuario con los componentes de la vista genera eventos que son enviados a los controladores.

El **controlador** juega el rol de coordinador entre la Vista y el Modelo. En esencia, recibe eventos de la vista para actualizar el Modelo y recibe datos del Modelo para cambiar la presentación, es decir la Vista.

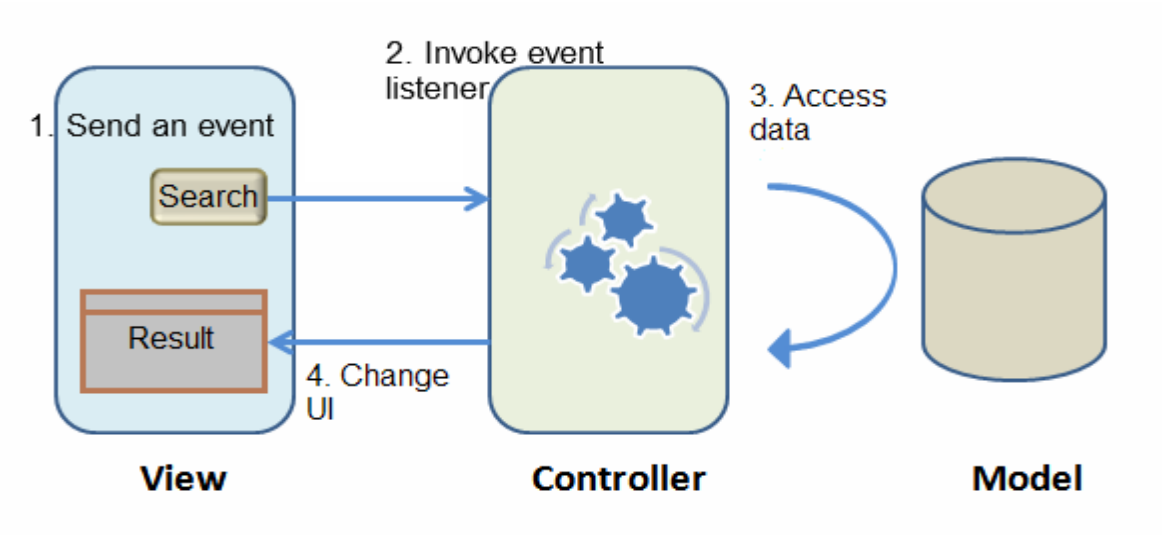


Figura 12: Esquema del patrón MVC.

3.3.1 View (Vista)

La **Vista** es la interfaz de usuario - una composición de componentes. La interfaz de usuario se puede implementar por un documento ZUML o en Java. En áreas de la descripción, ZUML se utiliza para ilustrar el concepto y características.

3.3.2 Controller (Controlador)

El **controlador** es un programa Java que se utiliza para enlazar la interfaz de usuario (view) y los datos (model).

Para una interfaz de usuario simple, no hay necesidad de preparar un controlador (controller). Por ejemplo, los datos de un listbox se pudieron extraer mediante la implementación ListModel. Para el acceso de base de datos típico, la lógica de enlazado (es decir, el control) puede ser manejado por una característica genérica llamada Enlace de datos. En otras palabras, las operaciones de lectura y escritura pueden ser manejadas automáticamente por un enlace de datos genéricos, y no es necesario escribir la lógica de cola en absoluto.

Para llevar a cabo un control personalizado, se puede extender desde SelectorComposer o implementar Composer desde cero. A continuación, especifique el elemento que quiere manejar en un documento ZUML.

3.3.3 Model (Modelo)

El **modelo** son los datos que maneja una aplicación. Dependiendo de los requisitos de aplicación, podría ser cualquier cosa, siempre y cuando el controlador lo sabe. Objetos típicos son POJOs, beans, Spring-managed beans y DAO.

En adición a la manipulación de los datos en un controlador, algunos componentes apoyan el modelo de abstracción para desacoplar la interfaz de usuario y los datos. Por ejemplo, grid, listbox y combobox, soportan ListModel, mientras tree soporta TreeModel.

3.4 Acceso a Bases de Datos con Hibernate

Es un Mapeado de Objeto Relacional (**ORM**), de código abierto, creado por un grupo de desarrolladores dirigido por Gavin King a finales de 2001. Es el ORM más utilizado actualmente – http://www.hibernate.org

3.4.1 Motivación

- Simplificar la programación de la capa de persistencia de las aplicaciones: utilizar JDBC directamente es muy tedioso
- Permite al desarrollador utilizar objetos persistentes en lugar de manipular directamente los datos de la BD
- Asocia clases persistentes con tablas de una BD relacional, haciendo transparente la API de JDBC y el lenguaje SQL
- Soporta relaciones entre objetos y herencia
- No oculta el tipo de BD (relacional), pero sí la BD concreta (Oracle, MySQL, PostgreSQL, etc.)

3.4.2 API dual

- Implementa la API de Persistencia de Java (JPA)
- Proporciona una API propia, porque en aplicaciones reales puede no llegar con la API de JPA

3.4.3 Lenguaje de Consultas

3.4.3.1 HQL (Hibernate Query Language)

- Lenguaje de consultas de búsqueda y borrados/actualizaciones en masa, con semántica orientada a objetos
- Sintaxis parecida a SQL
- Un subconjunto está estandarizado como JPA QL

3.4.3.2 API de Programación para QBC (Query By Criteria)

- Si es necesario, permite lanzar consultas SQL.

3.4.4 Mapeo objeto/relacional

Para que Hibernate sepa cómo mapear las instancias de clases persistentes (llamadas "entidades") a la BD, es necesario especificar meta-información como nombre de la tabla, nombres de las columnas a las que mapear los atributos, etc.

3.4.4.1 Alternativas para especificar la meta-información de mapeo.

Usar ficheros XML

- El código quedaría "limpio" de referencias a elementos de la BD.

Usar Anotaciones sobre la clase persistente

- Hibernate mantiene los nombres de anotaciones JPA (Java Persistence API) para las clases comunes y define otras adicionales.
- El código tiene enlazado nombres de tablas, columnas, etc.
- Sin embargo, es una solución más sencilla, y en consecuencia, representa el **enfoque aconsejado**

3.4.5 Anotación de Clases Persistentes

@Entity

Especifica que es una clase persistente

@Column

Permite especificar información sobre cómo mapear un atributo/propiedad a una columna de la BD

@Id

Especifica el atributo/propiedad clave. Sólo es válido para claves simples.

Hibernate tiene soporte para especificar claves compuestas, si fuese necesario. Normalmente sólo es necesario cuando la BD ya estaba creada y tenga claves compuestas. En general lo recomendable es usar siempre claves simples

@Table

Por defecto, cada entidad se mapea a una tabla con el mismo nombre que la entidad. **@Table** permite especificar un nombre particular para la tabla.

3.4.5.1 Tipos de relaciones

- Uno-a-Uno, Uno-a-Muchos/Muchos-a-Uno, Muchos-a-Muchos
- Uno significa 0..1 o 1; Muchos significa 0..N o N
- Atributos/propiedades que representan relaciones

Tipos: clases persistentes o colecciones de clases persistentes (utilizando las interfaces **Collection**, **List**, **Set** y **Map**)

Se utilizan anotaciones para especificar cómo mapear las relaciones a columnas y tablas

- Direccionalidad
- Unidireccionales o Bidireccionales

Hibernate por defecto utiliza una estrategia **LAZY** para relaciones, excepto en los casos Muchos-a-Uno y Uno-a-Uno (que utiliza una estrategia **EAGER** por compatibilidad con JPA).

LAZY: Aplaza la recuperación de la entidad hasta que es necesaria

EAGER: Cuando se recupera un objeto persistente, se recuperan también todos los objetos persistentes que presenten este tipo de asociación.

¿Qué ocurre cuando se invoca al método get que permite atravesar una relación LAZY?

- Si la relación es Uno/Muchos-a-Uno Hibernate devuelve un proxy del objeto relacionado que contiene la clave primaria de la entidad,
- Los proxies los genera Hibernate dinámicamente en tiempo de ejecución y son clases que extienden a la clase original
- En el caso de un proxy de una entidad que herede de otra, el proxy extiende a la clase raíz
- Si se invoca cualquier método sobre el proxy, excepto **get<<Clave>>**, Hibernate inicializa el proxy (es decir, recupera el estado de BD)
- Si la relación es Uno/Muchos-a-Muchos

- Hibernate devuelve una colección que implementa la interfaz usada (**List**, **Set**, etc.) para modelar la relación
- La colección se inicializa (es decir, recupera las instancias relacionadas de BD) cuando se invoca alguna operación de la colección (**size**, **contains**, iterar, etc.)

Muchos-a-Uno: (Account <--AccountOperation)

- Por defecto, cuando Hibernate obtiene una operación, recupera automáticamente su cuenta asociada – **EAGER.**
- Sin embargo, se ha utilizado el modificador **fetch** en la anotación de la relación para especificar que la estrategia a seguir para recuperar las instancias relacionadas es **LAZY.**

@ManyToOne(optional=false, fetch=FetchType.LAZY)

La documentación de Hibernate aconseja utilizar siempre LAZY para todas las relaciones, para evitar que cada vez que se carga un objeto persistente en memoria se tengan que recuperar los relacionados, lo que podría desencadenar demasiadas consultas o consultas con muchos JOINs.

org.hibernate.Session

- Constituye la principal abstracción de la API de Hibernate.
- Incluye soporte para las operaciones CRUD básicas sobre entidades.
- Permite construir búsquedas complejas mediante el objeto **Query.**
- Permite gestionar transacciones mediante el objeto **Transaction.**
- Representa un gateway contra la BD.
- Mantiene internamente una conexión con la BD.
- Contiene un mapa con las entidades que obtiene/almacena/modifica y funciona a modo de caché de objetos persistentes.
- Sincroniza de forma automática el estado de las entidades persistentes con el de la BD.
- Intenta retrasar la sincronización todo lo posible.
- Operaciones CRUD.

- Insertar un objeto persistente o actualizar uno existente.

3.4.6 Operaciones CRUD (Create Read Update Delete)

- **saveOrUpdate(Object entity)**

  Insertar un objeto persistente o actualizar uno existente.
- **delete(Object entity)**

  Eliminar un objeto persistente.
- **get(Class entityClass, Serializable primaryKey)**

  Recuperar un objeto persistente a partir de su clase y su clave primaria. Devuelve **null** si la instancia no existe.

saveOrUpdate y **get** asocian la instancia persistente con el objeto **Session.**

delete elimina la asociación de la instancia (que deja de ser persistente) con el objeto **Session.**

3.4.7 Transacción Típica con Hibernate

El objeto **Session** se obtiene a partir de un objeto **SessionFactory**, invocando el método **openSession.**

Un objeto **SessionFactory** representa una configuración particular de un conjunto de metadatos de mapping objeto/relacional.
Cuando se crea el objeto **Session**, se le asigna la conexión de BD que va a utilizar.

Una vez obtenido el objeto **Session**, se crea una nueva unidad de trabajo (**Transaction**) utilizando el método **beginTransaction**

Dentro del contexto de la transacción creada, se pueden invocar los métodos de gestión de persistencia proporcionados por el objeto **Session**, para recuperar, añadir, eliminar o modificar el estado de instancias de clases persistentes. También se pueden realizar consultas.

Si las operaciones de persistencia no han producido ninguna excepción, se invoca el método **commit** de la unidad de trabajo para confirmar los cambios realizados. En caso contrario, se realiza un **rollback** para deshacer los cambios producidos.

Sobre un mismo objeto **Session** pueden crearse varias unidades de trabajo.

Finalmente se cierra el objeto **Session** invocando su método **close.**

3.4.8 Ejemplos de Gestión de Persistencia

```
es.udc.pojo.minibank.test.experiments.hibernate.*
```

- **CreateAccount**

Crea una nueva cuenta para el usuario con identificador 1 y balance inicial 200.

```
Account account = new Account(1, 200);
session.saveOrUpdate(account);
```

- **FindAccount <accountId>**

Encuentra una cuenta a partir de su identificador, que recibe como parámetro.

```
Account account = (Account) session.get(Account.class, accountId);
```

- **RemoveAccount <accountId>**

Elimina una cuenta a partir de su identificador, que recibe como parámetro

```
Account account = (Account) session.get(Account.class, accountId);
if (account != null) {
session.delete(account)
```

3.4.9 Ciclo de Vida de una Entidad

- Un objeto es **persistent** si está asociado a una **Session** y tiene correspondencia en BD.
- Un objeto en este estado puede formar parte en transacciones.
- Si durante la transacción se realiza algún cambio sobre él (invocación de un método **set**), Hibernate actualiza automáticamente su estado en BD antes de terminar la transacción.
- Un objeto es **detached** si no está asociada a una **Session** pero tiene correspondencia en BD.
- Un objeto es **transient** si no está asociado a una **Session** y no tiene correspondencia en BD.

3.4.10 Lenguaje de Consultas

La **Session** también proporciona acceso a la API de consultas de Hibernate.

3.4.10.1 session.createQuery(hqlQuery)

HQL (Hibernate Query Language) Permite realizar consultas similares a SQL, en términos de clases persistentes y sus propiedades.

3.4.10.2 session.createCriteria(entityClass)

API de Programación para QBC (Query By Criteria) QBC es una alternativa a HQL, que al igual que ésta permite lanzar consultas, pero a diferencia de ella, no es un lenguaje de consultas, sino una API que permite expresar consultas en términos de objetos (existen objetos para hacer JOINs, restricciones, proyecciones, etc.).

3.4.10.3 session.createSQLQuery(sqlQuery)

SQL (Structured Query Language) Sólo debe de utilizarse cuando las otras opciones no son válidas (cuando se necesite utilizar una característica propia del SQL nativo de la BD).

Lenguaje de Consultas HQL

- Hibernate Query Language (HQL).
- Sintaxis parecida a SQL (para que sea fácil de aprender).
- No usa nombres de tablas ni columnas, sino referencias a objetos y propiedades.

En el caso de fechas, hay que usar la variante que recibe un parámetro que especifica si el valor pasado es un **DATE**, **TIME** o **TIMESTAMP** (alternativamente es posible usar el método **setCalendar**).

- Page-by-Page Iterator.
- **setMaxResults** permite especificar el número máximo de resultados.
- **setFirstResult** permite especificar el índice del primer resultado (de 0 en adelante).

Ambos métodos devuelven otra vez el objeto **Query.**

List permite ejecutar una consulta de lectura y devuelve una lista con los resultados.

Se utiliza **uniqueResult** cuando se sabe que sólo habrá un resultado o ninguno (**null** en este caso).

Lanza la excepción **NonUniqueResultException** si la consulta devuelve más de un resultado. Una DAO define una interfaz para las operaciones de persistencia (métodos CRUD y de búsqueda) relacionadas con una clase persistente particular.

3.4.11 Existen métodos comunes a todas las entidades.

Utilizando Generics (características introducida en Java SE 5.0), se puede diseñar un DAO genérico con las operaciones comunes para todas las clases persistentes como **save, find, exists y remove.** Cada entidad persistente tendrá su propio DAO, que extenderá el genérico para añadir operaciones propias.

- Normalmente añadirá operaciones de búsqueda utilizando diferentes criterios.
- No se ha definido el método **update** en el DAO porque Hibernate actualiza los objetos **persistent dirty** automáticamente.
- El DAO Genérico asume un ORM con estado, es decir, uno que dispone de una sesión en memoria con los objetos accedidos por el caso de uso (esos objetos son los objetos en estado persistent).
- El ORM sabe si los objetos de la sesión están modificados, si tienen que ser insertados o eliminados, de manera que cuando se hace el **commit**(o se fuerza un **flush** explícitamente), se lanzan las correspondientes sentencias de actualización/inserción o eliminación.
- No es factible definir un DAO completamente genérico.
- Éste es genérico pero asumiendo un ORM con estado.
- En cualquier caso, una vez definido, facilita el desarrollo de los casos de uso (y esa es su principal virtud).
- El DAO genérico se encuentra en el módulo **pojo-modelutil.**

La interfaz parametrizada del DAO genérico recibe 2 argumentos

- **E**, es la clase persistente para la que se implementará el DAO.
- **PK**, define el tipo del identificador de la clase persistente. El identificador debe ser **Serializable.**

Los métodos están definidos en base a esos parámetros y no están acoplados a ninguna tecnología de persistencia.

Pequeño ejemplo de hibernate donde se muestra 3 cajas de texto el cual se utiliza para introducir el id, el nombre y la dirección de una persona .ya teniendo los datos guardados se puede editar o borrar el registro que se encuentra en la base de datos .como se muestra a continuación.

Figura 13: Ejemplo hibernate.

3.5 Autenticación

Los servidores de aplicaciones crean una sesión para mantener el estado de un cliente. Cuando el servidor recibe la primera solicitud de un cliente, crea una sesión y le da un **identificador único**. De esta manera, el cliente envía al servidor una nueva solicitud con ese identificador único de sesión. Así, el servidor sabe a qué sesión pertenece la solicitud y la procesa.

La clase **Session** de ZK es un envoltorio de javax.servlet.http.HttpSession y sirve para almacenar los datos de usuario de una sesión:

- Obtener la sesión actual: Session.getCurrent ().
- Almacenar datos: Session.setAttribute ("key", data).
- Obtener datos: Session.getAttribute ("key").

Casi todas las aplicaciones requieren de un mecanismo de seguridad y la autenticación es una parte fundamental. Algunos recursos sólo deben estar disponibles para los usuarios autenticados.

La autenticación es el proceso para verificar que un usuario es quien dice ser. Después que un usuario se ha autenticado, la aplicación debe recordar al usuario e identificar las siguientes solicitudes para no autenticarlo repetidamente. Por lo tanto, una buena práctica consiste en almacenar las credenciales de un usuario en la sesión. De esta manera, la aplicación sabrá si una solicitud viene de un usuario autenticado.

3.5.1 Seguridad en las páginas

Aunque se implemente un mecanismo de autenticación, un usuario todavía podría acceder a una página si conoce su URL.

Por lo tanto, debemos proteger las páginas de accesos ilegales verificando las credenciales del usuario en su sesión cuando la página es solicitada por un usuario.

3.5.2 Inicialización en una página

ZK nos permite inicializar una página implementando un Initiator. Cuando aplicamos un iniciador a un zul, ZK lo usa para llevar a cabo una inicialización antes de crear los componentes.

Así, podemos crear un iniciador para verificar la existencia de las credenciales de un usuario en la sesión actual. Si el usuario está ausente, el acceso es ilegal y redirigiremos a la página de registro.

3.5.3 El atributo "if"

Es posible usar una expresión EL en el atributo "if" para determinar la creación de componentes, dependiendo de las credenciales de un usuario en una sesión.

Si el resultado de la evaluación de la expresión EL es verdadera, los componentes serán creados, de lo contrario, no se crearán.

3.5.4 Login

Es común solicitar un nombre de usuario y una clave para autenticación. Esto se hace desde una página que no tiene protección de autenticación y que presenta cajas de texto para introducir el usuario y la contraseña; ésta última debe ser de tipo "password".

3.5.5 Logout

Cuando el usuario cierra sesión (logout), deberemos eliminar sus credenciales de la sesión y redirigirlo.

3.5.6 Ejemplo de autentificación:

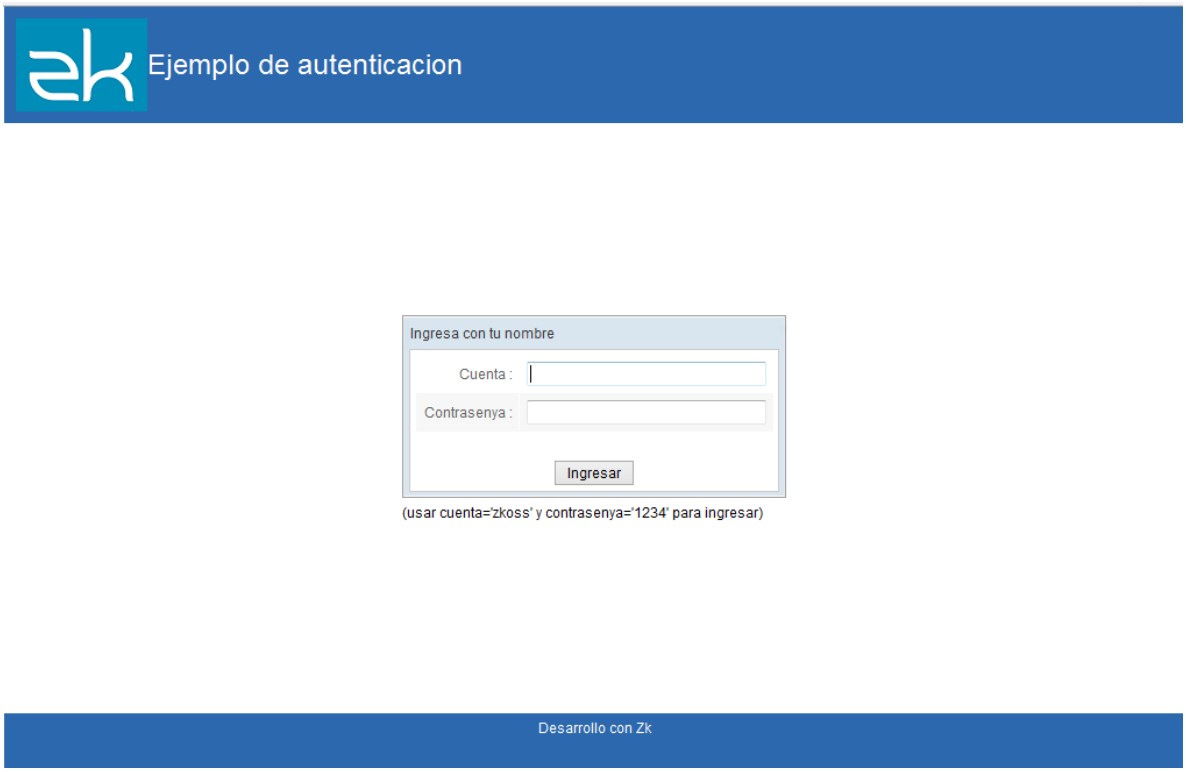


Figura 14: Ejemplo de autenticación.

Figura 15: Presentación de la página index luego de iniciar sesión.

3.6 El Patrón MVVM

El enfoque que introducimos en esta parte es que ZK controle por ti la interacción del usuario sobre los componentes de la vista. A esta manera de proceder se la identifica como patrón de diseño Modelo-Vista-VistaModelo (MVVM).

Partes:

- El **Modelo** consta de los datos de la aplicación y las reglas de negocio.
- La **Vista** cubre la interfaz de usuario.
- La **VistaModelo** es el tipo de Vista abstracción que contiene el estado y el comportamiento de una vista. Es el encargado de exponer los datos del modelo a la vista y proporcionar acción necesaria solicitada a la Vista.

Pero la capa **VistaModelo** no debe contener referencias a los componentes de la interfaz de usuario. Es ZK quien controla la comunicación y sincronización entre la Vista y la VistaModelo. Por lo tanto, bajo este enfoque, lo único que preparamos es una clase VistaModelo con los **setters**, **getters** y métodos que necesitemos, después los asignamos mediante las expresiones de **"data binding"** a los atributos de los componentes del zul (nuestra vista).

Hay un "**binder**" en ZK que sincroniza los datos entre la capa VistaModelo y los componentes de la vista automáticamente, en función de lo que definamos en las "binding expressions".

Por lo tanto, no es necesario que nosotros controlemos los componentes.

1. El usuario hace clic en el botón "Search" y el evento correspondiente es enviado.
2. El "binder" de ZK invoca el correspondiente método en la capa VistaModelo.
3. El método accede a la información de la capa Modelo y actualiza las propiedades necesarias de la capa VistaModelo.
4. El "binder" de ZK lee las propiedades de la VistaModelo y actualiza el estado de los componentes.

3.6.1 Abstrayéndonos de la vista

La capa de VistaModelo es una abstracción de la vista. Por lo tanto, cuando diseñamos la capa de VistaModelo, debemos analizar la funcionalidad de la interfaz de usuario (vista) para identificar el estado de los componentes y el comportamiento que esperamos de los mismos.
La capa de VistaModelo debe tener 3 variables (para los estados) y un método (para el comportamiento).

En ZK, crear la capa de VistaModelo es como crear un **POJO**, y este nos presenta su estado como si de las **propiedades** de un JavaBean se tratase, a través de métodos **setters** y **getters**.

3.6.2 Anotación

En ZK MVVM, cualquier comportamiento que pueda requerir la Vista lo entendemos como "comando" en la capa de VistaModelo.

Podemos enlazar un evento de un componente con un comando de la VistaModelo y ZK invocará el método cuando el evento que hemos enlazado sea lanzado desde la Vista.

Para permitir que ZK conozca qué comportamiento (método) es el que queremos lanzar, debemos aplicar una anotación de tipo @Command sobre el método. El nombre del comando es utilizado en la expresión que usamos para enlazar el evento con el método.

Bajo el patrón MVVM, nosotros construimos nuestra interfaz de usuario del mismo modo que lo hacemos con el patrón MVC. Entonces establecemos la relación entre la capa Vista (zul) y la capa VistaModelo mediante expresiones de "data binding" sobre los atributos de los componentes, y dejamos que ZK maneje el componente y sus eventos por nosotros.

Para asociar un componente a la capa VistaModelo, podemos aplicar un "composer" llamado "org.zkoss.bind.BindComposer". Esta clase "composer" procesa las expresiones de "data binding" e inicializa las clases de la capa VistaModelo. Luego asociamos el componente a la capa de VistaModelo estableciendo su atributo "viewModel" con la siguiente sintaxis:

@id('ID') @init('FULL.QUALIFIED.CLASSNAME')

@id() es usado para establecer el id de la variable que queramos de la capa VistaModelo. Usaremos ese id para referencia las propiedades de la capa VistaModelo (por ejemplo vm.carList) con las expresiones "data binding". Debemos proveer un nombre de clase completamente cualificado para @init() para inicializar el objeto de la capa VistaModelo**.**

3.6.3 Asociando la interfaz de usuario con el "ViewModel"

```
<window title="Search" width="600px" border="normal"
apply="org.zkoss.bind.BindComposer" viewModel="@id('vm')
@init('tutorial.SearchViewModel')">
<!-- omit other tags-->
</window>
```

Después de asociar la capa VistaModelo con el componente, todos sus componentes hijos pueden acceder a las propiedades de la capa VistaModelo asociada al componente padre.

En cuanto hemos declarado variables en la clase de la capa de VistaModelo para los estados de los componentes en la sección anterior, podemos asociarlos a atributos de componentes de la vista.

Después de asociar un atributo de un componente a la VistaModelo, ZK automáticamente sincronizará la información contenida en el atributo y la propiedad correspondiente en la capa de VistaModelo por nosotros. Podemos especificar qué atributo es asociado a qué propiedad mediante una expresión "data binding" directamente en un atributo del componente de la capa de la Vista con la sintaxis:

@bind(vm.aProperty)

Recuerda que "vm" es el id que le dimos en la anotación @id() anteriormente y ahora lo usamos para referenciar desde la capa de la Vista el objeto de la capa VistaModelo.

Capítulo IV: Prácticas de laboratorio

Práctica 0: Instalación del entorno.

Práctica 1: Controles y Manejo de Eventos.

Práctica 2: MVC.

Práctica 3: Acceso a Bases de Datos con Hibernate.

Práctica 4: Acceso a Varias Tablas de Bases de Datos con Hibernate.

Práctica 5: Autenticación.

Práctica 6: MVVM.

Práctica 0

Practica 0:

Instalación del entorno.

Descripción breve:

En esta práctica, se explicara todo el procedimiento necesario para tener el entorno de trabajo de ZK Studio completamente funcional y desarrollar un pequeño proyecto.

4.1.1 Objetivo:

Instalar y configurar el entorno de desarrollo eclipse, agregando apache Tomcat como servidor Web.

4.1.2 Duración:

4 horas.

4.1.3 Introducción

En esta práctica lograremos instalar el entorno de desarrollo en nuestro caso Eclipse Juno 4.2, descargar ZK Community Edition (ZK) agregarlo a Eclipse y activarlo, e instalar Java SE Development Kit.

4.1.4 Desarrollo

4.1.4.1 ¿Qué es ZKOSS?

ZK es un marco de interfaz de usuario que permite crear increíbles aplicaciones móviles y web sin tener que aprender JavaScript o AJAX. ZKoss nos ofrece las siguientes características:

- El diseño y los componentes Responsive.
- Resolución optimiza automáticamente para la web, móviles y tabletas.
- Sólo necesita escribir una vez con uno de los componentes y un conjunto de código base.
- Excelente experiencia de usuario a través de escritorios y dispositivos.

4.1.4.2 ¿Qué es ZK Studio?

ZK Studio es la manera más fácil de crear un proyecto nuevo. El nuevo proyecto del asistente guía a los desarrolladores a través de la creación de proyectos ZK paso a paso. Por lo que es excepcionalmente fácil para elegir el nombre del proyecto, la versión ZK y ejecución de destino.

Además el asistente por defecto, ZK Studio le permite crear un nuevo proyecto con Maven. ZK Studio creará el proyecto para usted lo que le permite concentrarse en la construcción de su proyecto.

4.1.4.3 Pre-requisitos para la instalación de ZK Studio

Existen algunos programas que es necesario instalar antes de poder instalar ZK Studio.

4.1.4.4 Java SE Development Kit

Java versión 5 o superior, se requiere para ejecutar Eclipse, puede descargarlo desde el sitio oficial de Oracle. Esta puede ser descargada desde el siguiente enlace:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Una vez allí dar clic en el ícono:

Figura 16: Icono JDK.

Esto nos mostrará una tabla que nos permitirá descargar el paquete correspondiente para la plataforma que estemos empleando:

Java SE Development Kit 7u25

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux x86	80.38 MB	jdk-7u25-linux-i586.rpm
Linux x86	93.12 MB	jdk-7u25-linux-i586.tar.gz
Linux x64	81.46 MB	jdk-7u25-linux-x64.rpm
Linux x64	91.85 MB	jdk-7u25-linux-x64.tar.gz
Mac OS X x64	144.43 MB	jdk-7u25-macosx-x64.dmg
Solaris x86 (SVR4 package)	136.02 MB	jdk-7u25-solaris-i586.tar.Z
Solaris x86	92.22 MB	jdk-7u25-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	22.77 MB	jdk-7u25-solaris-x64.tar.Z
Solaris x64	15.09 MB	jdk-7u25-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	136.16 MB	jdk-7u25-solaris-sparc.tar.Z
Solaris SPARC	95.5 MB	jdk-7u25-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	23.05 MB	jdk-7u25-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	17.67 MB	jdk-7u25-solaris-sparcv9.tar.gz
Windows x86	89.09 MB	jdk-7u25-windows-i586.exe
Windows x64	90.66 MB	jdk-7u25-windows-x64.exe

Figura 17: Tabla de enlaces para descargar JDK.

Se debe aceptar la licencia para el uso del paquete y descargar la versión que necesitemos. Si estamos en una plataforma Linux, generalmente podemos instalar el JDK de java desde el gestor de paquetes que normalmente traen incluidos. En el caso de Ubuntu 13.10 bastaría con escribir en la consola:

```
sudo apt-get install openjdk-7-jdk
sudo apt-get install openjdk-7-jre
```

4.1.4.5 Eclipse IDE for Java EE Developers

ZK Studio es un Plug-in para Eclipse, por lo tanto es necesario instalar Eclipse y se recomienda descargar el paquete Eclipse IDE para desarrolladores Java EE. En nuestro caso emplearemos Eclipse Juno.

Se puede descargar el Eclipse Juno desde el siguiente enlace:

http://www.eclipse.org/downloads/packages/release/juno/sr1

Estando allí seleccionar el paquete correspondiente para el sistema operativo que ha de emplear:

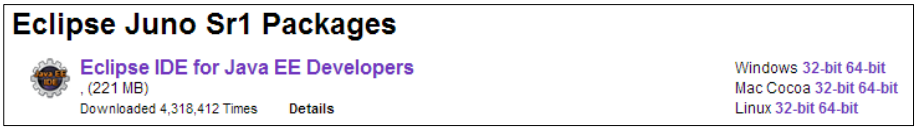


Figura 18: Enlace de descarga eclipse Juno.

La instalación se realiza con sólo extraer el archivo Zip descargado y ejecutar eclipse.exe (en Windows) o Eclipse (en Unix o Linux).

4.1.4.6 Servidor de aplicaciones

Antes de desarrollar aplicaciones web en Java utilizando ZK Framework, es necesario instalar un servidor de aplicaciones. Apache Tomcat es uno de los contenedores web más populares. Las versiones de Tomcat 6 y 7 son compatibles. En nuestro caso trabajaremos con la versión 7 de Tomcat. Para Windows puede ser descargado:

http://www.eng.lsu.edu/mirrors/apache/tomcat/tomcat-7/v7.0.42/bin/apache-tomcat-7.0.42.exe

Durante la instalación se pedirán las credenciales para crear al usuario root del servidor Tomcat. Si se desea instalarlo en Linux, se puede hacer desde los repositorios de Linux. En el caso de Ubuntu 12.04 lo haríamos de la siguiente manera:

```
sudo apt-get install tomcat7
```

Para una instalación personalizada de apache Tomcat, visite la página del proyecto en:

http://tomcat.apache.org/tomcat-7.0-doc/setup.html

4.1.4.7 Instalar ZK Studio

1. Descargue el archivo Zip de instalación desde la página de descarga de ZK Studio.
2. Abre Eclipse, haga clic en [**Help**] en el menú y seleccione [**Install New Software…**]. Una ventana de instalación emergente se mostrara, haga clic en **Add.** Como se ve en la **Figuta 19-20**.

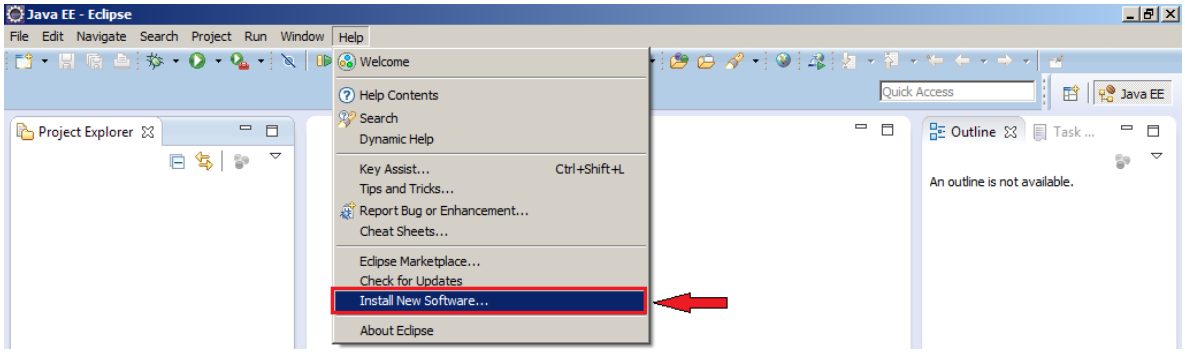


Figura 19: Instalar zk studio.

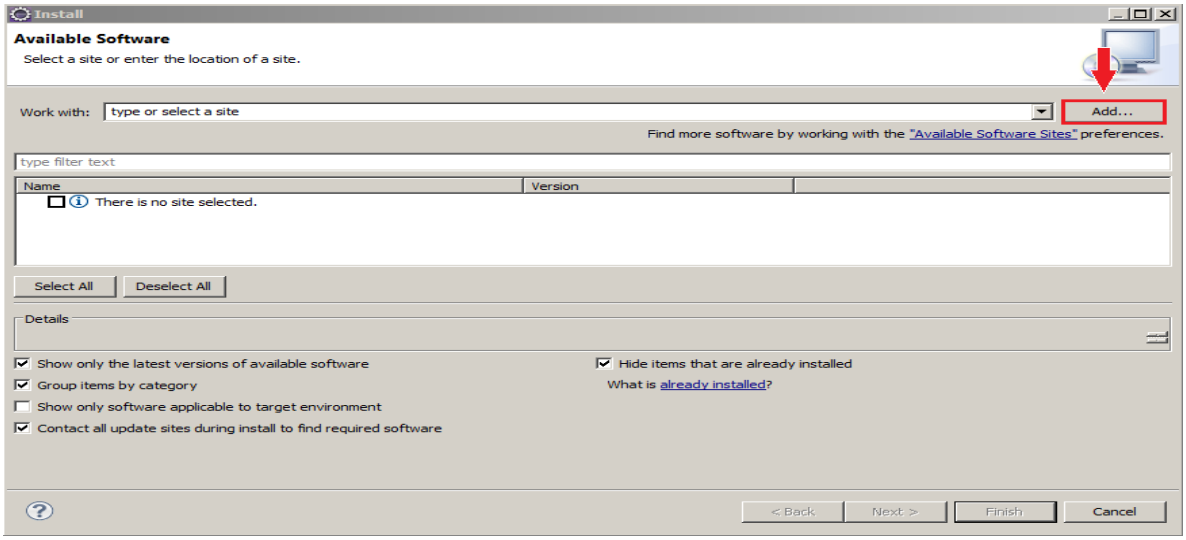


Figura 20: Clic en "Add…" para agregar software.

3. Haga clic en **Archive…** para abrir el explorador de archivos. A continuación, utilice el explorador de archivos para seleccionar el archivo descargado y haga clic en "OK" para instalar. **Figura 21.**

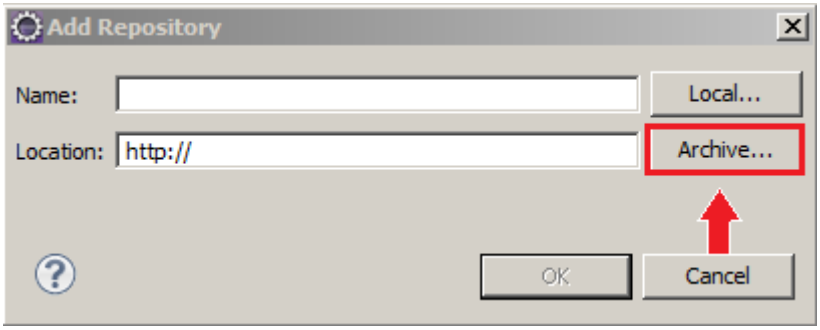


Figura 21: Seleccionar archivo.

4. A continuación se muestra una ventana en la cual hay que aceptar los términos de licencia de ZK Studio.

4.1.4.8 Activar ZK Studio

En el menú de Eclipse, elija por favor [**Help**] y seleccione [**Activate ZK Studio**]. Un cuadro de diálogo aparecerá (**Figura 22**):

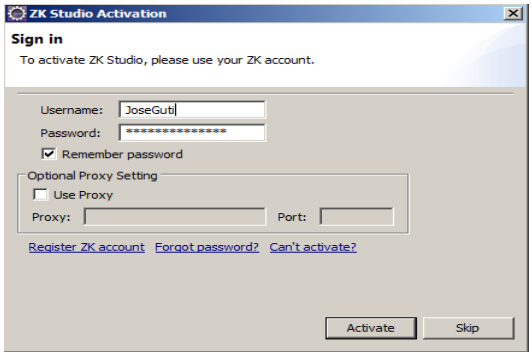


Figura 22: Activar zk studio.

4.1.4.9 Ejecución de un Proyecto ZK

1. Click derecho sobre **AppZK** en el explorador de proyectos y en el menú mostrado seleccionar **Run As -> Run on Server. Figura 23.**

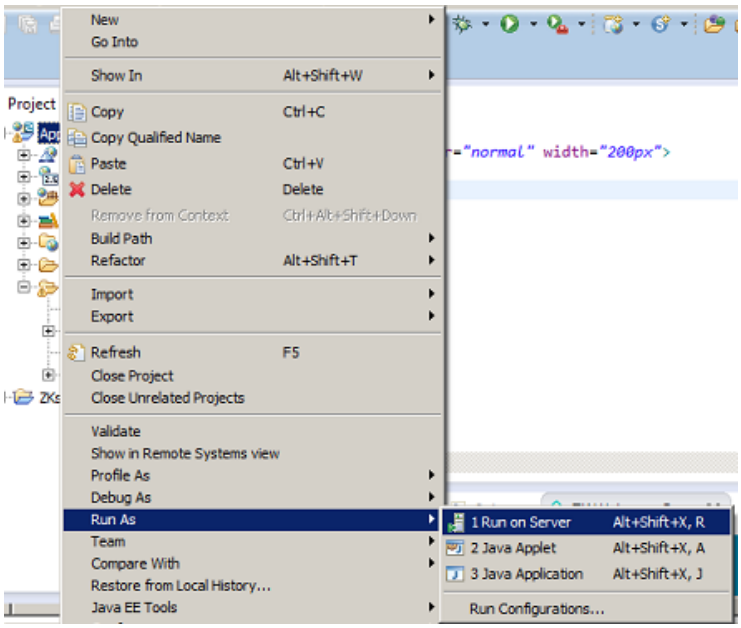


Figura 23: Ejecutar una aplicación.

2. Si no ha definido un servidor antes, Eclipse mostrara un dialogo con la opción seleccionada "Manually define new server". Por lo tanto seleccione **Apache -> Tomcat v6.0 Server** en el área "server type" y a continuación haga click en **Finish**.

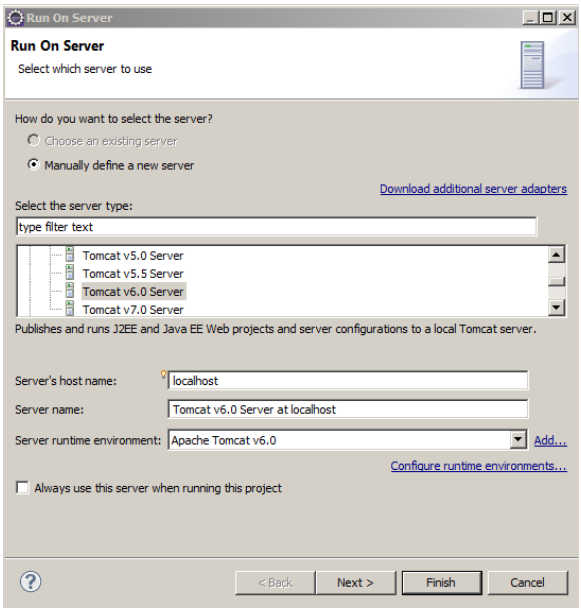


Figura 24: Seleccionar servidor.

3. Eclipse mostrara el resultado en su navegador interno o puede ver el resultado en su navegador preferido.

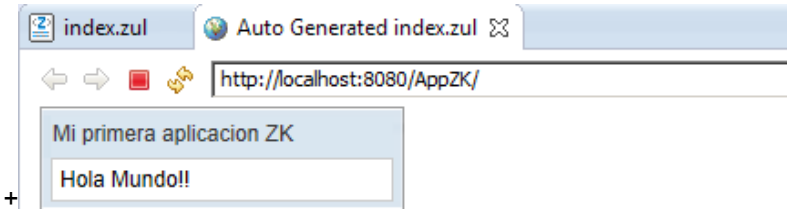


Figura 25: Aplicación en ejecución.

4.1.5 Referencia

- ✓ http://www.zkoss.org/download/zk
- ✓ http://books.zkoss.org/wiki/ZK_Installation_Guide
- ✓ http://books.zkoss.org/wiki/ZK%20Installation%20Guide/Setting%20up%20IDE/Eclipse%20with%20ZK%20Studio
- ✓ http://books.zkoss.org/wiki/ZK%20Installation%20Guide/Quick%20Start/Create%20and%20Run%20Your%20First%20ZK%20Application%20with%20Eclipse%20and%20ZK%20Studio

Práctica 1

Práctica 1:

Controles y manejo de eventos.

Descripción breve:

Esta práctica se basara en la explicación de los principales controles y manejadores de eventos del framework, y la realización de un proyecto sencillo utilizando interfaces de usuarios y layout.

4.2.1 Objetivo:

Familiarizarse con el framework realizando pequeños proyectos utilizando los controladores más conocidos, y manejo eventos de una forma sencilla.

4.2.2 Duración:

4 horas.

4.2.3 Introducción

Interfaz de usuario y diseño.

La construcción de la Vista en ZK es básicamente la creación de componentes y hay dos maneras de hacerlo: Java (programático) y basado en enfoque XML (declarativa). Incluso puedes mezclar estos dos enfoques.

ZK también proporciona un lenguaje de formato XML denominada interfaz de usuario ZK (lenguaje de marcado ZUML). Cada elemento XML instruye ZK Loader para crear un componente. Cada atributo XML describe lo que se debe asignar al componente creado.

4.2.4 Desarrollo

4.2.4.1 Controlando componentes

Al controlar un componente, debemos recuperarlo primero. En SelectorComposer, cuando especificamos una anotación **@Wire** en el campo o método setter, el SelectorComposer automáticamente encontrara el componente y asignara al campo o aprobara el método setter. Por defecto SelectorComposer encontrara el componente cuyo id y tipo equivalen al nombre y tipo de la variables respectivamente.

```
public class eje extends SelectorComposer<Component>  {

        @Wire
        private Textbox cel;
...
```

> **Nota: SelectorComposer** busca un **Textbox** cuyo id sea **"cel"** cuando lo encuentra se lo asigna a la variable **cel**.

4.2.4.2 Eventos y manejadores de eventos

Un evento ZK es una abstracción de una actividad hecha por el usuario, una notificación hecha por una aplicación, y una invocación empujada por el servidor. Por ejemplo, un usuario hace clic en un botón en el navegador, este desencadenara onClick enviado al servidor. Si hay un escuchador de eventos registrado para el evento onClick del botón, Zk pasara el evento al oyente para manejarlo. El mecanismo de detectores de eventos nos permite manejar todas las interacciones del usuario del lado del servidor. La llamada a los eventos los controlaremos con el componente **@listen**, este se encargara de escuchar la petición del evento y capturarla para realizar la función de esta.

```
...
@Listen ("onClick=#calcular")
        public void calcular()
        {
...
```

4.2.4.3 Creación de layout

Para crear un componente de ZK, tenemos que utilizar un lenguaje basado en XML llamado ZUL y todos los archivos escritos en ZUL deben tener la extensión". Zul ". En los archivos de zul, uno de los componentes se puede representar como un elemento XML (tag) y se puede configurar el estilo de cada componente, el comportamiento y la función mediante el establecimiento de los atributos del elemento.

1. En eclipse crearemos un nuevo proyecto ZK llamado **layout. (Figura 26.)**

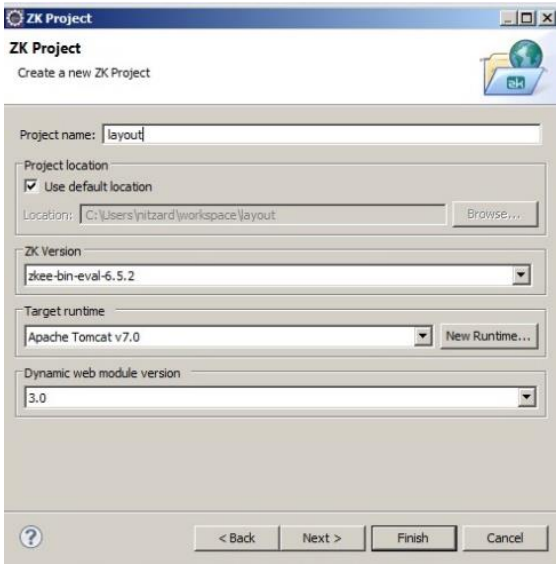


Figura 26: Ejecutar la aplicación.

2. Crearemos un archivo **layout.zul** y escribiremos el siguiente código.

```
<zk>
<borderlayout hflex="1" vflex="1">
<north height="100px" border="none">
<label style="font-size:50px">Norte de la cabecera</label>
</north>
<west width="260px" border="none" collapsible="true" splittable="true" minsize="300">
<label style="font-size:50px">Menu izquierdo</label>
</west>
<center id="mainContent" autoscroll="true">
<label style="font-size:50px">Centro del contenido </label>
</center>
<south height="50px" border="none">
<label style="font-size:50px">Pie de pagina  </label>
</south>
</borderlayout>
</zk>
```

3. Luego le damos clic derecho sobre el archivo y le damos ejecutar con el servidor.
4. Nos quedare de la siguiente manera. **Figura 27.**

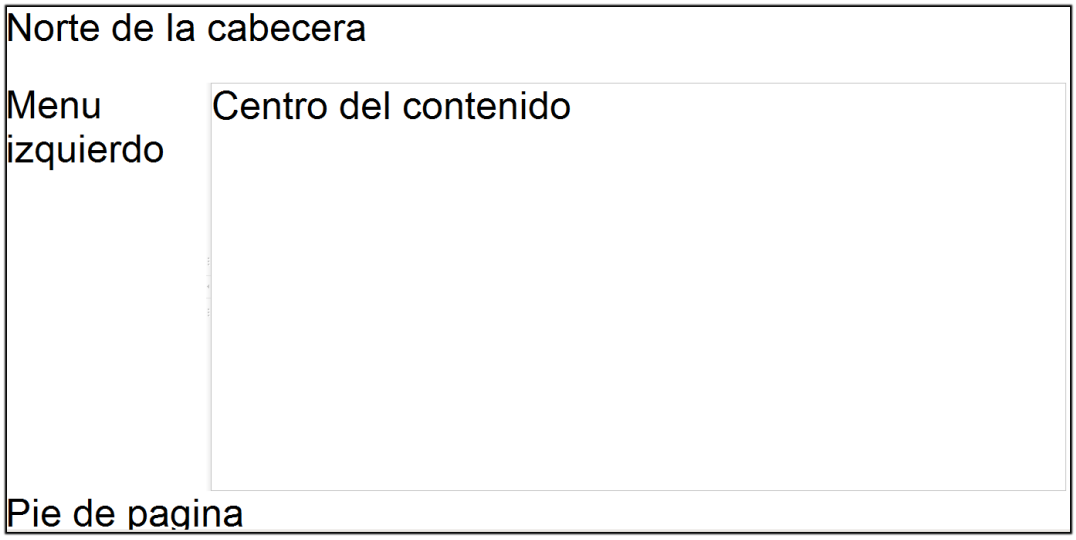


Figura 27: Vista final del código anterior.

4.2.4.4 Manejando controles y eventos con ZKoss

1. Crearemos un convertidor de temperatura de Grados C° a Grados F°. Primero agregaremos un proyecto zk, el cual lo llamaremos **convertidor**.
2. Accederos al archivo index.zul, agregaremos lo siguiente:

```
<window title="Convertidor de temperatura!!!" border="normal" width="300px"
apply="convert.convert">

 <label value="Grados C°:  " /><textbox id="cel" value=""/><separator/>

 <label value="Grados F°:   " /><textbox id="fa" value=""/><separator/>

<button label="calcular" id="calcular" /><space/><button label="limpiar" id="limp" />

</window>
```

3. En la carpeta JavaResources, agregaremos una clase llamada **convert** y el paquete de la misma clase también lo llamaremos **convert. Figura 28.**

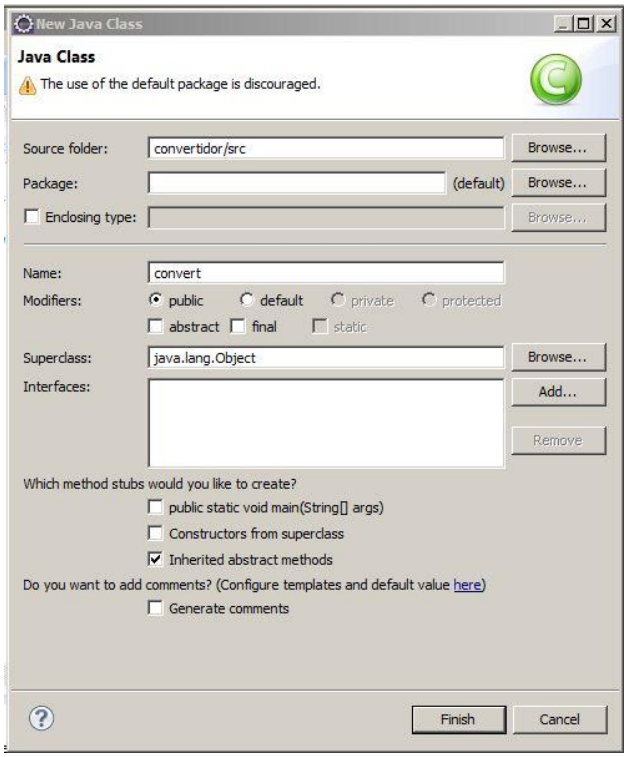


Figura 28: Crear clase nueva.

4. Le pondremos el siguiente codigo para enlazar la interfaz gráfica con la clase java para esto le pondremos la anotacion @Wire, a estas variables le pondremos el mismo nombre que está en la interfaz gráfica.

```
public class convert extends SelectorComposer<Component> {
        @Wire
        private Textbox cel;
        @Wire
        private Textbox fa;
        @Wire
        private Button cal;
        @Wire
        private Button limp;
```

5. Luego crearemos una función llamada calcular, esta se encargara de calcular los grados. También pondremos un componente llamado @Listen para controlar el evento clic del ratón en el botón d la interfaz gráfica.

```
@Listen ("onClick=#calcular")
        public void calcular()
        {
        float result;

         if(cel.getText().trim().length()==0)
                {
                result= (float) ((Integer.parseInt(fa.getText())+32)*0.55);
                cel.setValue(String.valueOf(result));
                }

        if(fa.getText().trim().length()==0)
                 {
                 result= (float) ((Integer.parseInt(cel.getText())*1.8)+32);
                 fa.setValue(String.valueOf(result));
                 }
```

6. Nos quedare de la siguiente manera.

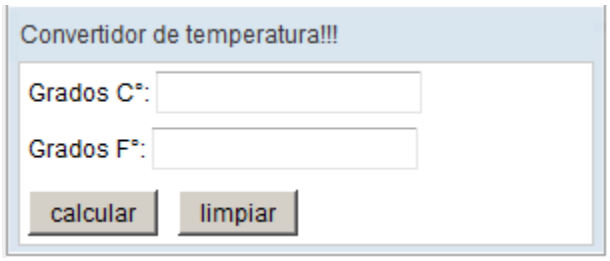


Figura 29: Aplicación en ejecución.

4.2.5 Ejercicios propuestos

a. Hacer un proyecto donde podamos saber la masa corporal de las personas ingresando su **altura (mtr)** y su peso **(Kg)**, la fórmula es la siguiente $\boldsymbol{IMC = \frac{peso}{altura2}}$

Clasificarlo de acuerdo a su peso

Bajo peso (IMC < 18,5)

Rango normal (IMC = 18,5-25)

Sobrepeso (IMC = 25-30)

Obesidad (IMC > 30)

b. Hacer un convertidor de monedas.

Córdoba	Dólar
Dólar	Córdoba
Euro	Dólar
Dólar	Euro
Córdoba	Euro
Euro	Córdoba

4.2.6 Referencia

- ✓ http://books.zkoss.org/wiki/ZK%20Component%20Reference
- ✓ http://books.zkoss.org/wiki/ZK_Getting_Started/Learn_ZK_in_10_Minutes

Práctica 2

Práctica 2:

Manipulando la entrada de usuario y emplear el patrón MVC.

Descripción breve:

En esta práctica diseñaremos un pequeño ejercicio donde emplearemos la manipulación de entradas de usuario con controles comunes, implementamos el patrón MVC (Model View Controller) y agregarlo a una plantilla con navegación basada en páginas.

4.3.1 Objetivo:

Preparar a los estudiantes para que aprendan a desarrollar con el patrón MVC de forma sencilla.

4.3.2 Duración:

6 horas.

4.3.3 Introducción

Implementamos toda la gestión de eventos y la lógica de presentación en un controlador sin presencia del código en el archivo ZUL. Este enfoque hace de la responsabilidad de cada función (**model** "Modelo", **view** "Vista" y **controller** "Controlador") más coherente y le permite controlar componentes directamente. Es muy intuitivo y muy flexible.

- EL **View** significa interfaz de usuario. Representa la página zul que contiene los componentes de ZK. La interacción de un usuario con componentes desencadena eventos que se enviarán a los controladores.
- El **Controller** desempeña el coordinador entre la Vista y modelo. Se recibe eventos de la Vista para actualizar el modelo y recuperar datos para cambiar la presentación de la vista.
- El M**odel** se compone de los datos de la aplicación y las reglas de negocio.

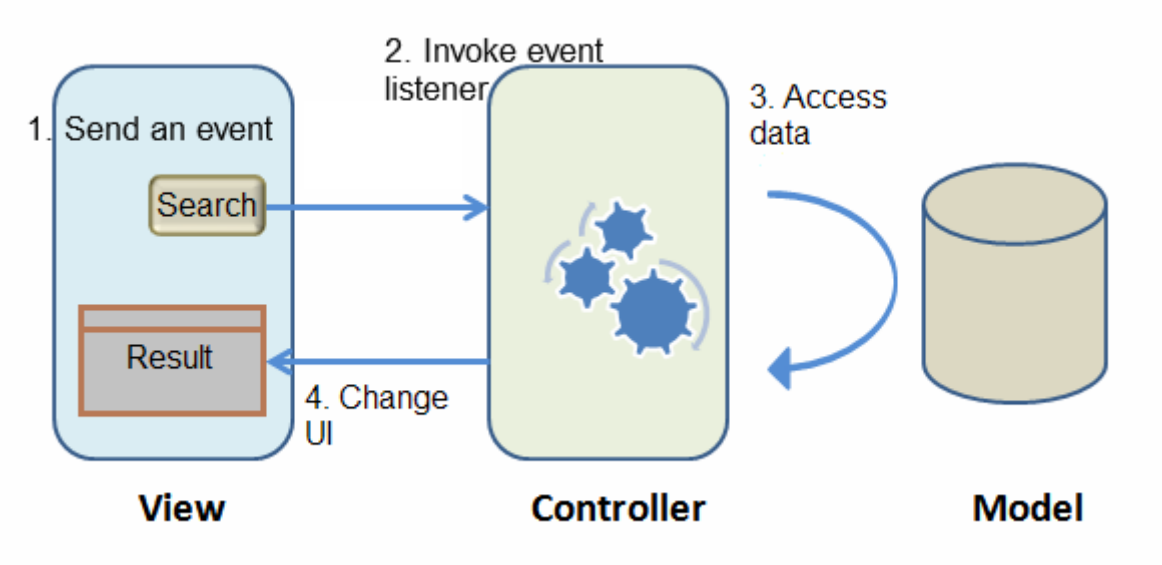


Figura 30: Esquema del funcionamiento del patrón MVC.

4.3.4 Desarrollo

4.3.4.1 Vista (View)

La primera parte de la vista vamos a crear una interfaz para ingresar datos a una lista local que ocuparemos como **Modelo. (Ver Figura 31.)**

Figura 31: Datos a solicitar.

La segunda parte, haremos una búsqueda de los registros de la lista.

Utilizaremos una etiqueta especial llamada <template>, para controlar la representación de cada elemento. ZK hará que el contenido de la etiqueta *Template* sea iterativa para cada objeto en el modelo de datos.

Pasos para utilizar **<template>:**

1. Utilice **<template>** para encerrar los componentes que queremos crear iterativamente.
2. Establezca el atributo "name" de la plantilla a "model".
3. Utilice variable implícita, cada uno, para asignar propiedades de dominio del objeto con los atributos del componente.
4. La etiqueta de template debe ser puesto dentro del listbox.
5. La <listitem> aquí cargaremos la lista de la búsqueda.

6. El **"each"** es una variable que hace referencia a un objeto de dominio en la lista persona de nuestra aplicación de ejemplo. Podemos utilizarlo para acceder a la propiedad de dominio del objeto con EL, por ejemplo, $ {each.fullName}.

```
…
<listbox id="personabox" height="160px" emptyMessage="No hay resultado">
        <listhead>
            <listheader label="Nombre" />
            <listheader label="Sexo" />
            <listheader label="Pais"/>
        </listhead>
        <template name="model">
            <listitem>
                <listcell label="${each.fullName}"></listcell>
                <listcell label="${each.sex}"></listcell>
                <listcell label="${each.pais}"></listcell>

            </listitem>
        </template>
    </listbox>
…
```

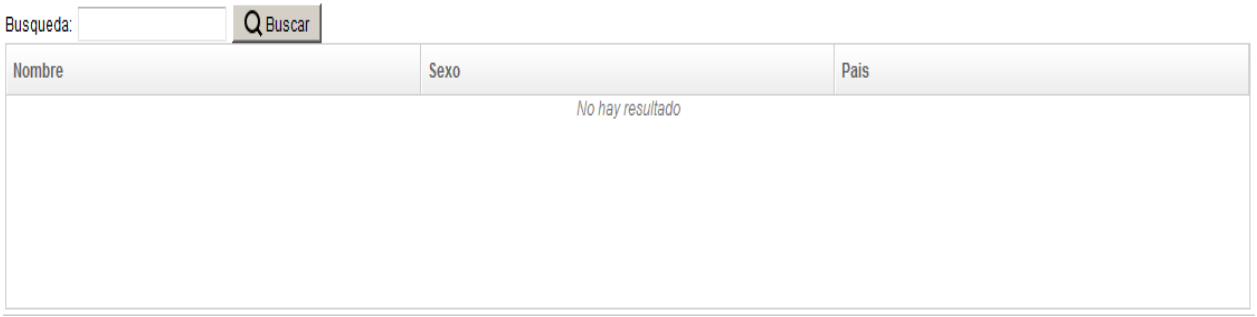


Figura 32: Interfaz de búsqueda.

4.3.4.2 Controller (Controlador)

El **controller** es un programa Java que se utiliza para enlazar la interfaz de usuario (view) con el modelo de datos (model).

Primero creamos una clase controladora llamada "controladormvc.java" en el paquete "controladormvc".

```
package controladormvc;
…
public class controladormvc extends SelectorComposer<Component> {
…
@Listen("onClick = #searchButton")
                public void search(){
                        String keyword = key.getValue().toString();
                        List<persona> result = serv.search(keyword);
                        personabox.setModel(new ListModelList<persona>(result));
                }
…

@Listen("onSelect = #personabox")
                public void showDetail(){
                        // Crear el codigo que muestre los detalles de una persona
                }
…
```

4.3.4.3 Model (Model)

El **model** (modelo) son los datos que maneja una aplicación. Dependiendo de los requisitos de la aplicación, podría ser cualquier cosa, siempre y cuando el controlador lo sabe. Objetos típicos son POJOs, beans, Spring-managed beans y DAO.

4.3.4.4 Crearemos 3 clases dentro del paquete "controladormvc" la clase "personaserviceimple.java", "personaservice.java" y "persona.java"

Interfaces

Las interfaces en java se definen como el conjunto de mensajes que se pueden aplicar a muchas clases a los que cada uno de ellas debe responder de forma adecuada. Por eso, una interfaz recibe también el nombre de protocolo.

Definir una interfaz

Una interfaz consta de 2 partes: el nombre de la interfaz precedida por la palabra reservada **interfaces** y el cuerpo de la interfaz encerrada entre las llaves.

[**public**] **interfaces** nombre_interfaces **extends** superinterfaces

La palabra **extends** significa que se está definiendo una interfaz que es una extensión de otras; también se puede decir que es una interfaz derivada de otras; estas otras se especifican de **extends** separadas por comas.

"personaserviceimple.java"

```
package controladormvc;
…
public class personaserviceimple implements personaservice{

        //modelo de datos.
        public List<persona> persolis= new LinkedList<persona>();

        //inicialización de libro de datos
        public personaserviceimple() {
                persolis.add(
                        new persona("Ana Perez",
                                                "09/12/1992",
```

```
                                                    "Femenino",
                                                    "Peru",
                                                    12345678,
                                                    "ana@gmailcom",
                                                    "Apacionada a la musica contry"));
        …//Seguir la misma estructura con los demás datos persona
}
                        public List<persona> findAll(){
                                return persolis;
                        }
          /*función edita y actualiza un registro en la tabla de datos*/
                        public void updatepersona(persona p){
                         …
                                }
                        /*busca y saca una lista con los resultados*/

                        public List<persona> search(String key){
                                …
                        }
                        /*agrega una registro en la tabla de datos*/
                        public void addPerson(persona per)
                        {
                                …
                        }
                        /*busca un registro en la tabla de datos y devuelve el registro*/
                        public persona searchOne(String fullName){
                                …
                          }
                };
```

"personaservice.java"

```
package controladormvc;
…
public interface personaservice {
                public List<persona> findAll();
                public List<persona> search(String key);
                public void updatepersona(persona p);
                public persona searchOne(String fullName);
                public void addPerson(persona per);
}
```

"persona.java"

```
package controladormvc;

public class persona {

        private String fullName;
        private String birthday;
        private String sex;
        private String pais;
        private Integer tel;
        private String email;
        private String bio;

        public persona(){}

        public persona(String fullName,String birthday, String sex, String pais, Integer
tel,String email, String bio){

        this.fullName=fullName;
```

```
            this.birthday=birthday;
            this.sex=sex;
            this.pais=pais;
            this.tel=tel;
            this.email=email;
            this.bio=bio;
            }
            public String getfullName() {
                        return fullName;
            }
            public void setfullName(String fullName) {
                        this.fullName= fullName;
            }
…
```

Como resultado nos quedara de la siguiente manera **Figura 33.**

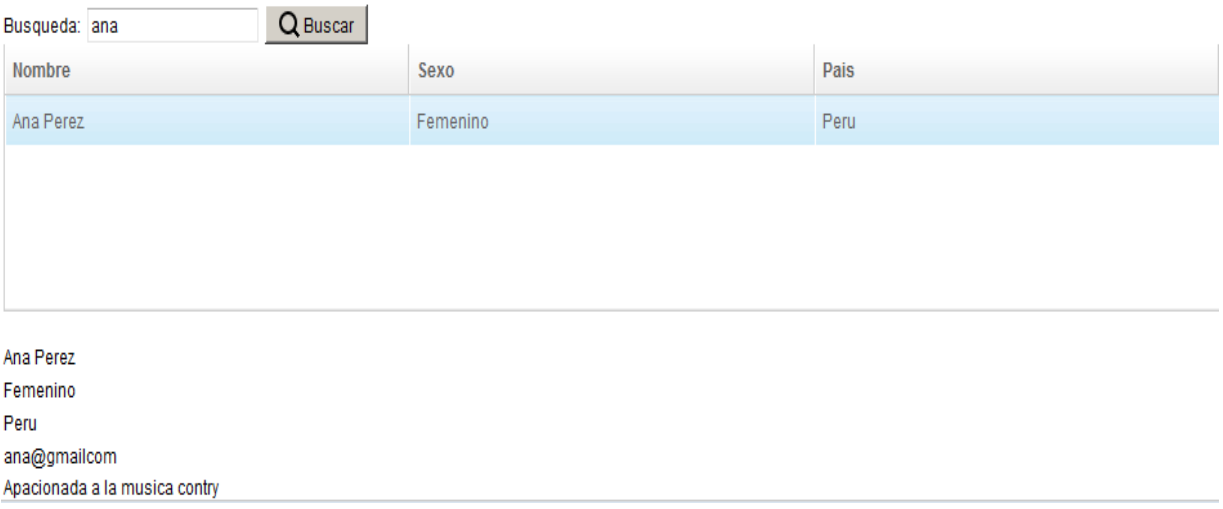


Figura 33: Resultado de la búsqueda.

4.3.5 Ejercicios propuestos

a. Hacer un proyecto llamado **"Equipos de Fútbol"** donde pongamos en práctica el patrón MVC y contenga las siguientes funciones.

1. Ingresar partidos.
2. Ver Partidos.

b. Integrar los proyecto a uno solo donde la navegación sea por página.

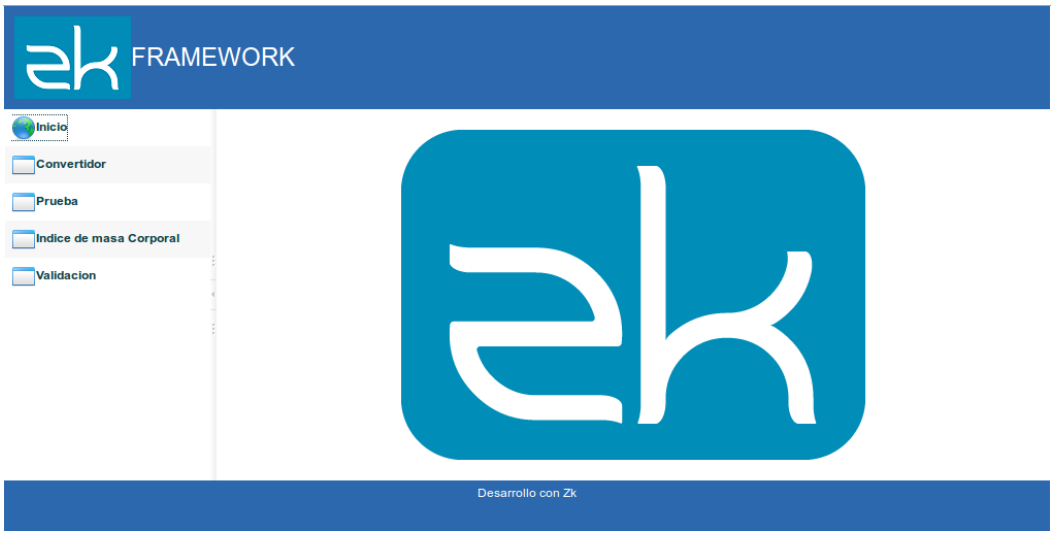


Figura 34: Presentación del ejercicio.

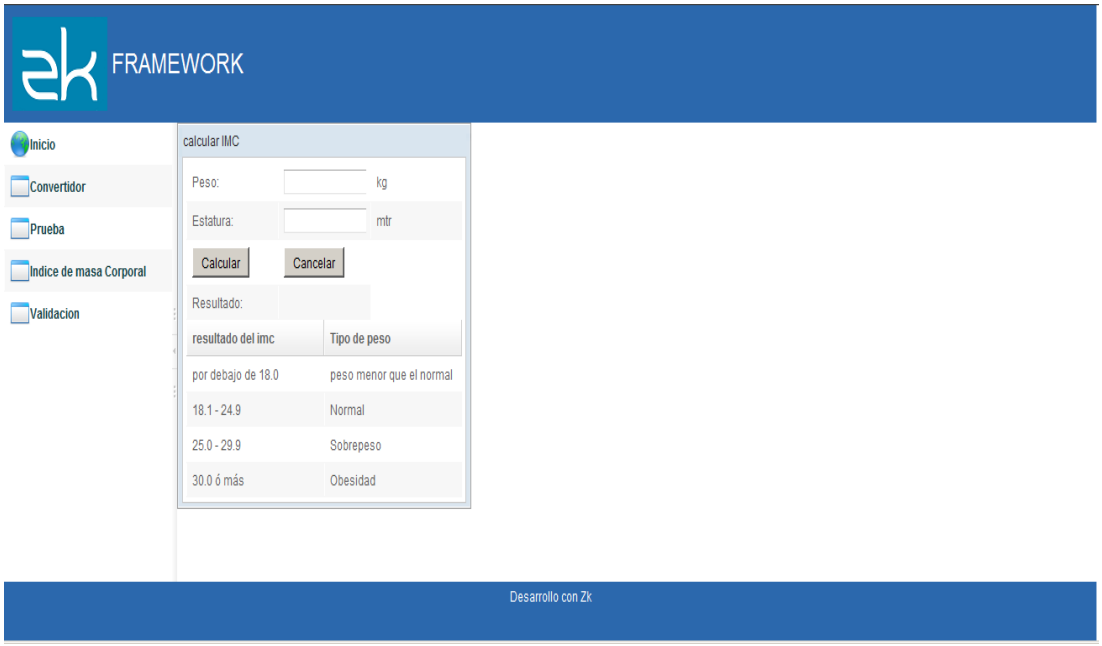


Figura 35: Calculador IMC.

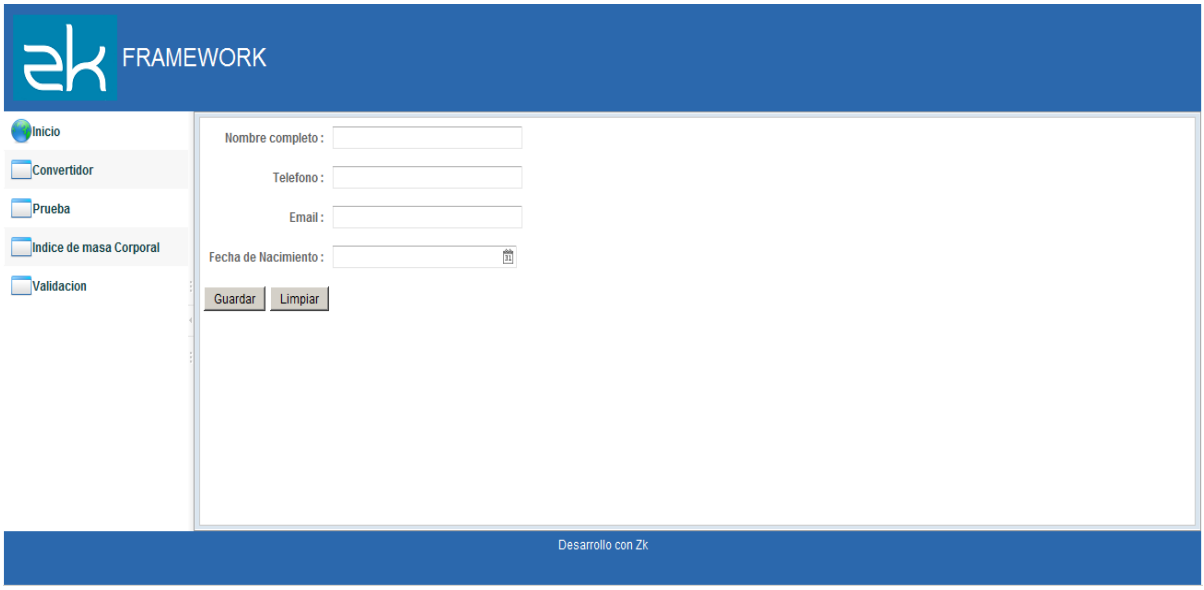


Figura 36: Aplicación de Validación.

4.3.6 Referencia

- ✓ Curso de programación, Java 2, 4ta edición – Fco. Javier Ceballos.
- ✓ http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVC

- ✓ http://books.zkoss.org/wiki/ZK_Getting_Started/Get_ZK_Up_and_Running_with_MVC
- ✓ http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVC/View/Template
- ✓ http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/MVC/View/Template/Listbox%20Template

Práctica 3

Práctica 3:

Accediendo a Base de datos con Hibernate.

Descripción breve:

En esta práctica implementaremos CRUD (Create Read Update Delete) con una Base de Datos y pondremos en práctica el patrón MVC.

4.4.1 Objetivo:

En esta práctica vamos a construir una aplicación para administrar una base de datos con 4 operaciones básicas, crear (Create), Leer (Read), actualizar (Update), y borrar (Delete) (CRUD por sus iniciales en inglés). Preparar a los estudiantes para desarrollar con el patrón MVC de una forma sencilla.

4.4.2 Duración:

4 horas.

4.4.3 Introducción

Debido al paradigma de combinación Objeto/Relacional, los desarrolladores tienden a usar el framework de Mapeo Objeto/Relacional (ORM por sus siglas en ingles) para convertir el modelo orientado a objetos en modelo relacional y vice-versa. **Hibernate** es el framework de ORM más popular en el mundo java.

Hibernate es una herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

Como todas las herramientas de su tipo, Hibernate busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional). Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información Hibernate le permite a la aplicación manipular los datos en la base de datos operando sobre objetos, con todas las características de la POO. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL. Hibernate genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

Hibernate ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo, una API para construir las consultas programáticamente (conocida como "criteria").

Hibernate para Java puede ser utilizado en aplicaciones Java independientes o en aplicaciones Java EE, mediante el componente Hibernate Annotations que implementa el estándar JPA, que es parte de esta plataforma.

4.4.4 Desarrollo

Diseñaremos una pequeña base de datos donde tendremos información de equipos de cómputo.
La interfaz de usuario es la siguiente **Figura 38.**

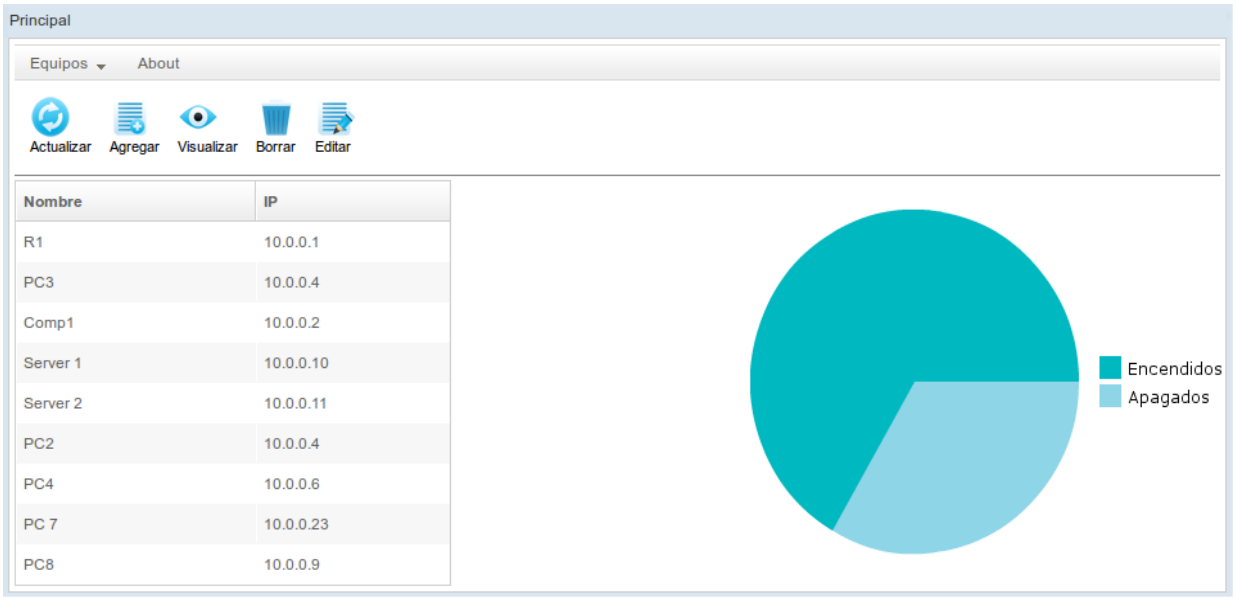


Figura 37: Muestra de la aplicación en ejecución.

Este es un sistema de administración de equipos que tiene las siguientes características:

a. Listar todo los elementos

b. Crear un elemento

c. Modificar un elemento

d. Borrar un elemento

4.4.4.1 Configuración de Hibernate

Para hacer que Hibernate trabaje con ZK, primero debemos copiar todos los archivos .jar a la carpeta **/WebContent/lib**. **Figura 39.**

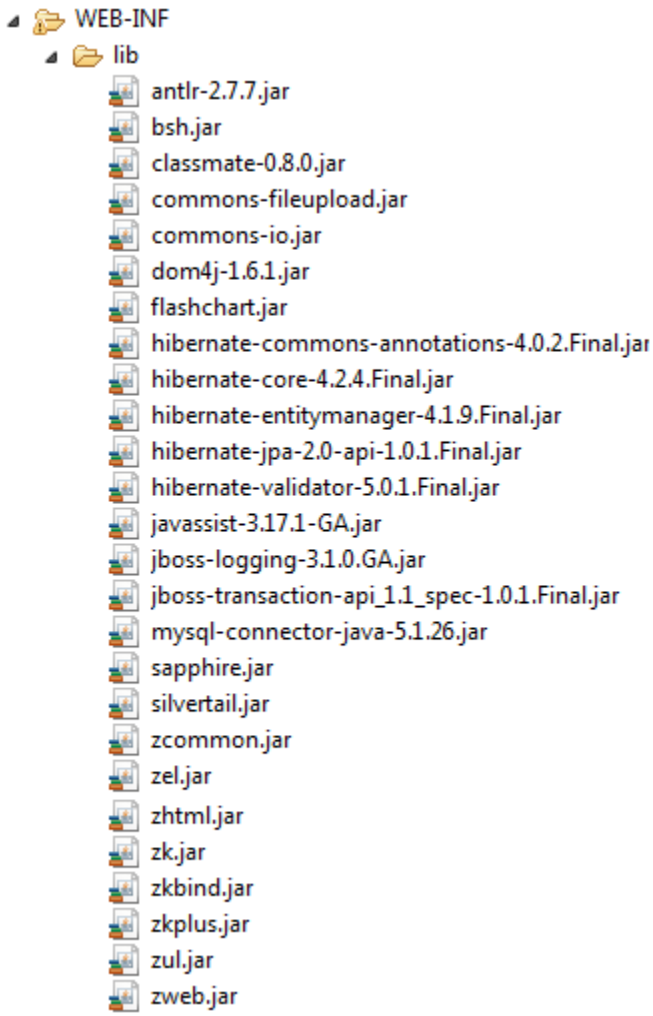


Figura 38: Archivos jar de hibernate.

Luego creamos el archivo de configuración **hibernate.cfg.xml** en la carpeta **javaSource/src**.

```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE hibernate-configuration PUBLIC
     "-//Hibernate/Hibernate Configuration DTD//EN"
     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
    <hibernate-configuration>
        <session-factory>
                <!-- Database connection settings -->
                 <property
    name="connection.driver_class">com.mysql.jdbc.Driver</property>
          <property
    name="connection.url">jdbc:mysql://localhost/equiposManager</property>
            <!-- SQL dialect -->
          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
            <!-- User / Password -->
          <property name="connection.username">root</property>
          <property name="connection.password"></property>
          <!-- JDBC connection pool (use the built-in) -->
          <property name="connection.pool_size">1</property>
          <!-- Enable Hibernate's automatic session context management -->
          <property name="current_session_context_class">thread</property>
          <!-- Disable the second-level cache  -->
          <property
    name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

          <!-- Echo all executed SQL to stdout -->
          <property name="show_sql">true</property>

                <mapping class="CRUD.Pojo.equipos" />
        </session-factory>
    </hibernate-configuration>
```

Copiamos las siguientes líneas en el archivo **zk.xml** que se encuentra en la dirección **/WebContent/WEB-INF**.

```
<listener>
 <listener-
class>org.zkoss.reference.developer.hibernate.web.OpenSessionInViewListener</listener-
class>
   </listener>
    <listener>
     <description>Hibernate SessionFactory life cycle</description>
  <listener-class>org.zkoss.zkplus.hibernate.HibernateSessionFactoryListener</listener-
class>
   </listener>
```

4.4.4.2 Configurando clases necesarias

En este apartado vamos a configurar las clases necesarias para hacer la conexión con la base de datos, posteriormente crearemos las funciones agregar, leer, modificar y borrar datos.

4.4.4.3 Creando el objeto Java y sus anotaciones

Hay dos maneras de decir a Hibernate como cargar y almacenar objetos de la clase persistente, una es usando anotación java, y la otra manera antigua es usando un archivo de mapeo. Nuestra aplicación usara la manera de anotación java. El código que se muestra a continuación es de la clase **equipos.java** que se encuentra en la carpeta **/src/CRUD/Pojo**. La clase base es la que se va encargar de enlazar los atributos de la tabla con nuestra aplicación.

```
import javax.persistence.*;
@Entity
@Table(name="equipos")
public class equipos {
    @Id
    @Column(name="ID")
    @GeneratedValue
```

```
    private Integer id;
@Column(name="nombre")
private String nombre;
…
```

4.4.4.4 Creación de los objetos DAO

El diseño Data Access Object (DAO) es una buena práctica para implementar capa persistente. Este diseño encapsula códigos de acceso a datos escritos por la API de Hibernate de la capa de negocio. Un objeto DAO expone una interfaz de objeto de negocio y realiza operaciones de persistencia. Creamos el archivo **DAO.java** en la carpeta **javaResource/src/CRUD/DAO**.

```
public interface DAO {
    public void add(equipos eq);
    public void delete(equipos eq);
    public void update(equipos eq);
    public List<equipos> listEquipos();
    public List<equipos> search(String n);
}
```

Luego creamos la clase **DAOImpl.java** en la carpeta **/src/CRUD/DAO**. Esta clase contiene las definiciones de cada uno de los métodos de la **interfaz DAO.java**.

```
public class DAOImpl implements DAO {
    private Session sf;

    public void add(equipos eq) {
        //Crear la function para agregar registros.
    }

    @Override
    public void delete(equipos eq) {
```

```
		// TODO Auto-generated method stub
	}

	@Override
	public void update(equipos eq) {
		// TODO Auto-generated method stub
	}

	@Override
	public List<equipos> listEquipos() {
		sf = HibernateUtil.getSessionFactory().getCurrentSession();
		Transaction tx = null;
		List<equipos> le = null;
		try{
			tx = sf.beginTransaction();
			le = sf.createQuery("from equipos").list();
			tx.commit();
		}catch(Exception ex){
			tx.rollback();
		}
		return le;
	}

	@Override
	public List<equipos> search(String n) {
		sf = HibernateUtil.getSessionFactory().getCurrentSession();
		Transaction tx = null;
		List<equipos> listEq = null;
		try {
			tx = sf.beginTransaction();
			listEq = sf.createQuery("select t from equipos t where t.nombre =
```

```
?").setString(0, n).list();
                    tx.commit();
                }catch (Exception ex) {
                    Messagebox.show(ex.getMessage());
                    tx.rollback();
                }
                return listEq;
            }
        }
```

4.4.4.5 Creando la clase HibernateUtil

Una manera simple de implementar una DAO es controlar las sesiones y transacciones de Hibernate manualmente, de ahí necesitamos utilizar una clase para obtener la SessionFactory.

```
public class HibernateUtil {
    private static SessionFactory sessionFactory;
    static {
     try {
        sessionFactory = new Configuration().configure().buildSessionFactory();
     } catch (Throwable ex) {
        throw new ExceptionInInitializerError(ex);
     }
    }

    public static SessionFactory getSessionFactory() {
      return sessionFactory;
    }
}
```

4.4.4.6 Implementación CRUD

4.4.4.6.1 Leer (Read)

Como hablamos en el capítulo anterior, podemos usar *Template para* definir como mostrar un modelo lista de datos con la variable implícita *each*.

```
<listbox id="lista" emptyMessage="No se encontro registros!!" style="clear:both;"
width="60%" mold="default">
    <listhead>
        <listheader label="Nombre" sort="auto" />
        <listheader label="IP" />
        <listheader label="Estado" />
    </listhead>
    <template name="model">
        <listitem>
            <listcell label="${each.nombre}" />
            <listcell label="${each.ip}" />
            <listcell label="${each.estado}" />
        </listitem>
    </template>
</listbox>
```

En el controlador, deberíamos proveer un modelo de datos para el *Listbox*.

```
@Wire
public Listbox lista;
public void recargar(){
    dao = new DAOImpl();
    ListModelList<equipos> le = new ListModelList<equipos>(dao.listEquipos());
    lista.setModel(le);
}
```

4.4.4.6.2 Crear (Create)

Para crear nuevos elementos y guardar en la base de datos se ha utilizado la siguiente interfaz **Figura 40.**

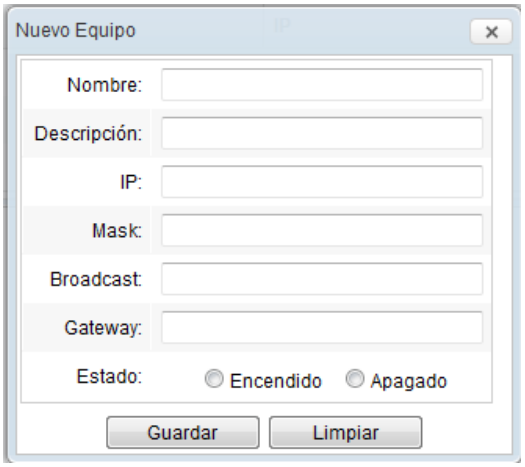


Figura 39: Interfaz para agregar equipos a la base de datos.

El evento clic del botón guardar quedaría como se muestra a continuación:

```
@Listen("onClick = #btGuardar")
public void guardar(){
// Completar el código de guardar
}
```

4.4.4.6.3 Actualizar (Update)

Para actualizar un elemento, deberíamos primero seleccionarlo en la lista mostrada al inicio de la página y dar clic en *Editar*. **Figura 41-42**.

Figura 40: Editar elemento.

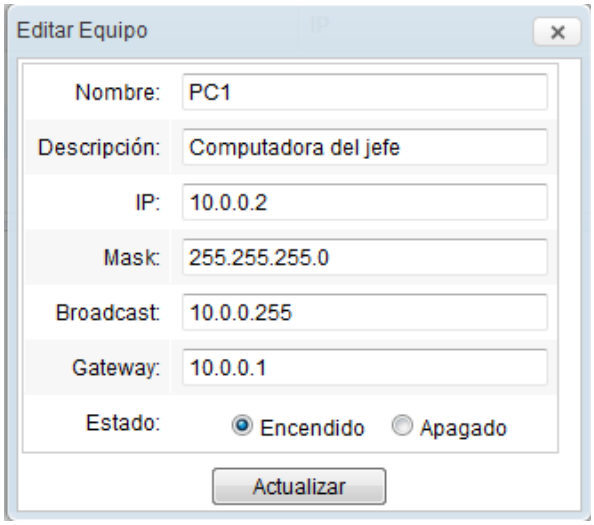


Figura 41: Actualizar elemento.

```
@Listen("onClick = #tbEditar")
public void editar(Event e) {
    // Completar el código de ediar
}
```

4.4.4.6.4 Borrar (Delete)

Implementar la función Borrar es similar al botón editar, ya q se obtiene el elemento seleccionado y precedemos a borrarlo.

```
@Listen("onClick = #tbBorrar")
public void borrar(Event e) {
    // Completar el código de borrar
}
```

4.4.5 Ejercicios propuestos

Desarrollar una aplicación para gestionar el control de productos en un almacén implementando CRUD y el patrón MVC. Los elementos de la base de datos serian: código del producto, nombre, precio y existencias.

4.4.6 Referencia

- ✓ Curso de programación, Java 2, 4ta edición – Fco. Javier Ceballos.
- ✓ http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/Integration/Persistence%20Layer/Hibernate
- ✓ http://docs.jboss.org/hibernate/core/3.6/reference/es-ES/html_single/
- ✓ http://www.zkoss.org/zksandbox/#l7
- ✓ http://www.zkoss.org/zksandbox/#t1
- ✓ http://www.zkoss.org/zksandbox/#t2
- ✓ http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVC
- ✓ http://books.zkoss.org/wiki/ZK_Getting_Started/Get_ZK_Up_and_Running_with_MVC
- ✓ http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVC/View/Template
- ✓ http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/MVC/View/Template/Listbox%20Template

Práctica 4

Práctica 4:

Acceso a varias tablas de Bases de Datos con Hibernate.

Descripción breve:

En esta práctica aprenderemos como administrar una base de datos con varias tablas relacionadas.

4.5.1 Objetivo:

Aprender con una pequeña aplicación que mostrara el uso de una base de datos con varias tablas relacionadas, aplicando Hibernate con ZKoss.

4.5.2 Duración:

6 horas.

4.5.3 Introducción

Se construirá una aplicación que implementa base de datos con Hibernate, y tiene varias tablas relacionadas. La idea es aprender a trabajar con las relaciones de las tablas debido a que su funcionamiento es muy diferente al de una base de datos con una sola tabla. A continuación, se muestra la interfaz que se utilizara en dicha aplicación. **Figura 42**

Figura 42: Aplicación en ejecución.

Este es un sistema de comercio electrónico que permite:

1) Agregar, editar y eliminar clientes.
2) Agregar, editar y eliminar vendedores.
3) Agregar, editar y eliminar productos.
4) Y agregar facturas por compras.

4.5.4 Desarrollo

4.5.4.1 Configuración de Hibernate

Para hacer que Hibernate trabaje con ZK, primero debemos copiar todos los archivos **.jar** de Hibernate a la carpeta **/WebContent/lib**.

Luego creamos el archivo de configuración **hibernate.cfg.xml** con el siguiente código en la carpeta **javaSource/src**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
        <session-factory>
                <!-- Database connection settings -->
    <!-- Se pasa la propiedad del driver de conexión. -->
 <property name="connection.driver_class">com.mysql.jdbc.Driver</property>

   <!-- Se pasa la dirección de la base de datos a la cual se conectara. -->
 <property name="connection.url">jdbc:mysql://localhost/Cuentas</property>

    <!-- SQL dialect -->
 <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

  <!-- User / Password -->
```

```
  <!-- Se especifica el usuario y contraseña de nuestro gestor de base de datos.
--->
      <property name="connection.username">root</property>
      <property name="connection.password"></property>
       <!-- JDBC connection pool (use the built-in) -->
      <property name="connection.pool_size">3</property>

       <!-- Enable Hibernate's automatic session context management -->
       <property name="current_session_context_class">thread</property>

       <!-- Disable the second-level cache  -->
    <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

       <!-- Echo all executed SQL to stdout -->
       <property name="show_sql">true</property>

<!-- Indicamos los objetos java con los que esta mapeada cada tabla de la base
de datos. --->
                    <mapping class="org.cuentas.POJO.clientes" />
                    <mapping class="org.cuentas.POJO.vendedor" />
                    <mapping class="org.cuentas.POJO.productos" />
                    <mapping class="org.cuentas.POJO.detalleFactura" />
                    <mapping class="org.cuentas.POJO.factura" />
          </session-factory>
 </hibernate-configuration>
```

Copiamos las siguientes líneas en el archivo **zk.xml** que se encuentra en la dirección **/WebContent/WEB-INF**.

```
<listener>
```

```
    <listener-class>
org.zkoss.reference.developer.hibernate.web.OpenSessionInViewListener
        </listener-class>
</listener>
  <listener>
    <description>Hibernate SessionFactory life cycle</description>
    <listener-class>
org.zkoss.zkplus.hibernate.HibernateSessionFactoryListener
        </listener-class>
</listener>
```

4.5.4.2 Configurando Clases necesarias

A continuación configuraremos las clases necesarias para lograr la conexión entre la base de datos y nuestra aplicación.

4.5.4.3 Creando los objetos java y sus anotaciones

Crearemos las clases correspondientes para enlazar las tablas de la base de datos con nuestra aplicación, esto se logra con anotaciones como se muestra a continuación:

clientes.java ubicada en la carpeta **javaResource/src/org/cuentas/POJO**

```
import javax.persistence.*;

@Entity
@Table(name="tbCliente")
public class clientes {
        @Id
        @Column(name="ID")
        @GeneratedValue
        private int id;
 @Column(name="cedulaCliente")
        private String cedulaCliente;
```

```
@OneToMany(mappedBy="tbCliente")
        private List<factura> lsFacturas;
…
```

Importamos el paquete necesario para poder dar uso a las anotaciones.

@Entity: Se usa para declarar la clase como objeto persistente.

@Table: Enlazamos la clase con la tabla "tbCliente" de la base de datos.

@Id, @GeneratedValue: Indica que el campo ID de la base de datos es autoincrementado y se enlaza con la variable id.

@Column: Es usada para mapear la entidad con la columna "cedulaCliente" de la base de datos.

@OneToMany(mappedBy="tbCliente"): Indica una relación de uno a muchos donde el objeto es una colección de tipo *factura.*

A continuación se muestra parte del código de la clase **factura.java** ubicada en la carpeta **javaResource/src/org/cuentas/POJO.**

```
import javax.persistence.*;

@Entity
@Table(name="tbFactura")
public class factura {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
private double totalFactura;
private String fechaFactura;


@ManyToOne
@JoinColumn(name="tbVendedor_id")
private vendedor tbVendedor;
@ManyToOne
```

```
@JoinColumn(name="tbCliente_id")
private clientes tbCliente;


@OneToMany(mappedBy="tbFactura", fetch=FetchType.EAGER)
    private List<detalleFactura> lstDetalle;
```

@Entity: Se declara que la clase será un objeto persistente.

@Table(name="tbFactura"): Mapeamos la clase con la tabla *tbFactura.*

@GeneratedValue(strategy=GenerationType.*IDENTITY*): Indica que el campo id es auto generado y se mapea con la variable id de este objeto.

@ManyToOne @JoinColumn(name="tbVendedor_id"): Indica una relación de muchos a uno con la tabla vendedor.

@OneToMany(mappedBy="tbFactura", fetch=FetchType.*EAGER*): Indica una relación de uno a muchos que será almacenado en una colección de tipo *detalleFactura.*

A continuación se muestra parte del código de la clase **detalleFactura.java** ubicado en la carpeta **javaResource/src/org/cuentas/POJO.**

```
import javax.persistence.*;

@Entity
@Table(name="tbDetalleFactura")
public class detalleFactura {
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        private int id;
        private int cantidadProducto;
        private double precioVenta;
        private double totalProducto;
        @ManyToOne
        @JoinColumn(name="tbFactura_id")
        private factura tbFactura;
```

```
        @ManyToOne
        @JoinColumn(name="tbProductos_id")
        private productos tbProductos;
..
```

@Table(name="tbDetalleFactura"): Indica que la tabla *tbDetalleFactura* sera mapeada por el objeto *detalleFacctura.*

@GeneratedValue(strategy=GenerationType.IDENTITY): Indica que el campo id es autogenerado.

@ManyToOne @JoinColumn(name="tbFactura_id"): Indica una relación de muchos a uno con la tabla factura y que tiene como llave foránea *tbFactura_id.*

@ManyToOne @JoinColumn(name="tbProductos_id"): Indica una relación de muchos a uno con la tabla productos y q tiene como clave foránea *tbProductos_id.*

A continuación se muestra parte del código de la clase **vendedor.java** ubicada en la carpeta **javaResource/src/org/cuentas/POJO.**

```
@Entity
@Table(name="tbVendedor")
public class vendedor {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
private String inssVendedor;
private String nombreVendedor;
private String apellidoVendedor;

@OneToMany(mappedBy="tbVendedor")
private List<factura> lsFactura;
```

@Entity: La tabla *tbVendedor* es mapeada con el objeto *vendedor.*

@Id @GeneratedValue(strategy=GenerationType.*IDENTITY*): Indica que el campo id es autogenerado.

@OneToMany(mappedBy="tbVendedor"): Indica una relación de uno a muchos con la tabla "tbVendedor".

***Nota*:** Se omitió parte del código debido al tamaño del mismo.

4.5.4.4 Creación de los objetos DAO

El diseño Data Access Object (DAO) es una buena práctica para implementar capa persistente. Este diseño encapsula códigos de acceso a datos escritos por la API de Hibernate de la capa de negocio. Un objeto DAO expone una interfaz de objeto de negocio y realiza operaciones de persistencia. Creamos el archivo **clienteService.java** en la carpeta **javaResource/src/org/cuentas/DAO.**

```
public interface clienteService {
        public void add(clientes cl);
        public void remove(clientes cl);
        public void update(clientes cl);
        public List<clientes> findAll();
}
```

A continuación creamos las clases de implementación, **clienteServiceImpl** ubicada en la carpeta **javaResource/src/org/cuentas/DAO.** Esta clase contiene la definicion de cada uno de los métodos de la **interfaz clienteService.java.**

```
private Session sessionFactory =
HibernateUtil.getSessionFactory().getCurrentSession();

@Override
public void add(clientes cl) {
        // TODO Auto-generated method stub
}
@Override
public void remove(clientes cl) {
```

```
            // TODO Auto-generated method stub
        }
        @Override
        public void update(clientes cl) {
                    // TODO Auto-generated method stub
            }
        @Override
        public List<clientes> findAll() {
            //Completsr el codigo.
        }
```

El código anterior: Indica que la clase **clienteServiceImpl** implementa la interfaz **clienteService.** Declara la variable que guardara la sesión actual de Hibernate. Definición de la función ADD que es para agregar registros a la base de datos .Definición de la función REMOVE que se utiliza para borrar registros de la base de datos. Define la función UPDATE que usamos para actualizar registros de datos. Función q retorna una lista de todos los registros de la tabla clientes..

4.5.4.5 Creando la clase HibernateUtil

Una manera simple de implementar una DAO es controlar las sesiones y transacciones de Hibernate manualmente para lograr esto debemos utilizar una clase para obtener la Session-factory. A continuación el codigo:

```
public class HibernateUtil {

        private static SessionFactory sessionFactory;
        static {
    try {
        sessionFactory = new Configuration().configure().buildSessionFactory();
    } catch (Throwable ex) {
        throw new ExceptionInInitializerError(ex);
    }
  }
```

```
        public static SessionFactory getSessionFactory() {
            return sessionFactory;
        }
    }
```

4.5.4.6 Enfoque MVC

El diseño de esta aplicación es bastante sencillo, ésta se divide con layaout en norte, sur, este y central. En la sección norte se presenta un encabezado con un logo. En la sección sur se presenta un pequeño pie de página. En la sección este una lista de enlaces a las diferentes páginas de nuestra aplicación. Y en la zona central, que será el contenedor, se mostraran las diferentes páginas con el mismo aspecto que la página principal. Ver **Figura 43**.

4.5.4.6.1 Lectura (Read)

En el capítulo anterior se hizo uso de *Template* para mostrar un modelo de lista de datos con la variable implicita *Each.*

```
<listbox id="lsCliente" emptyMessage="No se encontro registro!!" mold="paging"
pageSize="4">
        <listhead>
            <listheader label="Nombre" />
            <listheader label="Apellido" />
            <listheader label="Direccion" />
            <listheader label="Telefono" />
        </listhead>
        <template name="model">
        <listitem>
            <listcell label="${each.nombreCliente}" />
            <listcell label="${each.apellidoCliente}" />
            <listcell label="${each.direccionCliente}" />
            <listcell label="${each.telefonoCliente}" />
        </listitem>
        </template>
```

```
</listbox>
```

Se define el modelo para la lista de datos. Se pasa cada dato con la variable implícita *each.*

A continuación se presenta la interfaz que muestra la lista de los clientes registrados:

4.5.4.6.2 Agregar (Create)

Para agregar nuevos registros de facturas a la base de datos, se usó la interfaz siguiente:

Figura 43: Interfaz para agregar usuarios.

En esta interfaz se observa que se puede seleccionar cliente, vendedor y productos de listas obtenidas de la base de datos. También se muestra un atributo fecha y otro que indica la cantidad del producto a comprar. Luego se guarda la lista previamente creada en la base de datos.

4.5.4.6.3 Actualizar (Update)

Para actualizar un registro, primero se debe seleccionar un elemento de la lista mostrada al inicio de la página y dar clic en el botón editar. Luego se mostrará una ventana o pestaña donde se cargan los datos del elemento seleccionado de la lista para ser actualizados.

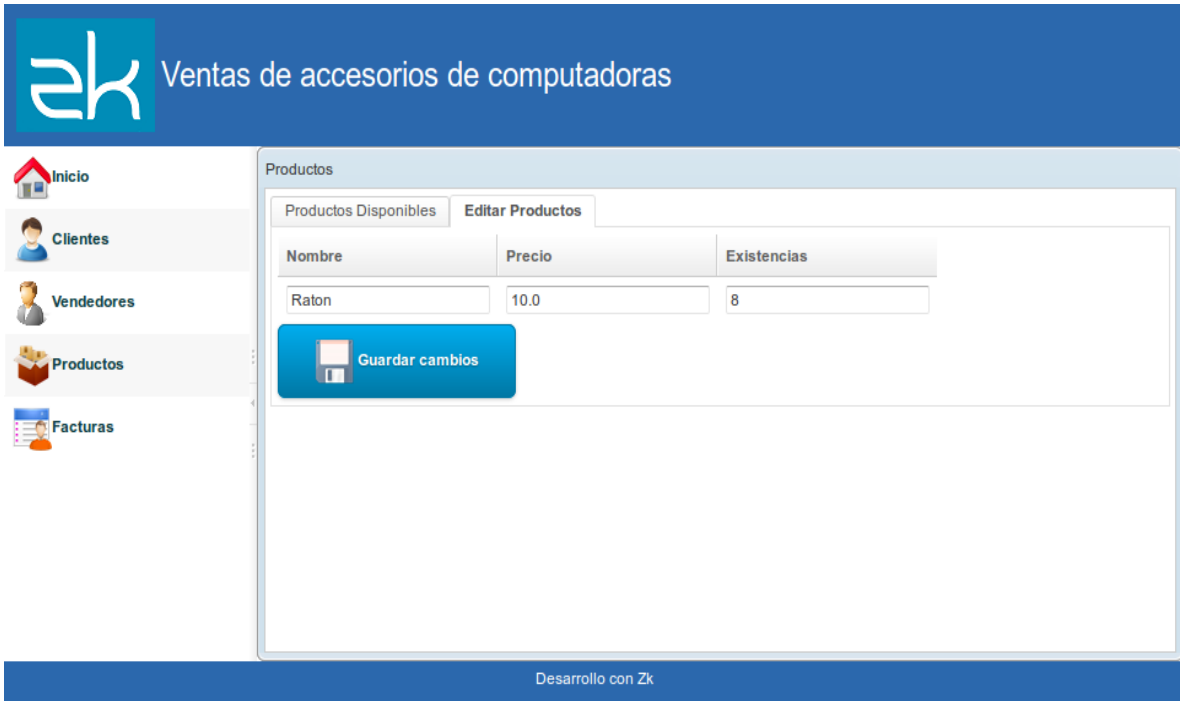


Figura 44: Interfaz para editar.

4.5.4.6.4 Borrar (Delete)

Implementar la función Borrar es similar al botón editar, ya que se obtiene el elemento seleccionado y procedemos a borrarlo.

4.5.5 Ejercicios propuestos

Terminar de hacer todas las funcionalidades que no se explicaron en esta práctica para que el proyecto quede completo.

4.5.6 Referencia

- ✓ http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/Integration/Persistence%20Layer/Hibernate
- ✓ http://www.zkoss.org/zksandbox/#g6

Práctica 5

Práctica 5:

Autenticación

Descripción breve:

En esta práctica implementaremos autenticación para evitar accesos no autorizados a nuestras páginas.

4.6.1 Objetivo:

En este capítulo, vamos a demostrar cómo implementar la autenticación y proteger sus páginas de acceso ilegal.

4.6.2 Duración:

2 horas.

4.6.3 Introducción

Una aplicación web funciona sobre el protocolo HTTP que es apátrida; cada solicitud y su respuesta correspondiente se manejan independientemente. Por lo tanto, un servidor HTTP no puede saber si una serie de peticiones es enviada desde el cliente mismo o de diferentes clientes. Eso significa que el servidor no puede mantener el estado del cliente entre varias solicitudes.

Los Servidores de Aplicaciones conservan una *sesión* para mantener el estado del cliente. Cuando el servidor recibe la primera solicitud de un cliente, el servidor crea una sesión y le da un identificador único. El cliente debe enviar una solicitud con el identificador de sesión. El servidor puede determinar cuál sesión pertenece la solicitud.

En un entorno Java EE, un servidor de aplicación crea un objeto *javax.servlet.http.HttpSession* para realizar el seguimiento de sesión del cliente. **Session** de ZK es un contenedor de *HttpSession*, lo puede utilizar para almacenar datos del usuario cuando manejas eventos. El uso:

- Obtener sesion actual: *Sessions.getCurrent()*
- Almacenar datos de una sesion: *Session.setAttribute("key", data)*
- Recuperar datos de una sesion: *Session.getAttribute('key')*

Casi todas las aplicaciones de negocio requieren de un mecanismo de seguridad y la autenticación es la parte fundamental de la misma. Algunos recursos sólo están disponibles para los usuarios autenticados. La autenticación es un proceso para verificar que el usuario es quien dice ser. Después de autenticar un usuario, la aplicación tiene que recordar al usuario e identifica sus solicitudes posteriores de modo que la aplicación no necesita autenticarse repetidamente para cada

nueva solicitud. Por lo tanto, el almacenamiento de credenciales de usuario en una sesión es una buena práctica y podemos decirle a quien pertenece la solicitud de petición a la sesión. Además, no podemos decir si llega una petición de un usuario autenticado mediante la comprobación de las credenciales de un usuario en la solicitud de sesión.

4.6.4 Desarrollo

4.6.4.1 Asegure sus páginas

Incluso si ha configurado un mecanismo de autenticación, un usuario puede acceder a una página si conoce la dirección URL de la página. Por lo tanto, debemos proteger una página de acceso ilegal mediante la comprobación de las credenciales del usuario en su sesión cuando una página es solicitada por un usuario.

Crearemos una página de inicio de sesión sin una barra lateral de la siguiente manera:

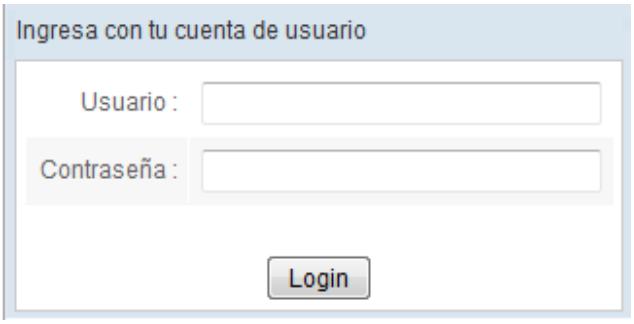


Figura 45: Autenticación de página.

Después de inicio de sesión, podemos redirigir a los usuarios a la página índice y mostrar el nombre de usuario en el lado derecho de la cabecera.

Figura 46: Página inicial.

4.6.4.2 Servicio de autenticación

Podemos implementar una clase de servicio que realiza las operaciones de autenticación.

4.6.4.2.1 Get user credentials

```
public class AuthenticationServiceImpl implements AuthenticationService,Serializable{

public UserCredential getUserCredential(){
        Session sess = Sessions.getCurrent();
        UserCredential cre =
        (UserCredential)sess.getAttribute("userCredential");
        if(cre==null){
            cre = new UserCredential();
            sess.setAttribute("userCredential",cre);
        }
        return cre;
```

```
        }
```

4.6.4.2.2 Log in / Log out

```
public class AuthenticationServiceImpl implements AuthenticationService,Serializable{
    UserInfoService userInfoService = new UserInfoServiceImpl();
@Override
    public boolean login(String nm, String pd) {
        User user = userInfoService.findUser(nm);
        //Verificacion de la contraseña
        if(user==null || !user.getPassword().equalsIgnoreCase(pd)){
            return false;
        }
        Session sess = Sessions.getCurrent();
        UserCredential cre = new serCredential(user.getAccount(),user.getFullName());

        if(cre.isAnonymous()){
            return false;
        }
        sess.setAttribute("userCredential",cre);

        return true;
    }
    @Override
    public void logout() {
        Session sess = Sessions.getCurrent();
        sess.removeAttribute("userCredential");
    }
}
```

4.6.4.3 Inicialización de la página

ZK permite inicializar una página zul mediante la implementación de un **iniciador**. Cuando aplicamos un iniciador a un zul, ZK lo utilizará para realizar operaciones de inicialización antes de crear los componentes.

Podemos crear un iniciador para comprobar la existencia de las credenciales de un usuario en la sesión. Si las credenciales de un usuario están ausente, determinamos que es una solicitud ilegal y se re direcciona a la página de inicio de sesión.

4.6.4.3.1 Iniciador de la página para comprobar las credenciales de un usuario

```
public class AuthenticationInit implements Initiator {

        //servicios
        AuthenticationService authService = new AuthenticationServiceImpl();

        public void doInit(Page page, Map<String, Object> args) throws Exception {

                UserCredential cre = authService.getUserCredential();
                if(cre==null || cre.isAnonymous()){
                        Executions.sendRedirect("/login.zul");
                        return;
                }
        }
}
```

4.6.4.3.2 Index.zul

Index.zul es la página principal, así que aplicamos el iniciador de páginas es esta.

```
<?link rel="stylesheet" type="text/css" href="/estilos.css"?>
<!—Proteger pagina por el iniciador de autenticacion -->
<?init class="org.cuentas.util.AuthenticationInit"?>
```

```
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/plantilla.zul"?>
<zk>
        <include self="@define(contenido)" src="home.zul"/>
</zk>
```

Se debe aplicar el iniciador en cada una de las páginas de nuestro proyecto ya que, al no hacerlo, los usuarios que conocen la URL podrán acceder a la página con facilidad.

4.6.4.3.3 El atributo "if"

Podemos utilizar una expresión EL en el atributo if para determinar la creación de un componente de acuerdo a las credenciales de un usuario en la sesión. Si el resultado de la evaluación de la expresión EL es true, el componente se mostrara de lo contrario no se creará.

4.6.4.3.4 Plantilla.zul

sessionScope es una variable implícita que le permite acceder a los atributos de este período de sesiones.

El *Borderlayout* sólo se crea cuando userCredential no es anónimo, de lo contrario, muestra el *Div* para redirigir.

```
<zk>
        <!-- Se crea solamente cuando el usuario actual no es anonimo -->
    <borderlayout hflex="1" vflex="1" if="${not sessionScope.userCredential.anonymous}">
        ...
        </borderlayout>
    <div if="${sessionScope.userCredential.anonymous}">
                Redirigir a la pagina de login.....
        </div>
</zk>
```

4.6.4.4 Login

Es una forma común de solicitar una cuenta y una contraseña para la autenticación. Creamos una página de inicio de sesión para cobrar la cuenta de usuario y contraseña, y la página de inicio de sesión también utiliza una plantilla zul para mantener un estilo coherente con la página de índice. Sin embargo, no tiene ninguna barra lateral porque los usuarios, sin iniciar sesión, no deberían poder acceder a las funciones principales.

Esta página está hecha para los usuarios anónimos, no tenemos que proteger esta página con el atributo if. El formulario de login se construye con Grid. Esta página debe ser accesible para todos los usuarios, así que no tenemos que aplicar AuthenticationInit.

4.6.4.4.1 template-anonymous.zul

```
<zk>
    <!-- Plantilla de acceso libre, sin barra lateral  -->
    <borderlayout hflex="1" vflex="1">
        <north height="100px" border="none" >
            <include src="/cabecera.zul"/>
        </north>
        <center id="mainContent" autoscroll="true" border="none"
self="@insert(content)">
        </center>
        <south height="50px" border="none">
            <include src="/pie.zul"/>
        </south>
    </borderlayout>
</zk>
```

4.6.4.4.2 Login.zul

```
<?link rel="stylesheet" type="text/css" href="/estilos.css"?>
<!-- it is a login page, no authentication protection and use anonymous template -->
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/template-anonymous.zul"?>
<zk>
<hbox self="@define(content)" vflex="1" hflex="1" align="center"
                pack="center" spacing="20px">
                <vlayout>
                        <window id="loginWin"
                                apply="org.cuentas.controller.LoginController"
                                title="Login with you name" border="normal" hflex="min">
…
</zk>
```

Este controlador de inicio de sesión recoge la cuenta y la contraseña y la valida con una clase de servicio de autenticación. Si la contraseña es correcta, la clase de servicio de autenticación guarda las credenciales del usuario en la sesión.

```
public class LoginController extends SelectorComposer<Component> {

        @Wire
        Textbox account;
        …
AuthenticationService authService = new AuthenticationServiceImpl();

@Listen("onClick=#login; onOK=#loginWin")
public void doLogin(){
        String nm = account.getValue();
        String pd = password.getValue();

if(!authService.login(nm,pd)){
```

```
                message.setValue("account or password are not correct.");
                return;
        }
        UserCredential cre= authService.getUserCredential();
        message.setValue("Welcome, "+cre.getName());
        message.setSclass("");
        Executions.sendRedirect("/");
                }
        }
```

Después de iniciar sesión, queremos mostrar la cuenta de un usuario en el banner. Podemos usar EL para obtener la cuenta de un usuario de UserCredential en la sesión.

4.6.4.4.3 Cabecera.zul

La **sessionScope** es un objeto implícito se puede utilizar dentro del atributo EL para acceder a la sesión. Funciona como el mismo *getAttribute()*. Se puede utilizar para obtener el atributo de sesión con notación de puntos, por ejemplo, $ {sessionScope.userCredential} es igual a llamar getAttribute ("UserCredential") de un objeto *Session*.

```
<div hflex="1" vflex="1" sclass="banner">
  <hbox hflex="1" vflex="1" align="center">

  <!-- Otros componentes -->

 <hbox hflex="1" vflex="1" pack="end" align="end"
     apply="org.cuentas.controller.LoginController">
      <label value="${sessionScope.userCredential.name}"
      if="${notsessionScope.userCredential.anonymous}"/>
<label id="logout" value="Logout" if="${not sessionScope.userCredential.anonymous}"
sclass="logout"/>
    </hbox>
  </hbox>
```

```
</div>
```

4.6.4.4.4 Logout

Cuando un usuario cierra la sesión, por lo general desaparecen sus credenciales y lo dirigimos a la página de inicio de sesión. En nuestro ejemplo, haciendo clic en la etiqueta "logout" en el banner se realice la desconexión.

4.6.4.4.5 Controlador de logout

```
public class LoginController extends SelectorComposer<Component> {

AuthenticationService authService = new AuthenticationServiceImpl();


@Listen("onClick=#logout")
        public void doLogout(){
                authService.logout();
                Executions.sendRedirect("/");
        }
}
```

4.6.5 Ejercicio propuestos

Autenticar el proyecto que se hizo en la práctica N°3, el de control de productos de almacén.

4.6.6 Referencia

- ✓ http://books.zkoss.org/wiki/ZK%20Essentials/Chapter%208:%20Authentication

Práctica 6

Práctica 6:

Poniendo en práctica el patrón MVVM.

Descripción breve:

En esta práctica diseñaremos un pequeño ejercicio donde emplearemos la manipulación de entradas de usuario con controles comunes, integraremos el patrón MVVM (Model View ViewModel) y el uso de Hibérnate con este patrón.

4.7.1 Objetivo:

Preparar a los estudiantes para que aprendan a desarrollar con el patrón MVVM.

4.7.2 Duración:

6 horas.

4.7.3 Introducción

ZK soporta el patrón de diseño Model-View-ViewModel (MVVM) que automatiza las tareas de enlace de datos con los componentes de la vista. Este patrón divide una aplicación en tres partes. EL **View** significa interfaz de usuario. La página de zul que contiene los componentes de ZK. La interacción de un usuario con los componentes desencadena eventos que se enviarán a los controladores.

- El **model** se compone de datos de aplicaciones y reglas de negocio.
- La **view** significa interfaz de usuario. La página zul que contiene los componentes de ZK.
- El **viewmodel** es un tipo de abstracción que contiene el estado y el comportamiento de una vista. Es el encargado de exponer los datos del modelo a la vista y proporciona acciones necesarias solicitadas por la Vista.

Hay un **binder** en el ZK que sincroniza datos entre la VistaModelo y la Vista, y gestiona eventos de forma automática de acuerdo a los datos de las expresiones de enlace.

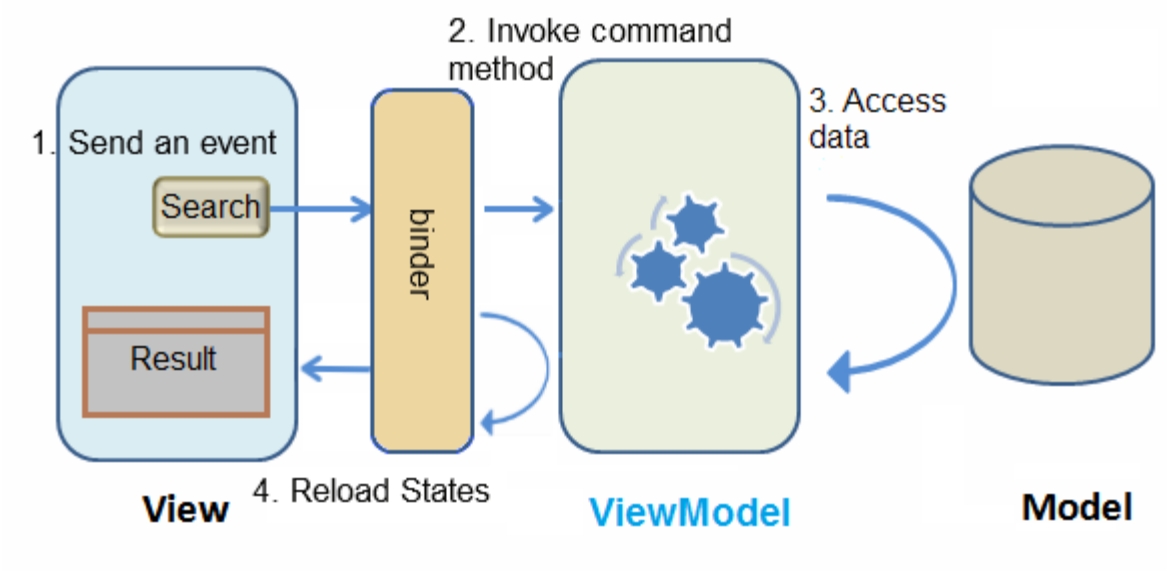


Figura 47: Esquema de funcionamiento del patrón MVVM.

4.7.4 Desarrollo

4.7.4.1 Model (Model)

El modelo (model) son los datos que maneja una aplicación. Dependiendo de los requisitos de aplicación, podría ser cualquier cosa. Entre los objetos típicos están POJOs, beans, Spring-managed beans y DAO.

Aquí crearemos 3 clases dentro del paquete "controladormvvm" la clase "personaservice.java", "personaserviceimple.java" y "persona.java".

4.7.4.1.1 Interfaces

Las interfaces en java se definen como el conjunto de mensajes que se pueden aplicar a muchas clases, a los que cada uno de ellas debe responder de forma adecuada. Por eso, una interfaz recibe también el nombre de protocolo.

4.7.4.1.2 Definir una interfaz

Una interfaz consta de 2 partes: el nombre de la interfaz precedida por la palabra reservada **interfaces**.

[**public**] **interfaces** nombre_interfaces **extends** superinterfaces

La palabra **extends** significa que se está definiendo una interfaz que es una extensión de otras; también se puede decir que es una interfaz derivada de otras; estas otras se especifican de **extends** separadas por comas.

- **personaservice.java**

Definición de las funciones a utilizar.

```
public interface PersonaService {
        public List<Persona> findAll();
        public List<Persona> search(String fullName);
        public void add(Persona p);
        public void update(Persona p);
        public void delete(Persona p);
}
```

- **personaserviceimple.java**

Declaramos las funciones y la lista de personas

```
public class PersonaServiceImpl implements PersonaService {

    //Modelo de datos.
        public List<Persona> persolis= new LinkedList<Persona>();

        //Inicializacion de libros de datos
        public PersonaServiceImpl() {
            persolis.add(
                new Persona("Ana Perez",
                    "Femenino",
                    "Peru",
                    12345678,
                    "ana@gmailcom",
                    "Apacionada a la musica contry"));
```

```
                persolis.add(
                        new Persona("Rosa Rivaz",
                                "Femenino",
                                "Honduras",
                                12356485,
                                "rosa@gmailcom",
                                "Apacionada a la pintura"));
                persolis.add(
                        new Persona("Julio Castro",
                                "Masculino",
                                "Nicaragua",
                                23445689,
                                "julio@ymailcom",
                                "Programador experto"));
                persolis.add(
                        new Persona("Juan Castillo",
                                "Masculino",
                                "Estados Unidos",
                                23445654,
                                "juan@hotmailcom",
                                "cantante experto"));
                }
        @Override
        public List<Persona> findAll() {
                // Crear la función que retorne una lista de todos los datos.
        }
        @Override
        public List<Persona> search(String key) {
                // Crear la función de Busqueda de un registro especifico.
        }
        @Override
```

```
        public void add(Persona p) {
                persolis.add(p);
        }
        @Override
        public void update(Persona p) {
                // Crear la función Actualizar.
        }
        @Override
        public void delete(Persona p) {
                // Crear la función Eliminar.
        }
}
```

- **persona.java**

Aquí declararemos los métodos set y get de las variables.

```
public class Persona {
        private String fullName;
        private String sex;
        private String pais;
        private Integer tel;
        private String email;
        private String bio;
        private List<String> country;

    // Constructor sin parámetros.
        public Persona(){}

    // Constructor con parámetros.
public Persona(String fullName, String sex, String pais, Integer tel,String email, String bio){
```

```
            this.fullName=fullName;
            this.sex=sex;
            this.pais=pais;
            this.tel=tel;
            this.email=email;
            this.bio=bio;
            this.country = new ListModelList<String>(CommonInfoService.getCountryList());
            }
            public String getfullName() {
                  return fullName;
            }
            public void setfullName(String fullName) {
                  this.fullName= fullName;
            }
      ...
     // Crear setter y getter faltantes.
}
```

4.7.4.2 Vista modelo (ViewModel)

Es una abstracción de la vista. Por lo tanto, cuando se diseña un modelo de vista, hay que analizar las funciones de la interfaz de usuario por lo que afirma que contiene y qué comportamientos tiene.

```
public class vmpersona {
            private String fullName;
            private List<Persona> personalist;
            private Persona selectedper = new Persona();
            private PersonaService perser = new PersonaServiceImpl();
      ...
}
```

Cualquier comportamiento que puede ser solicitado por la Vista es un comando en un **ViewModel**, usted debe solicitar una anotación **@Command** en el método de comportamiento (también llamado

método de comando). Después de la ejecución del método de comando, un **ViewModel** con **@NotifyChange** reflejaría los cambios de estado realizado a la vista.

```
public class vmpersona {
  ...
@Command
   @NotifyChange("personalist")
   public void search(){
         personalist = perser.search(fullName);
   }
}
```

4.7.4.3 Vista (View)

Es la interfaz de usuario - una composición de componentes. La interfaz de usuario se puede implementar por un documento ZUML o en Java. En áreas de la descripción, ZUML se utiliza para ilustrar el concepto y características.

Enlazamos la capa view y viewmodel

Para enlazar la **Vista** a un **ViewModel**, se debe aplicar un compositor llamado **org.zkoss.bind.BindComposer** y utilizar **@id('VMID') @init('FULL.QUALIFIED.CLASSNAME')** de sintaxis en el atributo viewModel. **@id()** se utiliza para establecer el id del **ViewModel** a un nombre de variable arbitraria de su elección. Usted utilizará este **ID** para hacer referencia a las propiedades del ViewModel. **@init()** se utiliza para inicializar el objeto ViewModel.

```
<window title="MVVM" border="normal" width="600px"
apply="org.zkoss.bind.BindComposer"
viewModel="@id('vm') @init('controladormvvm.vmpersona')">
```

❖ **Enlazamos la propiedad con el atributo**

Para enlazar un atributo de componente a una propiedad de ViewModel, utilice **@bind(VMID.PROPERTY_NAME)** en el atributo del componente.

```
<textbox value="@bind(vm.fullName)" />
```

❖ **Enlazamos el componente a un comando del view**

Para enlazar un evento de componente a un comando de **ViewModel**, utiliza **@command('COMMAND_NAME')** en el caso de los componentes.

```
<button label="Buscar" image="/imgs/search.png" onClick="@command('search')" />
```

❖ **Enlazamos los componentes con los datos de la lista encontrada.**

```
<listbox id="personabox"  model="@bind(vm.personalist)"
selectedItem="@bind(vm.selectedper)"
        height="160px" emptyMessage="No se encontro registro!!" >
            <listhead>
                <listheader width="" label="Nombre" />
                <listheader label="Sexo" />
                <listheader label="Pais"/>
                <listheader label="Telefono"/>
                <listheader label="E-mail"/>
                <listheader label="Biografia"/>
            </listhead>
            <template name="model">
                <listitem>
                    <listcell><textbox inplace="true"
value="@bind(each.fullName)"/></listcell>
                    <listcell>
```

```
                                        <combobox                    inplace="true"
value="@bind(each.sex)">
                                        <comboitem label="Masculino"/>
                                        <comboitem label="Femenino"/></combobox>
                                    </listcell>
                                    <listcell><combobox                    inplace="true"
model="@load(each.country)" selectedItem="@bind(each.pais)" >
                                        <template name="model">
                                            <comboitem label="@bind(each)" />
                                        </template>
                                    </combobox></listcell>
                                    <listcell><textbox                    inplace="true"
value="@bind(each.tel)"/></listcell>
                                    <listcell><textbox                    inplace="true"
value="@bind(each.email)"/></listcell>
                                    <listcell><textbox inplace="true" value="@bind(each.bio)"
multiline="true"/></listcell>
                                </listitem>
                            </template>
                        </listbox>
```

❖ **El proyecto se observará de la siguiente manera :**

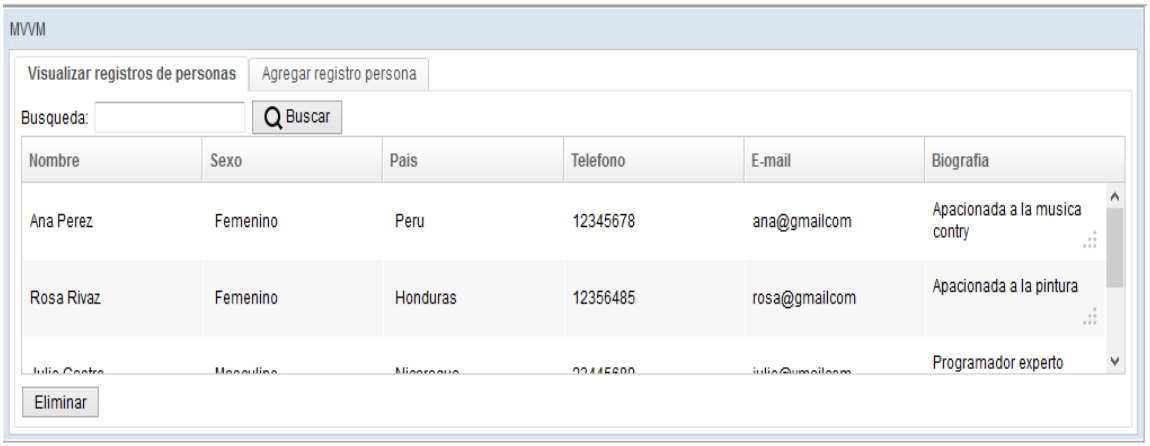


Figura 48: Resultado del proyecto.

4.7.5 Ejercicios propuestos

MVVM con acceso a datos

Ocuparemos una base de datos de autos, esta contiene campos como: id, modelo, fabricante y descripción. En esta aplicación hacemos uso de Hibernate para acceder a los datos que mostrara, asi como: crear, leer, actualizar y eliminar registros de la misma. La interfaz a usar se muestra a continuación:

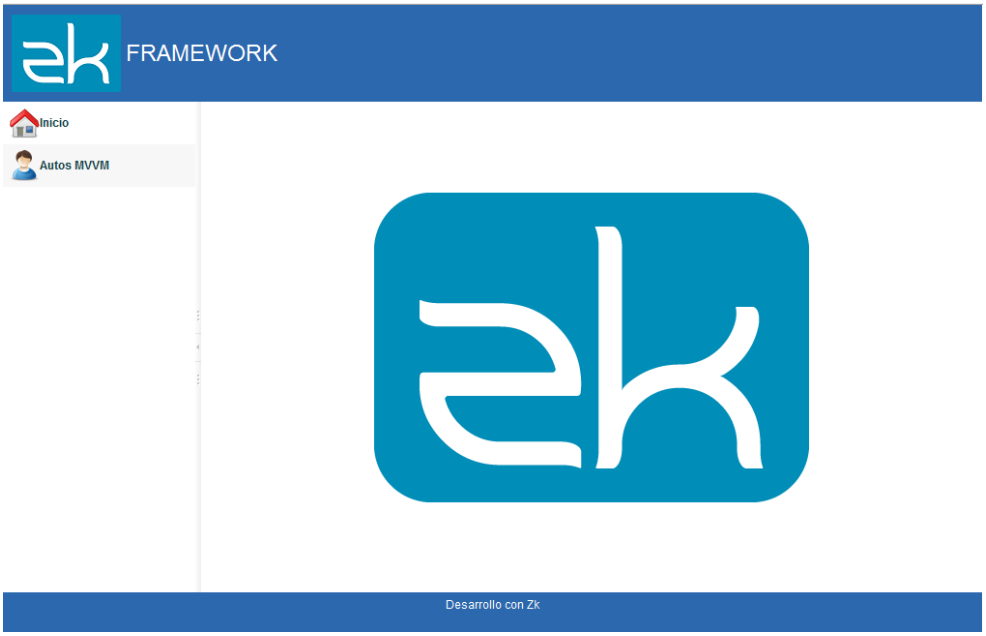


Figura 49: Apariencia de la aplicación.

Crear la clase base **tbAutos.java** que es la que se encarga de enlazar los atributos de la tabla con nuestra aplicación.

Luego creamos la interfaz **ServicioAutos.java** que contiene la definición de los métodos a implementar en este proyecto.

```
public interface ServicioAutos {
        public void add(tbAutos p);
        public void delete(tbAutos p);
        public void update(tbAutos p);
        public List<tbAutos> all();
        public List<tbAutos> search(String s);
}
```

A continuación la declaración de cada método se hace en la clase **implServicioAutos.java.**

```
public class implServicioAutos implements ServicioAutos {
        private Session sf;

        public void add(tbAutos p) {
                // Completar el codigo de esta funcion.
        }

        public List<tbAutos> all() {
                // Completar el codigo de esta funcion.
        }

        @SuppressWarnings("unchecked")
        public List<tbAutos> search(String s) {
                // Completar el codigo de esta funcion.
        }
}
```

4.7.5.1 Implementando CRUD

CRUD significa Create, Read, Update y Delete. En nuestro caso sería: Agregar, Leer, Modificar y Eliminar registros de la base de una base de datos.

4.7.5.1.1 Leer (Read)

Para mostrar un modelo de lista de datos hacemos uso de Template.

```
<listbox id="list" model="@bind(vm.lista)" emptyMessage="No se encontro resultado..."
selectedItem="@bind(vm.a)"
                        mold="paging" pageSize="5" checkmark="true">
        <listhead>
                <listheader label="Fabricante" />
                <listheader label="Modelo" />
                <listheader label="Descripcion" />
        </listhead>
        <template name="model" >
        <listitem >
                <listcell><textbox inplace="true" value="@bind(each.creador)"/></listcell>
                <listcell><textbox inplace="true" value="@bind(each.modelo)"/></listcell>
                <listcell><textbox inplace="true"
value="@bind(each.descripcion)"/></listcell>
        </listitem>
        </template>
</listbox>
```

A continuación se presenta la interfaz que muestra la lista de autos:

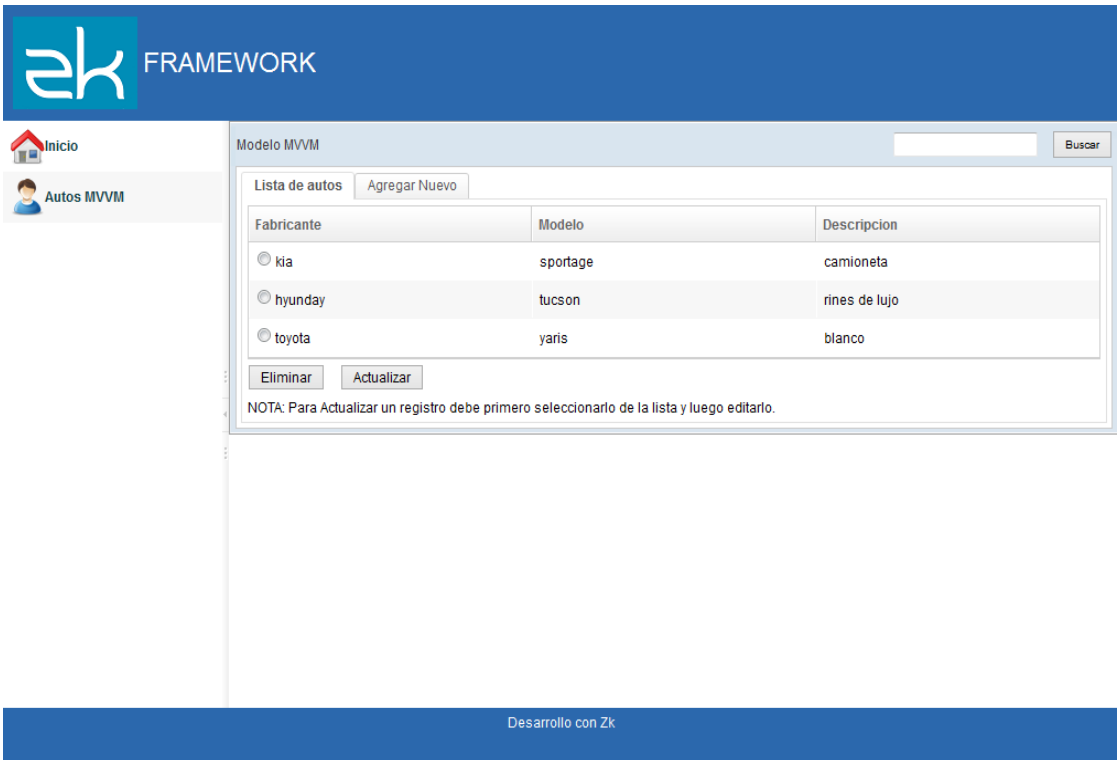


Figura 50: Probando la funcionalidad de búsqueda.

4.7.5.1.2 Agregar (Create)

Para crear nuevos elementos y guardar en la base de datos se ha utilizado la siguiente interfaz que se muestra a continuación:

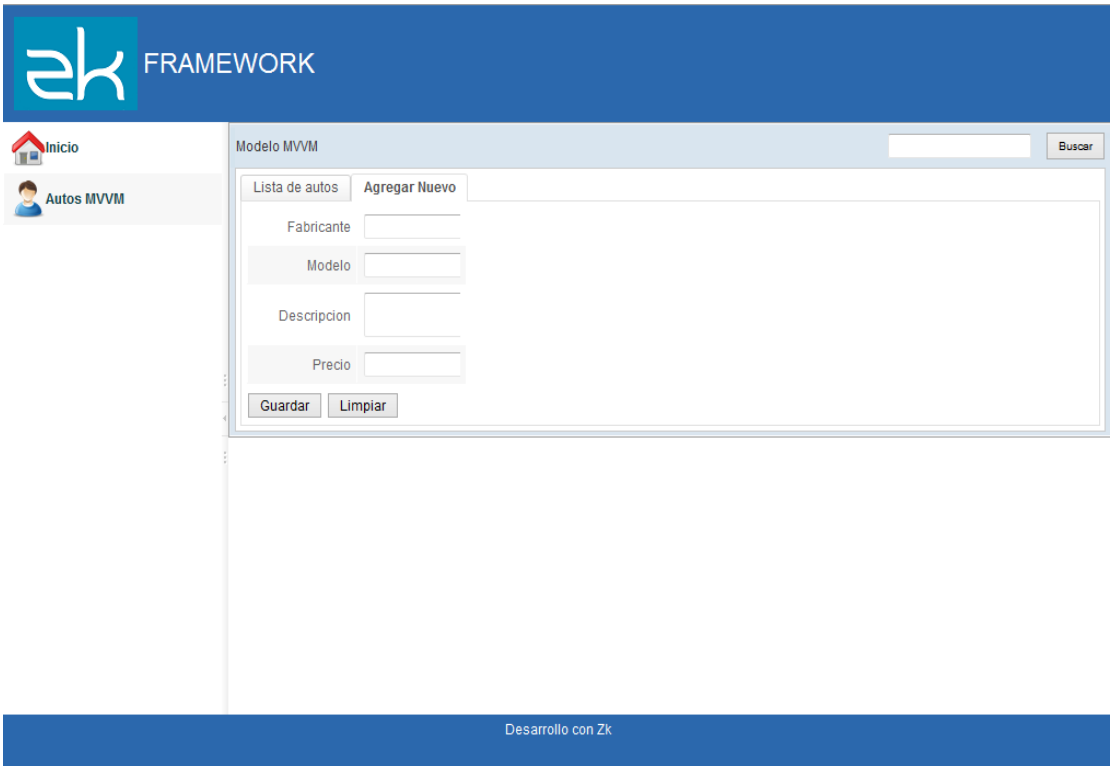


Figura 51: Agregando nuevo registro.

4.7.5.1.3 Actualizar (Update)

Para actualizar algún registro, primero hay que seleccionar un elemento de la lista, luego editarlo y por ultimo hacer clic en el botón **Actualizar.**

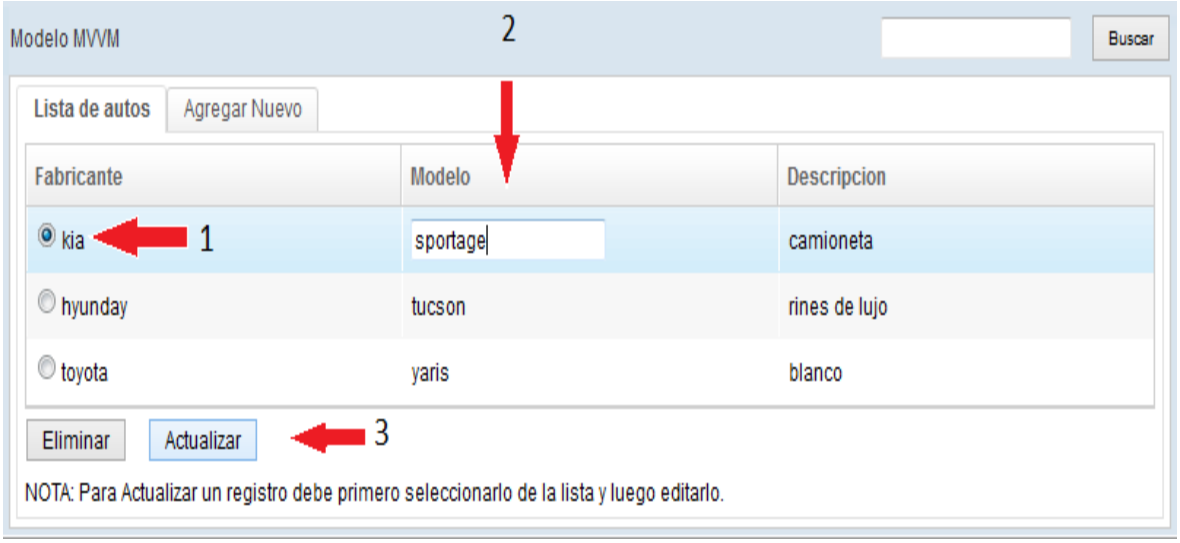


Figura 52: Funcionalidad editar.

4.7.5.1.4 Borrar (Delete)

Implementar la función eliminar es muy parecido a la función actualizar, ya que hay que seleccionar un elemento de la lista y luego hacer clic en el botón **Eliminar**.

4.7.6 Referencia

- ✓ Curso de programación, Java 2, 4ta edición – Fco. Javier Ceballos.
- ✓ http://www.zkoss.org/zkdemo/getting_started/mvvm
- ✓ http://books.zkoss.org/wiki/ZK%20Developer's%20Reference/MVVM/Data%20Binding
- ✓ http://books.zkoss.org/wiki/ZK_Developer's_Reference/MVVM
- ✓ http://books.zkoss.org/wiki/ZK_Developer's_Reference/MVVM/Advanced/Parameters

Capítulo V: Conclusiones

5.1 Conclusión.

5.2 Trabajos futuros.

5.1 Conclusión

En el presente trabajo utilizamos el framework ZK ya que es ideal para realizar aplicaciones web en Java y que fácilmente pueden ser adaptadas e integradas con otros framework adicionales o utilizar tecnologías de soporte en diferentes capas de la plataforma Java Enterprise Edition (JEE) como Hibernate o Java Persistence API (JPA). Esto reduce el tiempo de desarrollo de las aplicaciones web ya que abstrae la implementación de tecnologías importantes para la construcción de aplicaciones webs dinámicas como Ajax y JavaScript.

Los componentes del framework ZK se comportan de igual forma en cualquier navegador web, lo que no sucede con componentes del framework JSF que sufre de incompatibilidades al momento de desplegarse en algunos navegadores.

Los laboratorios se hicieron detalladamente para que los estudiantes aunque no tengan conocimiento sobre zkoss se puedan guiar y practicar dicho framework, sin la necesidad de tener mucho conocimiento sobre ello. Para lograr esto, las prácticas propuestas siguen una metodología guiada en el cual, en la primera parte se orienta al estudiante a conocer los nuevos elementos a través de la realización de un ejercicio que va paso a paso y posteriormente se le pide que con los conocimientos adquiridos, realice otros ejercicios similares.

Se lograron abordar los principales conceptos de la tecnología ZKOSS con lo cual se puede afirmar que quién utilice este documento, tendrá las bases suficientes para desarrollar una aplicación robusta empleando esta tecnología.

5.2 Trabajos futuros.

Durante el desarrollo de este trabajo se ha propuesto siempre el correcto funcionamiento de los componentes utilizados en el framework zkoss por medio de prácticas de laboratorio, por este motivo los trabajos que vemos con factible realizar son los siguientes:

- Desarrollar aplicaciones móviles con ZKOSS a fin de explorar las capacidades que este framework ofrece para este tipo de dispositivos.

- Evaluar la experiencia de estudiantes que hayan utilizado este framework a fin de conocer sus opiniones sobre el mismo.
- Promover el uso de este framework en el desarrollo de proyectos monográficos que requieran desplegarse en una plataforma web en la que ZKOSS sea una alternativa.

Bibliografía

Ceballos, F. J. (2013). *Curso de programación, Java 2, 4ta edición.* Leon, Nicaragua: RAMA.

Corporation, P. (Oct de 2013). *Create and Run Your First ZK Application with Eclipse and ZK Studio*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Installation%20Guide/Quick%20Start/Create%20and%20Run%20Your%20First%20ZK%20Application%20with%20Eclipse%20and%20ZK%20Studio

Corporation, P. (Sep de 2013). *Download ZK Framework*. Obtenido de ZKOSS: http://www.zkoss.org/download/zk

Corporation, P. (Sep de 2013). *Eclipse with ZK Studio*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Installation%20Guide/Setting%20up%20IDE/Eclipse%20with%20ZK%20Studio

Corporation, P. (2013). *Hibernate*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/Integration/Persistence%20Layer/Hibernate

Corporation, P. (2013). *Listbox Template*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/MVC/View/Template/Listbox%20Template

Corporation, P. (2013). *MVC*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVC

Corporation, P. (2013). *MVVM_Advanced_Parameters*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVVM/Advanced/Parameters

Corporation, P. (2013). *Template*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVC/View/Template

Corporation, P. (Nov de 2013). *ZK Component Reference*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Component%20Reference

Corporation, P. (2013). *ZK Essentials*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Essentials

Corporation, P. (2013). *ZK Getting Started_Get ZK Up and Running with MVC*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Getting_Started/Get_ZK_Up_and_Running_with_MVC

Corporation, P. (2013). *ZK Getting Started_Learn ZK in 10 Minutes*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Getting_Started/Learn_ZK_in_10_Minutes

Corporation, P. (2013). *ZK Live Demo*. Obtenido de ZKOSS: http://www.zkoss.org/zkdemo/

Corporation, P. (2013). *ZK Sandbox*. Obtenido de ZKOSS: http://www.zkoss.org/zksandbox/

Corporation, P. (2014). *Chapter 8: Authentication*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Essentials/Chapter%208:%20Authentication

Corporation, P. (2014). *Data Binding*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/MVVM/Data%20Binding

Corporation, P. (May de 2014). *ZK Getting Started_Get ZK Up and Running with MVVM*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Getting_Started/Get_ZK_Up_and_Running_with_MVVM

Corporation., P. (Sep de 2013). *ZK Installation Guide*. Obtenido de ZKOSS: http://books.zkoss.org/wiki/ZK_Installation_Guide

Red Hat, I. (2014). *Documentación de referencia de Hibernate*. Obtenido de HIBERNATE: http://docs.jboss.org/hibernate/core/3.6/reference/es-ES/html_single/

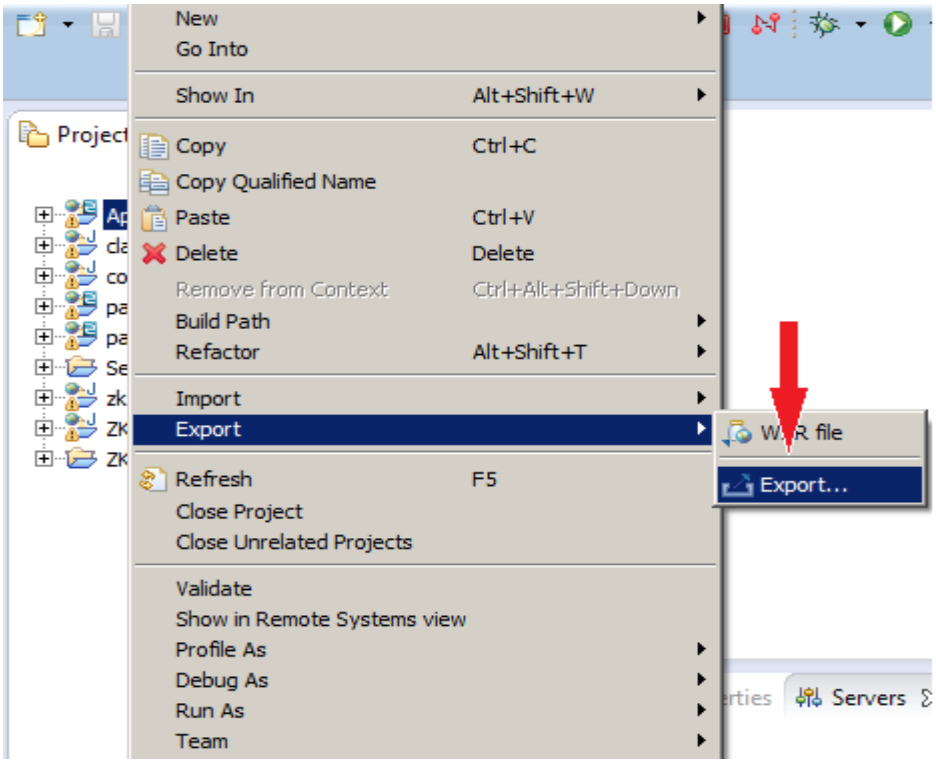
Anexos

Exportación de proyectos

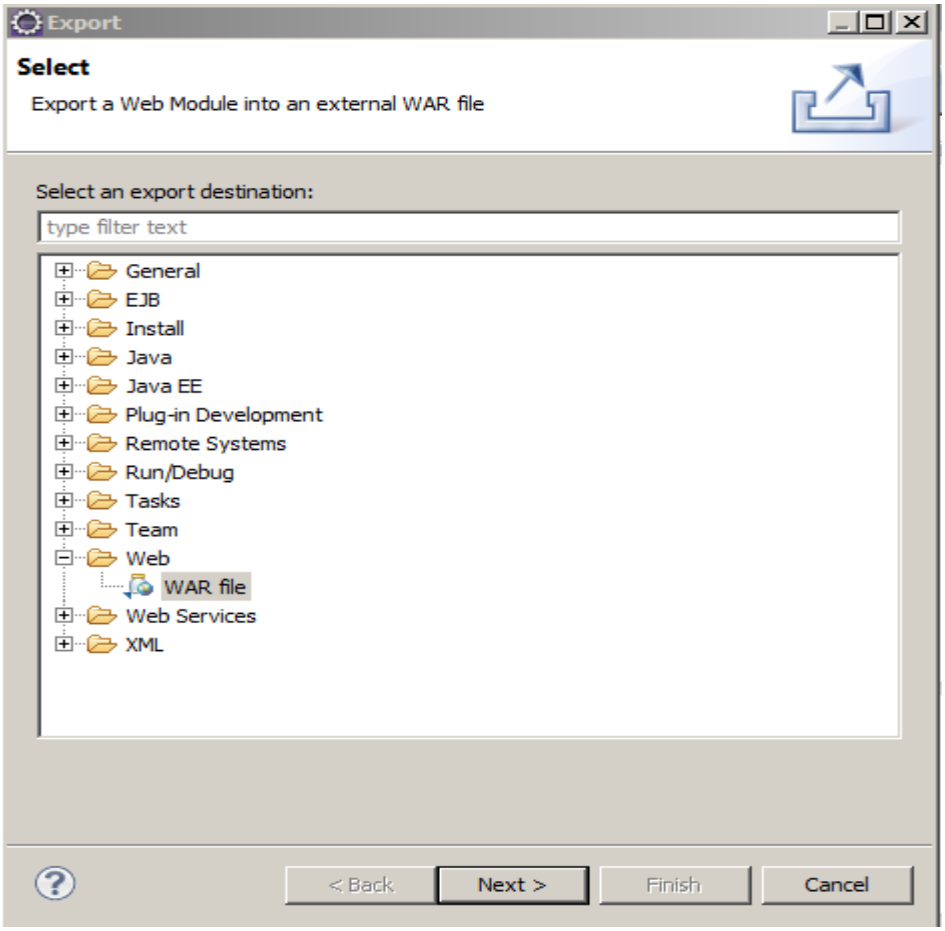
En muchos casos no estamos trabajando en una maquina propia entonces nos preguntamos:

¿Cómo hago para llevarme mi proyecto? Aquí les tengo esa respuesta.

1. Clic derecho en el proyecto a exportar, en el navegador de proyectos, a continuación seleccionamos **Export -> Export**.



2. Seleccionamos **Web -> War file**.



3. Seleccionamos un destino donde guardar el archivo y damos clic en **Finish**.

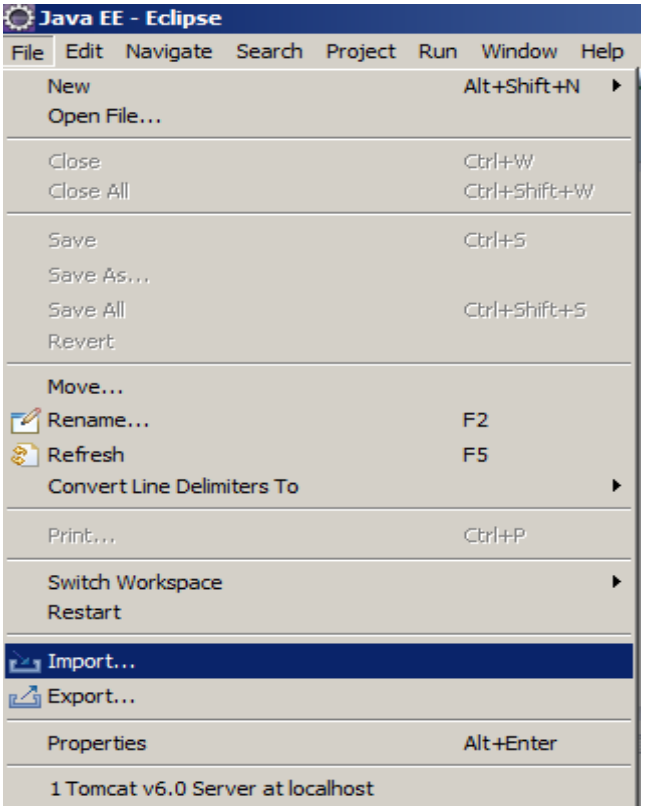
¿Por qué con extensión .WAR?

Sencillo un archivo WAR es un archivo compreso que crea Eclipse para llevar nuestros proyectos a donde queramos y así modificarlo a nuestro gusto.

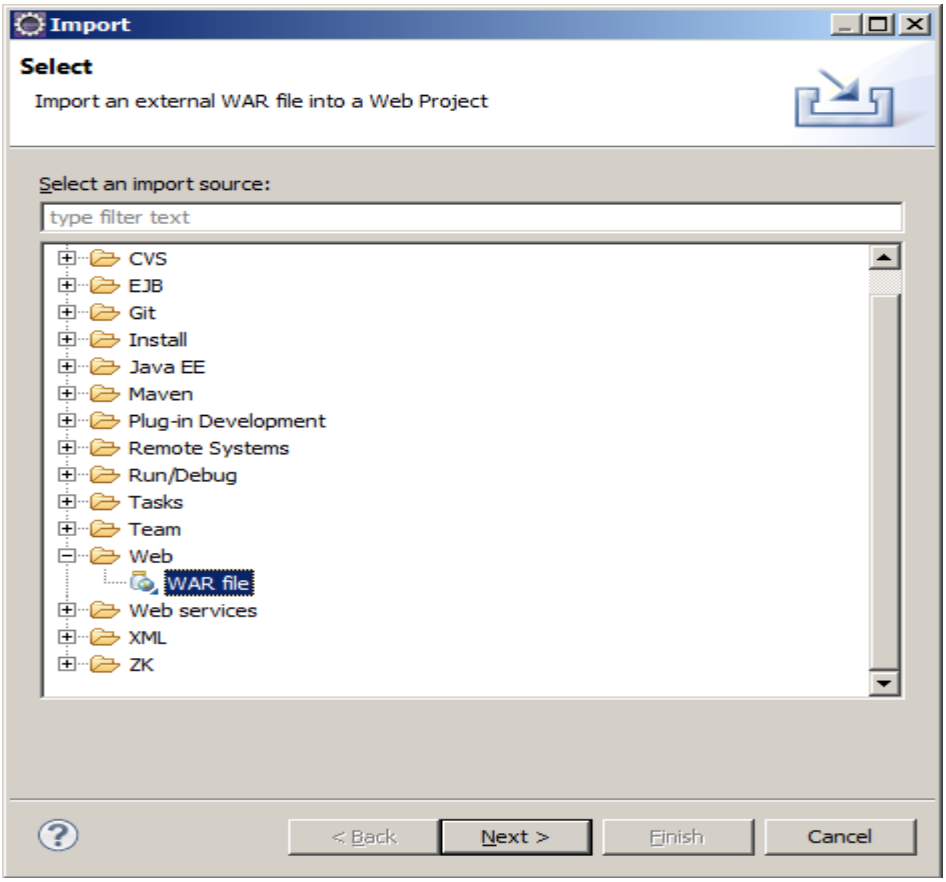
Importación de proyectos

Y luego al llegar a casa y trabajar con nuestro PC como puedo importar el proyecto que hice fuera de casa.

1. Para importar un proyecto se debe ir a: **file -> import**



2. Seleccionamos **Web -> War file.**



3. Luego aparecerá otra ventana donde deberá seleccionar el archivo a importar y escribir el nombre del proyecto. Damos clic en **Finish**.