Licencia Creative Commons.

Esta obra está bajo una licencia Reconocimiento-No comercial-Sin obras derivadas 2.5 España de Creative Commons. Puede copiarlo, distribuirlo y transmitirlo públicamente siempre que cite al autor y la obra, no se haga un uso comercial y no se hagan copias derivadas. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/2.5/es/deed.es.

MEMORIA DEL PROYECTO FINAL DE CARRERA.

UN LOGIN A TRES FRAMEWORKS.

Autor: José Antonio Delgado Trujillo.

INDICE.

1 . - Descripción del PFC....6
1.1 . - Objetivos generales y específicos....6
2 . - STRUTS2....7
2.1 . - Configuraciones previas....7
2.2 . - Descriptor web....7
2.3 . - Vistas JSP y clases helper....8
2.3.1 . - Etiquetas interfaz de usuario....8
2.3.2 . - Etiquetas generales....8
2.4 . - Clases Acción....9
2.4.1 . - Action....10
2.4.2 . - Validateable....10
2.4.3 . - ValidationAware....10
2.4.4 . - LocaleProvider....10
2.4.5 . - TextProvider....10
2.5 . - Configurar las acciones....11
2.5.1 . - Archivo de configuración XML....11
2.5.2 . - El plugin de convenciones....12
2.6 . - Anotaciones....13
2.6.1 . - Funcionamiento de la configuración....14
2.7 . - Validaciones....15
2.7.1 . - Método validate....15
2.7.2 . - Fichero de configuración XML....16
2.7.3 . - Anotaciones....18
2.7.4 . - El plugin de bean-validation....18
2.8 . - Interceptores....18
2.9 . - Patrones empleados en STRUTS2....21
2.9.1 . - Intercepting Filter....21
2.9.2 . - Front Controller....21
2.9.3 . - Application Controller....21
2.9.4 . - View Helper....22
2.9.5 . - Service To Worker....23
3 . - JSF....24
3.1 . - Configuraciones previas....24
3.2 . - Archivos de configuración....24
3.3 . - Vistas Facelets y Beans Administrados....25
3.3.1 . - Etiquetas....25
3.3.2 . - Beans administrados....26
3.4 . - Implementando eventos de acción....29
3.5 . - Validación....29
3.5.1 . - Validadores por defecto....29
3.5.2 . - Método de un Bean Administrado....30
3.5.3 . - Implementar un Validador personalizado....30
3.6 . - Ciclo de vida de una aplicación JSF....31
3.6.1 . - Fase de restauración de la vista....31
3.6.2 . - Fase de aplicación de valores de la petición....31
3.6.3 . - Fase de proceso de validación....31
3.6.4 . - Fase de actualización del modelo de datos....31
3.6.5 . - Fase de invocación....31
3.6.6 . - Fase de generación de la respuesta....32
3.7 . - Patrones empleados por JSF....32
3.7.1 . - Front Controller....32
3.7.2 . - Context Object....32
3.7.3 . - View Helper....32
3.7.4 . - Service To Worker....32
4 . - Vaadin....33
4.1 . - Arquitectura Vaadin....33

4.1.1 . - UI (Interfaz de usuario)....34
4.1.2 . - Componentes/Widgets UI....34
4.1.3 . - Motor del lado cliente....34
4.1.4 . - Servlet de Vaadin....34
4.1.5 . - Temas....34
4.1.6 . - Eventos....34
4.1.7 . - Push del servidor....34
4.1.8 . - Vinculación de datos....34
4.1.9 . - Aplicaciones lado cliente....35
4.1.10 . - Back-end....35
4.2 . - Aplicaciones del lado del servidor....35
4.2.1 . - UI....35
4.2.2 . - Page....35
4.2.3 . - VaadinSession....35
4.2.4 . - Componentes UI....35
4.2.5 . - Eventos y oyentes....35
4.2.6 . - Recursos....35
4.2.7 . - Temas....36
4.2.8 . - Vinculación de datos....36
4.3 . - Configuraciones previas....36
4.4 . - Componentes UI....37
4.4.1 . - Interfaces y abstracciones....37
4.4.2 . - Características comunes....38
4.5 . - Componentes layouts....38
4.5.1 . - VerticalLayout / HorizontalLayout....39
4.5.2 . - FormLayout....39
4.5.3 . - GridLayout....39
4.5.4 . - Panel....39
4.6 . - Componentes Field....39
4.7 . - Componentes selección....41
4.8 . - Eventos y oyentes....42
4.8.1 . - Componente Botón....42
4.8.2 . - Componente Compuesto....44
4.9 . - Vinculación de componentes....44
4.10 . - Validación....45
4.11 . - Navegación y vistas....46
4.11.1 . - Navegador....46
4.11.2 . - Proveedores de vistas....46
4.11.3 . - Implementando una vista....46
4.11.4 . - Gestionando los parámetros....46
5 . - CDI Weld....47
5.1 . - Beans....47
5.1.1 . - Tipo....47
5.1.2 . - Calificadores....47
5.1.3 . - Scope....48
5.1.4 . - Nombre EL....48
5.2 . - Métodos productores....48
5.3 . - Campos productores....48
5.4 . - Puntos de inyección....48
6 . - Vaadin CDI....48
7 . - Diseño e implementación Login y Register....49
7.1 . - Nivel layout....49
7.2 . - Nivel componente....49
7.3 . - Nivel Vista....51
7.4 . - Vinculación campos con el modelo y registro de validadores....51
7.5 . - Implementación de validadores personalizados....51
7.6 . - Implementación y registro de oyentes en botones....52
7.7 . - Diseño del backend....52
7.7.1 . - Entidad....52

7.7.2 . - Unidad de persistencia.....52
7.7.3 . - EntityManager.....53
7.7.4 . - Deltaspike.....53
7.7.5 . - PersonService.....54
8 . - Juego de pruebas.....54
8.1 . - Caso 1: Validarse usuario registrado.....54
8.2 . - Caso 2: Validarse usuario no registrado 54
8.3 . - Caso 3: Crear cuenta usuario existentes.....55
8.4 . - Caso 4: Crear cuenta usuario inexistente.....55
8.5 . - Caso 5: Validarse con la cuenta creada.....55
9 . - Bibliografía.....57

1 . - Descripción del PFC.

La plataforma J2EE se basa en un modelo de cuatro capas para aplicaciones distribuidas: cliente, presentación, negocio y datos de la empresa. En el diseño de aplicaciones web (thin-client) la capa de presentación es muy importante y debe ofrecer las siguientes funcionalidades:

- Gestionar todas las peticiones de entrada de un cliente.
- Obtener y validar los parámetros de estas peticiones.
- Convertir los parámetros a formato adecuado y trasladarlos a la capa de negocio.
- Invocar los métodos adecuados de la capa de negocio.
- Obtener los resultados y decidir la siguiente página a mostrar al cliente.
- Generar la respuesta con el estado que se tenga.
- Dar servicio de seguridad tanto de autenticación como de autorización.
- Gestionar los errores.
- Proporcionar internacionalización del lenguaje según el usuario.

La mayor parte de las veces el desarrollador implementará la capa de presentación haciendo uso del patrón de arquitectura MVC (Modelo – Vista – Controlador). Ahora bien existen distintas alternativas para el diseño de esta arquitectura así como distintas tecnologías que le dan soporte.

Para facilitar aún mas la labor al desarrollador se recurren a frameworks de presentación. Estos son librerías de componentes reutilizales que esconden gran parte de la complejidad de la capa de presentación. De esta manera el desarrollador puede volcarse más en la programación de la capa de negocio asegurándose una buena implementación por el uso de estos frameworks.

En nuestro proyecto elegimos tres frameworks de presentación y analizaremos como hacen uso de los patrones de diseño para implementar las funcionalidades que demanda la capa de presentación. Para disponer de una perspectiva de comparación veremos como cada uno de los frameworks implementa un componente **Login** en una aplicación educativa. Este componente nos servirá, además, de hilo conductor para ir explicando el funcionamiento de cada framework observando las ventajas e inconvenientes a la hora de implementar la funcionalidad que debe ofrecer el componente.

Elegiremos uno de los framework y lo extenderemos con nuevos componentes: **Registro** (formulario para darse de alta en la aplicación).

1.1 . - Objetivos generales y específicos.

Nos hemos decantado por los siguiente frameworks de presentación:

- **STRUT2** – Presenta una estructura clásica MVC con patrones fácilmente reconocibles.
- **JSF** – Es la especificación que propone la plataforma JEE.
- **Vaadin** – Diseña de forma rápida componentes de usuario gestionando su interacción a base de eventos y listeners.

Para el análisis y la comparación partiremos de un ejemplo práctico: La entrada mediante usuario y contraseña (login) en una aplicación educativa. Iremos ejemplificando como cada framework se enfrenta al diseño de un componente que encapsule esta funcionalidad al mismo tiempo que explicamos las diferencias y similitudes entre los frameworks en los siguientes apartados:

- El modelo de componentes que usa.
- La gestión de la interacción con el usuario.
- Validación de datos.
- La vinculación con la capa de negocio.
- La gestión de la navegación de páginas.
- La validación de datos.
- La gestión de errores.

- La generación de la respuesta.
- El uso de patrones de diseño.

Las funcionalidades que tendrán los componentes son:

Login (parametrizado por formato de usuario y contraseña):
- Genera el formulario de login.
- Valida el formato de entrada de los campos del formulario.
- Llama al servicio que comprueba si el usuario está autorizado.
- Ofrece la oportunidad de registrarse si el usuario no está dado de alta.

Registro (parametrizado por los campos del formulario):
- Genera el formulario.
- Valida el formato de entrada de los campos del formulario.
- Llama al servicio que creará al usuario en la base de datos de la plataforma.

Para probar los componentes diseñados implementamos una aplicación web para simular la entrada en una aplicación educativa con posibilidad de registrarnos y permitiéndonos la visión de los planos de clase de los cursos de los que seamos profesores.

Usaremos las siguientes herramientas:
- IDE: Eclipse Oxigen.2 Release (4.7.2)
- Maven : apache-maven-3.5.0
- Servidor de aplicaciones: Wildfly 11.0.0.CR1

2 . - STRUTS2

2.1 . - Configuraciones previas.

Iniciamos creando con Eclipse un proyecto Maven sin arquetipos. En el POM realizamos las siguientes configuraciones:
- Indicamos la versión java para compilar (1.8).
- Añadimos la dependencias:
  - `struts2-core` → para poder usar las clases de Struts2.
  - `javax-servet-api` → para implementar los servlets.
- Añadimos el plugin maven de wildfly para desplegar en nuestro servidor.

Para gestionar mejor las versiones de cada dependencia usamos atributos `properties`.

2.2 . - Descriptor web.

Struts2 dispone de un filtro servlet para gestionar todas las peticiones de entrada. Lo registramos en el descriptor web.

```
<filter>
  <filter-name>struts2</filter-name>
  <filter-class>org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>struts2</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Img 1: Configuración del filtro servlet en Strust2

2.3 . - Vistas JSP y clases helper.

Para la vista de entrada a la plataforma usaremos un archivo JSP que genera la interfaz de usuario y un POJO que guarde los valores de los distintos campos.

2.3.1 . - Etiquetas interfaz de usuario.

Struts 2 dispone de una librería de etiquetas para generar los distintas interfaces de usuario. Tan sólo es necesario incluir la directiva de la librería al inicio del archivo JSP. En la siguiente tabla tenemos las principales etiquetas de interfaz de usuario.

Etiqueta	**Elemento HTML que genera**
`textfield`	Campo de entrada de texto.
`textarea`	Campo de entrada multilínea.
`password`	Campo de tipo contraseña.
`checkbox`	Casilla de verificación.
`checkboxlist`	Lista de casillas de verificación.
`radio`	Lista de botones de opción.
`select`	Lista desplegables de opciones.
`form`	Formulario de entrada
`submit`	Botón de tipo submit (enviar formulario)

Atributos comunes de las etiquetas.

- `name` – nombre del campo del POJO al que está vinculado la etiqueta.
- `value` – valor que almacena el elemento.
- `label` – etiqueta que se muestra para el elemento.
- `key` – sustituye a los tres anteriores. Para poder utilizar key debemos crear un archivo de tipo `properties` que es donde Struts 2 buscará el valor del atributo `label`.

Atributos propios de los componentes de selección.

- `list` – campo de la clase acción que genera todas las opciones.
- En el caso de que las opciones que se muestran al usuario sean diferentes a los valores que se guardan (por ejemplo, si se guardan abreviaturas y se muestran nombres completos).
- `list` – objetos con dos campos (el que se muestre y el que se guarda).
- `listkey` – campos de los objetos list que se usan para guardar.
- `listvalue` – campos de los objetos list que se usan para mostrar.

Atributo propio de la etiqueta `form`.

- `action` – nombre de la clase que gestionará los valores del formulario. Debe contener como campo el objeto vinculado al formulario.

2.3.2 . - Etiquetas generales.

`property` – muestra le valor de la propiedad de un objeto dado por el atributo value. O en su defecto la propiedad del objeto que se encuentre en la cima de la pila de valores (value stack).

`url` – Crea un enlace hacia la URL. En caso de que queramos añadir parámetros adicionales usaremos etiquetas param anidadas. Los pricipales atributos son:

- `action` – nombre de la acción que genera la URL si no se usa el atributo value.
- `var` – nombre para referenciar el enlace.
- `value` – destino del enlace si no se usa el atributo action.

Nuestra interfaz constará de dos vistas jsp:

- `login` para dar acceso a la plataforma.
- `register` para registrase y crear una cuenta.

En login insertaremos:

- Un campo de entrada texto para el usuario.
- Un campo de tipo contraseña.
- Un botón para enviar el formulario que valide al usuario.

```
<s:form action = "login">
    <s:textfield key="credentialBean.user"/>
    <s:password key="credentialBean.pass"/>
    <s:submit key="submit" />
</s:form>
```

Img 2: jsp que genera el formulario login.

Al usar los atributos key debemos crear una archivo de `properties` donde Struts busque las etiquetas para estos campos.

```
Login.properties
1 credentialBean.user=Usuario
2 credentialBean.pass=Contraseña
3 submit=Acceder
```

Img 3: Archivo properties que genera la etiquetas.

A esta vista vincularemos el POJO `Credential` que almacenará en sus propiedades los valores de los campos.

```
public class Credential {

    private String user;
    private String pass;
```

Img 4: Clase de los objetos que guarda los valores del formulario

En register insertaremos los siguientes campos:

- Campos de entrada de texto para nombre, apellidos, email y usuario.
- Campo de contraseña.
- Casilla de verificación para saber si un profesor es tutor.
- Lista desplegable para indicar el IES donde trabaja el profesor.
- Lista de casilla de verificación para indicar los cursos que da el profesor.

Para dar respaldo a la vista creamos el POJO `Person`.

2.4 . - Clases Acción.

En Struts 2 las acciones son aquellas clases que responden a la acción del usuario, ejecutan la lógica de negocio y devuelven un resultado que le permite al framework decidir que vista generar.

En Struts2 se considera una clase acción cualquiera que implemente la interfaz Action. Sin embargo, lo más normal es extender la clase AcctionSupport que implementa por defecto una serie de interfaces que nos da la mayoría de las funcionalidades necesarias. Veamos con un poco más de detalle que nos ofrecen estas interfaces.

2.4.1 . - Action.

Dispone de los campos `static` de tipo `String` que determinan los resultados de la acción y permiten decidir que vista generar. Estos campos son:

- `ERROR` – Hubo un fallo en la ejecución de la acción.
- `INPUT` – La acción requiere más entradas para poder ejecutarse.
- `LOGIN` - La acción no puede ejecutarse hasta que el usuario se identifique.
- `NONE` – La acción se ejecutó con éxito pero no muestra ninguna vista.
- `SUCCESS` – La acción se ejecutó con éxito.

Dispone además, del método `String execute()` donde debe implementarse la lógica de la acción.

2.4.2 . - Validateable.

Dispone del método `void validate()` donde se implementa la lógica de las validaciones.

2.4.3 . - ValidationAware.

Proporciona un grupo de métodos para gestionar los mensajes de error a nivel de campo (en un `Map`) o de acción (en `Collection`). Destacamos entre otros los siguientes métodos:

- `void addFieldError( String fieldName, String errorMessage)` – Añade un mensaje de error al campo dado.
- `Collection<String> getActionErrors()` - Obtiene todos los errores a nivel de acción.

2.4.4 . - LocaleProvider.

Permite personalizar el `locale` que vamos a utilizar. El valor por defecto lo proporciona el navegador pero podemos sobre escribir el método `Locale getLocale()` para usar otro.

2.4.5 . - TextProvider.

Permite acceder a mensaje localizados en ficheros `properties`. Para ello hace uso de distintas variantes del método `getText`:

- `String getText(String key)` – Nos da el mensaje pasando la clave por parámetro.
- `String getText(String key, List<?> args)` – Nos da un mensaje parametrizado pasando la clave y la lista de los parámetros.

Junto con la anterior interfaz, Struts2 permite la internacionalización. Basta con crear distintos archivos `properties` que guardan los mensajes en distintos idiomas añadiendo la terminación del `locale` (_es, _fr, …).

En nuestro ejemplo implementamos la clase Login que extiende `ActionSupport` y en cuyo método `execute` comprobaremos si el usuario y contraseña son válidos.

```java
public class Login extends ActionSupport {

    private static final long serialVersionUID = 1L;
    private Credential credentialBean;
    private FakeService service = new FakeService();

    /**
     * Comprobamos si el usuario y contraseña son válidos
     */
    public String execute() throws Exception {
        String result = SUCCESS;
        if (!service.validar(credentialBean)) {
            addFieldError("credentialBean.user", "usuario o contraseña no válido");
            result = INPUT;
        }
        return result;
    }
```

Img 5: Método execute de la acción Login.

En el código observamos que:

- Delegamos en un servicio la lógica de validar.
- El usuario y la contraseña son los campos del objeto `credentialBean` que está vinculado al formulario de entrada.

2.5 . - Configurar las acciones.

Para configurar una acción debemos:

- Enlazar la acción con la clase que la implementa.
- Enlazar los distintos resultados que genera con las correspondientes vistas.

Disponemos de tres formas diferentes de configurar:

- Archivo de configuración XML.
- Plugin de convenciones.
- Anotaciones.

2.5.1 . - Archivo de configuración XML.

Struts2 utiliza el archivo `strust.xml` para configurar cualquier elemento del framework. Para el caso de acciones usaremos la etiqueta `action` con los siguientes atributos:

- `name` – nombre de la acción.
- `class` – nombre completo de la clase que implementa la acción.
- `method` – método de la clase que implementa la lógica de la acción.

Anidada a la etiqueta `action` tenemos la etiqueta `result` con los atributos:

- `name` – resultado en formato `String` que devuelve la acción
- `type` – el tipo de respuesta a enviar (por defecto `dispatcher`)

Entre la etiqueta `result` tendremos el resultado que genera la acción (normalmente un archivo jsp).

Aunque lo normal es generar una vista también son posibles otro tipos de `result`. Veamos los distintos valores del atributo `type` de la etiqueta `result`.

`dispatcher`.
Es el tipo de resultado por defecto y genera un página JSP que se entrega al servidor mediante el método `RequestDispatcher.foward()`.

`redirect`.

Llama al método estándar `response.sendRedirect()`, provocando que el navegador cree una nueva petición con la localización dada por parámetro. Con esta redirección se perderán los parámetros de la petición inicial.

`chain`.
Nos permite ejecutar otra acción como resultado de la acción inicial. Esto quiere decir que la salida está controlada por la acción encadenada. Al contrario que en `redirect` los parámetros iniciales con copiados y por tanto accesibles en la acción encadenada.

Existen otros tipos de `result` que nos dirigen a otras tecnologías de vistas como son FreeMaker y Velocity. Ambas son motores de plantillas con etiquetas más simplificadas que las empleadas por JSP.

En nuestro ejemplo debemos enlazar la acción login con su clase e indicar que vistas generar para los dos posibles resultados: `input` y `success`.

```
<!-- action configuration by XML -->
    <action name="inputLogin" class="uoc.jdelgadot.loginstruts.action.Login" method="input">
        <result name="input">/login.jsp</result>
    </action>
    <action name="login" class="uoc.jdelgadot.loginstruts.action.Login" method="execute">
        <result name="input">/login.jsp</result>
        <result name="success">/uoc.jsp</result>
    </action>
```

Img 6: configuración de la acción login vía XML.

Vemos en el archivo de configuración dos acciones:

- `inputLogin` – Genera el formulario de entrada a la plataforma y que será invocado por un enlace desde la página de inicio.
- `login` – Invocada por la etiqueta form y que es la que gestiona el control de la entrada a a plataforma.

Tan sólo nos queda crear el enlace que invoca la acción inputLogin desde una página de inicio.

```
<body>
    <h2>Inicio</h2>
    <s:url action = "inputLogin" var="linkLogin"></s:url>
    <p><a href="${linkLogin}" >Acceso a la plataforma</a></p>
</body>
```

Img 7: Enlace a la acción inputLogin.

2.5.2 . - El plugin de convenciones.

Si queremos evitar la configuración mediante XML podemos usar el plugin de convenciones que permite a Struts2 bajo ciertas condiciones:

- Enlazar nombres de acciones (invocados por URL) con métodos de clases de acciones.
- Localizar las clases de las acciones en determinados paquetes.
- Localizar las vistas en determinados paquetes.
- Generar la vista adecuada en función del resultado de la acción.

El convenio buscará las acciones entre aquellas clases que implemente la interfaz `Action` o cuyo nombre acabe en `Action` en aquellos paquetes cuyos nombres contengan `struts`, `struts2`, `action` ó `actions`. Estos paquetes serán considerados la raíz.

Estas clases se invocarán ante determinadas URLs. Esta URL se basará en el nombre completo de la acción:

- El espacio de nombre de la URL será el formado entre la raíz y el paquete en el que la clase acción se encuentre.
- El recurso de la URL se formará usando el nombre de la clase. Se quita el final `Action`, si lo tuviera, y se transforman las mayúsculas por guiones.

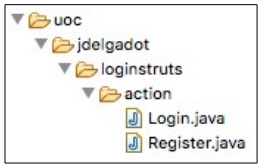
Veamos un ejemplo:
- Clase que implementa Action: `com.example.actions.products.ShowCompany`
- Espacio de nombre de la URL: `/products`
- Recurso de la URL: display: `show-company`
- URL completa: `/products/show-company`

El convenio localizará las vistas en la carpeta `/WEB-INF/content`. Para saber que vista generar usará el nombre de la acción añadiendo un guión y el nombre del `result`.

Si el método execute de la clase ShowCompany devuelve `SUCCESS` Strust2 invocará la vista `show-company-success.jsp`

Podemos modificar distintos comportamientos por defecto del plugin mediante properties:
`struts.convention.action.packages` - Indica los paquete raíz que contienen las clases acción.
`struts.convention.result.path` - Indica la carpeta que contienen las vistas.

En nuestro ejemplo vamos a usar el convenio para configurar la acción `register` que se invoca cuando el usuario confirma el formulario de registro.
Nuestra clase `Register`, que implementa la interfaz `Action` (ya que extiende `ActionSupport`) está en el paquete `uoc.jdelgadot.action` por tanto localizable por el convenio. Analizando su nombre completo podremos ver qué URL la invoca:
- nombre de la clase: `uoc.jdelgadot.action.Register`
- Espacio de nombre de la URL: `/`
- Recurso de la URL: `register`
- URL completo: `/register`



Img 8: Localización clases acción.

El usuario al pulsar el botón de registro envía la URL: `/register`. Struts2 ejecutará el método `execute` de la clase `Register`. Si se completa la acción con éxito (return `SUCCESS`) generará la vista `register-success.jsp`. que debería localizarse en `WEB-INF/content`.

2.6 . - Anotaciones.

En ocasiones el convenio no es suficiente para configurar la acción. Por ejemplo puede suceder que quisiéramos:
- Asociar varias URL a una misma acción.
- Invocar métodos distintos de execute.

Para configurar la invocación de la acción, `@Action` con los atributos:
- `value` - url que invoca la acción.
- `results` - lista de anotaciones @Result para indicar las distintas vistas en función del String devuelto.

Para configurar los resultados de la acción, `@Result` con los atributos:
- `name` - indica uno de los String que puede devolver la acción

- `type` - tipo de resultado que se generará (por defecto una vista jsp)
- `location` - lugar donde se localiza la vista.
- `params` - lista de par.ametros que podemos pasar a la vista.

Estas anotaciones pueden ser a nivel de acción (si por ejemplo todos results se aplican a todos los métodos de la acción) o a nivel de método (si queremos invocar distintos métodos de una misma acción).

En nuestro ejemplo vamos a configurar con anotaciones las acciones inputLogin y login, a nivel de método, quitando la configuración realizada con XML.

```
@Action(value = "login", results= {@Result(name="input", location="login.jsp"),
        @Result(name="success", location="uoc.jsp")} )
public String execute() throws Exception {
```

Img 9: Anotaciones para login

Img 10: Anotaciones para inputLogin.

2.6.1 . - Funcionamiento de la configuración.

Ya que henos explicado las tres formas de configurar las acciones veamos como funciona nuestro código.

Cuando el usuario pulsa el enlace Acceso a la plataforma se invoca la acción `inputLogin`.

Img 11: Vista de la página de inicio.

Struts2 localizará según la configuración (XML, plugin o anotaciones) la acción y el método a ejecutar. Instancia la acción `Login` y ejecuta el método `input`. Éste tan sólo devuelve el `String INPUT` que le permite saber a Struts2 que debe generar la vista `login.jsp`. La clase `Login` dispone de un campo de tipo `Credential` que encapsula el usuario y la contraseña que quedan vinculados a los campos del formulario a través del atributo key.

Img 12: Vista del formulario login.

Tras rellenar los datos el usuario pulsa el botón de acceder invocando la acción `login`. Struts2 comprueba que debe ejecutar la acción `execute` de la clase `Login`. Usará los `setter` de `Credential` para almacenar los valores que ha introducido el usuario y ejecutará el método

`execute`. Este delega en la clase `FakeService` para comprobar si el usuario y contraseña, devolviendo `INPUT` (si no son válidos) ó `SUCCESS` (si lo son).
En el primer caso vuelve a generar la vista `login.jsp` con el mensaje de error.

Img 13: Vista tras login fallido.

En el segundo caso genera una vista de entrada a la plataforma confirmando el usuario que se ha validado.

Img 14: Vista tras login con éxito.

2.7 . - Validaciones.

Las validaciones nos permiten añadir restricciones a los campos de entrada. Strust2 permite realizar validaciones de 4 formas diferentes.

2.7.1 . - Método validate.

Esta sería la forma más sencilla de validación aprovechando la funcionalidad que da la clase `ActionSupport` gracias a la implementación de la interfaces:

- `Validateable` que proporciona le método `void validate()` para implementar la lógica de la validación.
- `ValidationAware` que proporciona los métodos necesario para la gestión de los mensajes de error.

En caso de que el método validate añade mensajes de error devolverá el `String INPUT` como resultado de la acción invocada.

En nuestro ejemplo queremos añadir las siguientes restricciones para los usuario y contraseñas permitidos en nuestra plataforma:

- Ambos campos son obligatorios y deben tener una logintud mínima de 6 caracteres.
- La contraseña debe incluir al menos un carácter no alfanumérico.

Implementamos el método validate de la clase Login de la siguiente forma:

```
public void validate() {

int lengthUser = credentialBean.getUser().length();
int lengthPass = credentialBean.getPass().length();
boolean validPass =Pattern.matches(".*\\W.*", credentialBean.getPass());

// comprobación de la longitud de usuario
if(lengthUser==0) {
    addFieldError("credentialBean.user", "Introdude un usuario"); }
else{
    if(lengthUser < 6)
        addFieldError("credentialBean.user", "Al menos 6caracteres"); }

// comprobación de la longitud de la contraseña.
if(lengthPass==0) {
    addFieldError("credentialBean.pass", "Introduce una contraseña"); }
else {
    if(lengthPass < 6) addFieldError("credentialBean.pass", "Al menos 6 caracteres"); }

// comprobación del patrón de contraseña requerido.
if(!validPass)
    addFieldError("credentialBean.pass", "Al menos un carater no alfanumérico");

}
```

Img 15: Implementación del método validate

Además hemos configurado el `result INPUT` para que se vuelva a ejecutar de nuevo la vista `login.jsp` con los mensaje de error asociados a los campos.

Img 16: Mensajes de error en usuario y contraseña.

2.7.2 . - Fichero de configuración XML.

Struts 2 dispone de una serie de implementaciones de validaciones por defecto entre los que destacan.

Nombre	Funcionalidad que comprueba con el campo asociado
`conversion`	Error de conversión.
`date`	Date dentro de un rango determinado.
`double`	Double dentro de un rango determinado.
`email`	String es un email válido.
`fieldexpressi`	Valida el String con una expresión OGNL.

on	
regex	Valida el String con una expresión regular.
required	Campo nulo.
requiredstring	String no nulo o con longitud cero ("")
stringlength	String tiene una determinada longitud.
url	String es una URL válida.

Para registrar los validadores debemos crear un archivo con el nombre de la clase acción que gestiona los campos a validar añadiendo `-validation.xml`. Este fichero debe estar en el mismo paquete en que se encuentre la acción, dentro de la carpeta `resources`.

Este archivo tiene como etiqueta raíz `validators`. Disponemos de dos formas para registrar los validadores en los campos asociados.

2.7.2.1 . - Agrupando los validadores por tipo.

Usamos la etiqueta validator con el atributo type para indicar el nombre del validador anidando una etiqueta param para indicar el nomre del campo a validar.

```
<validator type= “...”>
     <param name=”fieldName”>...</param>
</validator>
```

2.7.2.2 . - Agrupando los validadores por campo.

Usamos una etiqueta field con atributo name para indicar el nombre del campo a validar. Anidamos etiquetas field-validator con atributo type paraindicar el validador.

```
<field name=”...”>
     <fiel-validator type = “” />
</field>
```

Para indicar los mensajes de error anidamos etiquetas `message`.

En nuestro ejemplo vamos a registrar las validaciones necesarias en los campos usuario y contraseña del formulario register.
Para ello creamos un archivo de validaciones Register-xml.

```
<field name="personBean.credential.pass">
    <field-validator type="requiredstring">
        <message>Introduce una contraseña</message>
    </field-validator>
    <field-validator type="stringlength">
        <param name="minLength">6</param>
        <message>Al menos 6 caracteres</message>
    </field-validator>
    <field-validator type="regex">
        <param name="regex"><![CDATA[.*\W.*]]></param>
        <message>Al menos un caracter no alfanumérico</message>
    </field-validator>
</field>
```

Img 17: Validaciones XML para el campo contraseña en el formulario register.

2.7.3 . - Anotaciones.

Si queremos evitar el uso de ficheros XML podemos utilizar anotaciones para registrar las validaciones.

Las clases acción que contienen los campos que van a ser validados deben tener la anotaci´n @Validation
Las anotacines pueden ser de dos formas:

2.7.3.1 . - A nivel de método.

Usará la anotación @Validations y se agruparán las validaciones por tipos. Las principales agrupaciones disponibles son:

- requiredFields
- requiredStrings
- stringLengthFields
- regexFields
- emails
- fieldExpressions

Dentro de cada agrupación irá entre llaves las anotaciones de los validadores del mismo tipo.

En nuestro ejemplo configuramos todas las validaciones de campos requeridos del formulario register para los campos nombre, apellidos y email.

```
@Validations(
    requiredStrings = {
            @RequiredStringValidator(fieldName="personBean.name", message="Introduce tu nombre"),
            @RequiredStringValidator(fieldName="personBean.surname", message="Introduce tu apellido"),
            @RequiredStringValidator(fieldName="personBean.email", message="Introduce tu email")
        }
)
public String execute() {
```

Img 18: Validaciones campos requeridos con anotaciones a nivel de método.

2.7.3.2 . - A nivel de propiedad.

Usará las anotaciones propias de un validador en el método setter de la propiedad. La anotación se formará anteponiendo el signo @ al nombre de la clase del validador y tendrá como parámetros el mensaje de error con el atributo message.

2.7.4 . - El plugin de bean-validation.

2.8 . - Interceptores

Struts 2 como framework de presentación realiza una serie de tareas antes y después de invocar una acción. Así, antes de la llamada al método `execute` se debe entre otras tareas:

- Gestionar las excepciones que se generen.
- Convertir los parámetros de tipo `String` de una petición a los tipos Java adecuados.
- Completar todos los campos de los objetos con los parámetros de la petición cuyos nombres coincidan.
- Llamar a los métodos de validación.

Y después de la llamada debemos:

- Gestionar las excepciones que se generen.
- Convertir los tipos java de los campos de los objetos a tipo `String` para que puedan mostrarse en la vista.
- Seleccionar la vista adecuada teniendo en cuenta el `result` devuelto por el método `execute`.

De todas estas tareas se encargan los interceptores que son en realidad clases java que se ejecutan en un determinado orden. Las clases implementan la interfaz Interceptor que define tres métodos.

- `void destroy()` - Elimina cualquier recurso que haya almacenado el interceptor.
- `void init()` - Se llama justo después de crear el interceptor y antes de procesar la petición con intercept, dando la oportunidad de inicializar cualquier recurso que se necesite.
- `String intercept(ActionInvocation invocation)` – Procesa la petición o la cortocircuita devolviendo un `String result`.

Algunos interceptores hacen su trabajo sin tener en cuenta la acción que está siendo invocada pero otros actuarán solo según las interfaces que implemente la acción. La propiedad `action` de `ActionInvocation` contienen la instancia de la acción que está siendo invocada. Mediante el método `instanceof` el interceptor revisa la acción y decide si debe procesar la petición o no.

Los interceptores usan el método `invoke()` de `ActionInvocation` para pasar la invocación de la acción al siguiente interceptor o a la propia acción si es el último interceptor de la pila.

Strust2 proporciona una pila de interceptores por defecto, para poder utilizarla tan sólo indicamos en el fichero `strust.xml` en la etiqueta `package` un atributo `extends` con el valor struts-default.

```
<package name="loginstruts" extends="struts-default">
```

Img 19: Uso de la pila de interceptores por defecto.

En la siguiente tabla describimos brevemente la funcionalidad de los 19 interceptores que forman la pila por defecto.

Nombre	**Funcionalidad**
`exception`	Vincula clases de tipo `Exception` con nombres de tipo `result`. De esta manera cuando una acción lanza una excepción devuelve un tipo de result, por lo que la gestión de excepciones queda dentro del framework.
`alias`	Da un alias común a parámetros con nombres diferentes. Esto nos permite encadenar acciones que tienen propiedades similares pero nombres diferentes.
`servletConfig`	Establece en un Map todas las propiedades basadas en las interfaces de tipo `aware` que la acción implementa (parámetros de servlet, contexto, sesión,...)
`i18n`	Establece el locale especificado en la sesión como el `locale` para la petición de la acción.
`prepare`	Ejecuta el método prepare de aquellas acciones que implementan la interfaz `Preparable`.
`chain`	Copia todas las propiedades de cada objeto de la pila de valores, permitiendo la acción actual acceder a todas las propiedades de la acción previa. De esta forma podemos encadenar acciones.
`scopedModelDriven`	Si la acción implementa `ScopedModelDriven` obtiene el objeto modelo y lo almacena llamando al método `setModel`.
`modelDriven`	Si la acción implementa `ModelDriven`, llama al método `getModel()` y pone su resultado en la pila de valores. De esta manera en la cima de la pila estará el modelo y no la acción.
`fileUpload`	Establece las propiedades de la acción relacionadas con la subida d eun fichero. Además gestiona los mensajes de errores que se puedan producir.

checkbox	Detecta las casillas de verificación no seleccionadas e inserta el valor false en la propiedad de la acción.
datetime	
multiselect	Detecta en los campos de selección múltiple de un formulario que no se ha seleccionado ninguna opción y establece a null el parámetro de la acción correspondiente.
saticParams	Si la acción implementa la interfaz Parametrizable, un Map de parámetros estáticos configurados en el archivo struts.xml se pasan automáticamente a la acción.
acctionMapppingParams	
params	Establece las propiedades de la acción a partir de los parámetros de la petición.
conversionError	Añade los errores de conversión desde ActionContext a los campos de error de la acción.
validation	Ejecuta las validaciones de nuestras acciones y añade los mensajes de error.
workflow	Si la acción implementa ValidationAware, comprueba la presencia de errores y devuelve el String input sin invocar la acción.

Podemos configurar nuestra propia pila de interceptores en lugar de usar la que proporciona por defecto Struts2.
Para ello disponemos de los siguientes elementos:

- interceptor - indica el nombre y la clase de un interceptor
- interceptor-ref – indica el nombre de un interceptor o de un apila.
- interceptor-stack - indica el nombre de una pila
- default-interceptor-ref – Indica la pila por defecto de interceptores que usará Struts2.

Por ejemplo si quisiésemos añadir para todas nuestras acciones los interceptores logger y timer configuraríamos en strus.xml de la siguiente manera.

```
<package name="basicstruts2" extends="struts-default" >
    <interceptors>
        <interceptor-stack name="appDefault">
            <interceptor-ref name="timer" />
            <interceptor-ref name="logger" />
            <interceptor-ref name="defaultStack" />
        </interceptor-stack>
    </interceptors>

    <default-interceptor-ref name="appDefault" />
```

Img 20: Pila de interceptores personalizada.

Podemos modificar el comportamiento por defecto de algún interceptor mediante el uso de parámetros. Así por ejemplo podemos usar <param name=excludeMethods> para indicar una lista de métodos que no se verán afectados por el interceptor.

También es posible configurar interceptores para una acción individual en lugar de para todas las acciones del paquete. Para ello debemos anidar en el elemento acción al definición de la pila.

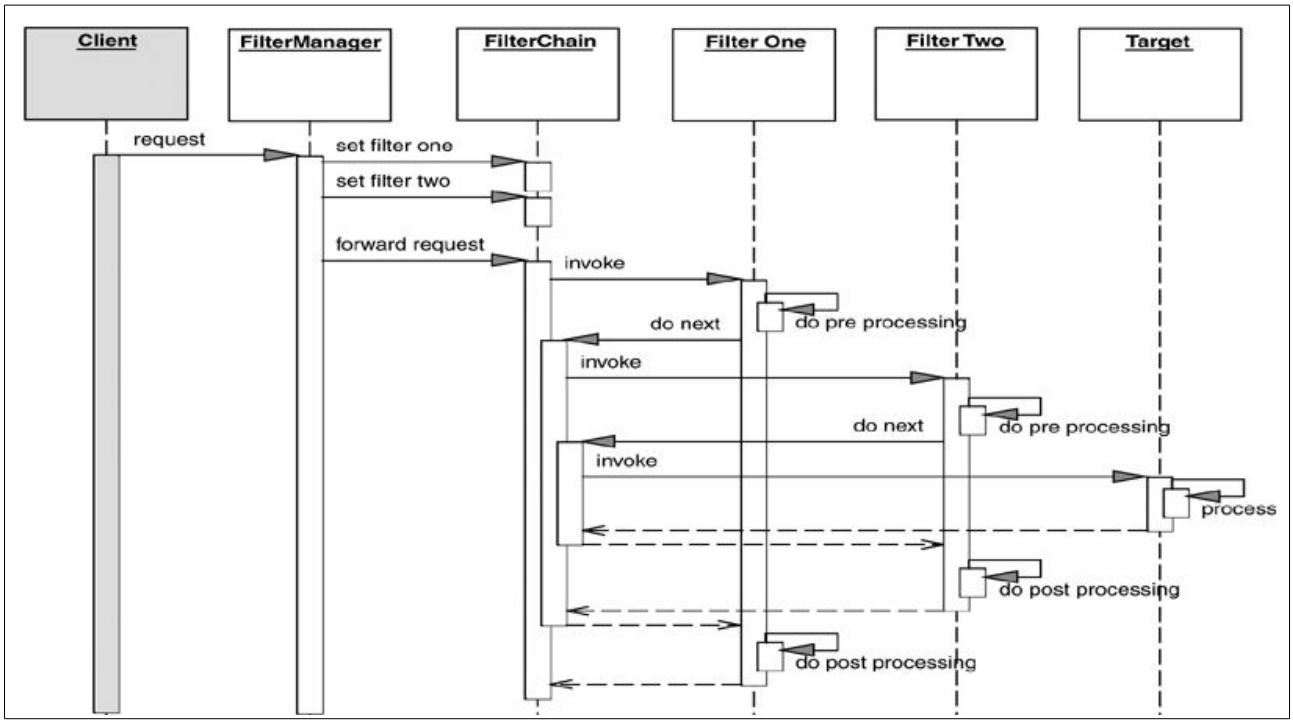
2.9 . - Patrones empleados en STRUTS2.

Tras estudiar el framework analicemos ahora que patrones de diseño se han empleado para su implementación.

2.9.1 . - Intercepting Filter.

Struts2 dispone de una pila de interceptores por defecto encargada de realizar todas aquellas tareas comunes de un framework de presentación antes y después de invocar la lógica de la acción. Incluso es posible configurar una pila personalizada de interceptores a través del fichero `struts.xml`

Se sigue por tanto el patrón de `Intercepting Filter`.



Img 21: Diagrama de secuencia del patrón Intercepting Filter.

2.9.2 . - Front Controller.

Ya vimos que Struts2 utiliza la clase `org.apache.struts2.dispatcher.filetr.StrutsPrepareAndExeuteFiler` como punto central de gestión de entrada de todas las peticiones.

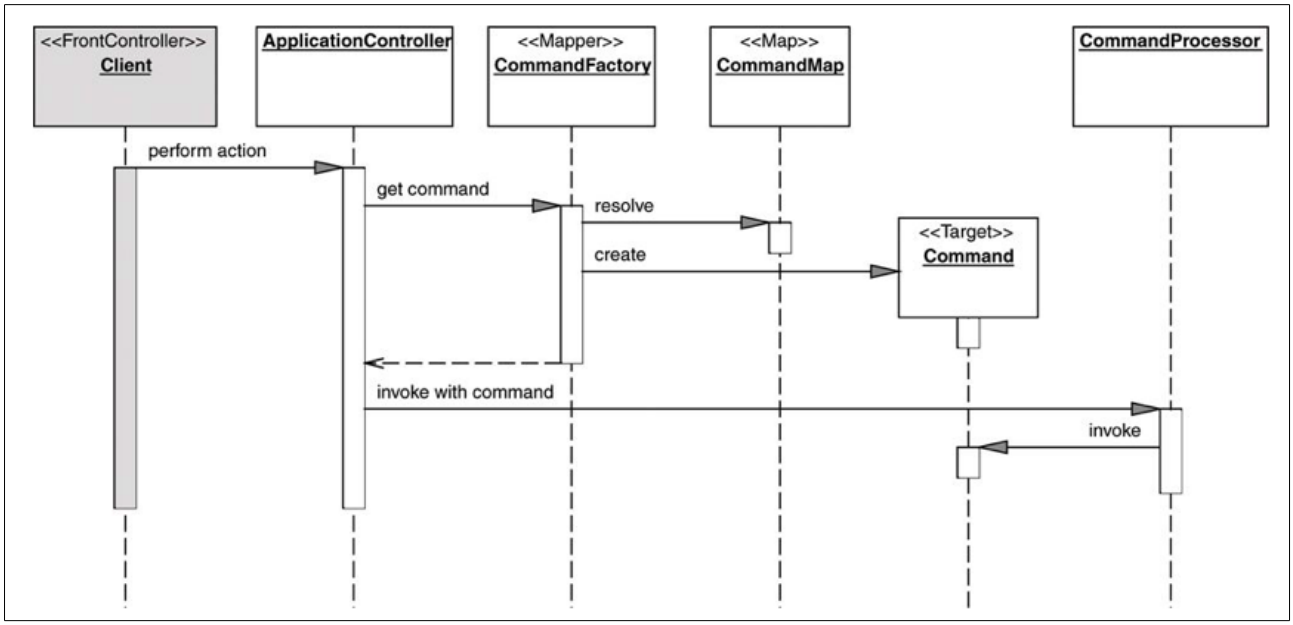
2.9.3 . - Application Controller.

Struts 2 dispone de las clases acción (aquellas que extiende `ActionSupport`) para dar servicio a las distintas peticiones de entrada. Para localizar la acción adecuada a una determinada petición se recurre, bien a:

- El fichero de configuración `struts.xml`
- El plugin de convenciones.
- Anotaciones.

Estas tres posibilidades establecen una relación entre el nombre de la acción, extraída de la URL de la petición y la clase acción a instanciar así como el método a invocar.

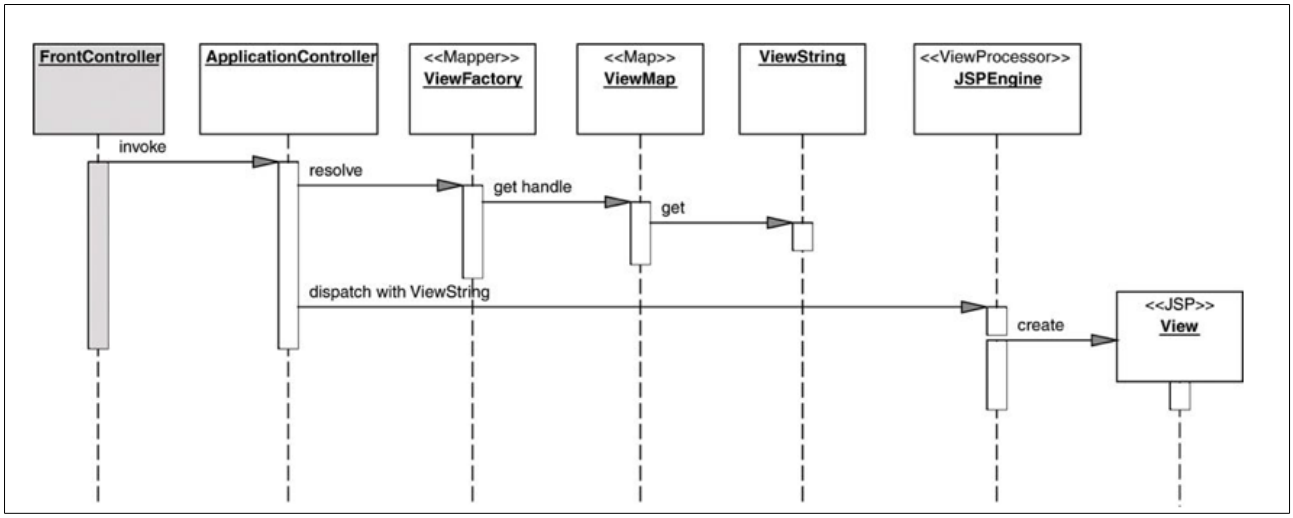
Se sigue por tanto la estrategia `Handler Command`.



Img 22: Estrategia Command Handler del patrón Application Controller.

Una vez invocada la acción para localizar la vista se hace uso del código `result` que devuelve la acción. Usando alguna de las tres técnicas anteriores se establece la relación entre el `result` y el nombre de la vista que se debe generar.

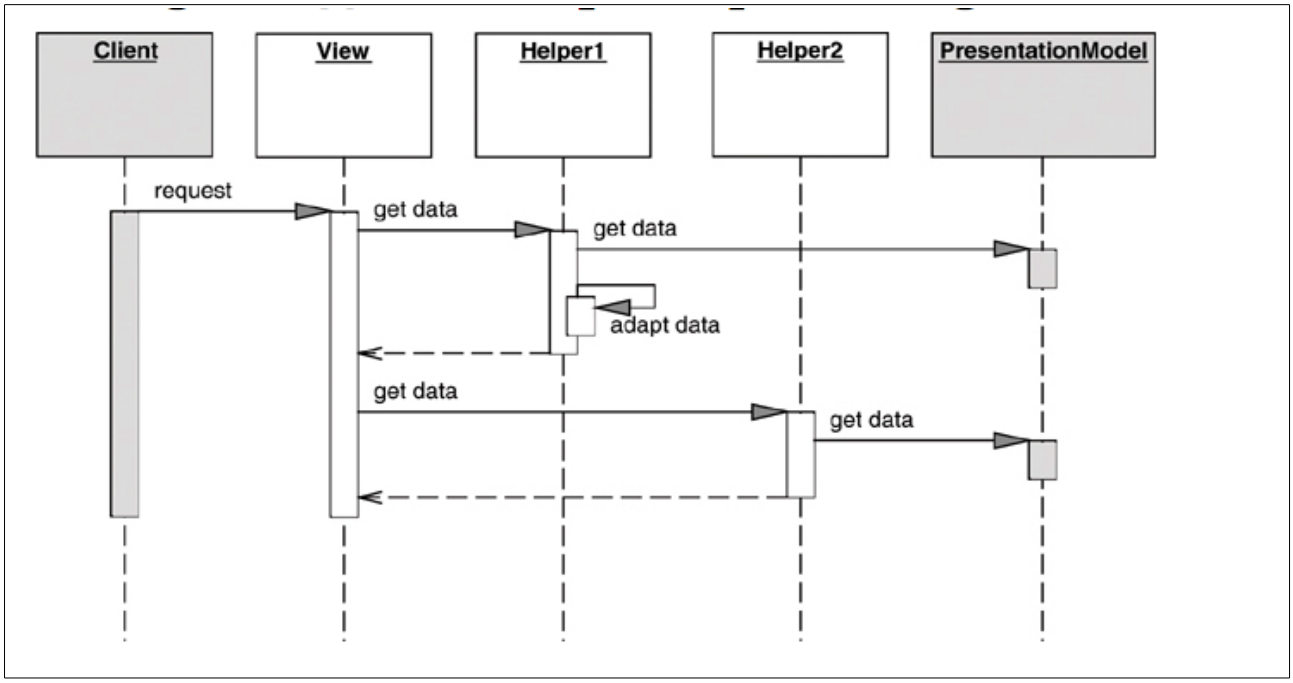
Se sigue por tanto la estrategia `View Handler`.



Img 23: Estrategia View Handler del patrón Application Controller

2.9.4 . - View Helper.

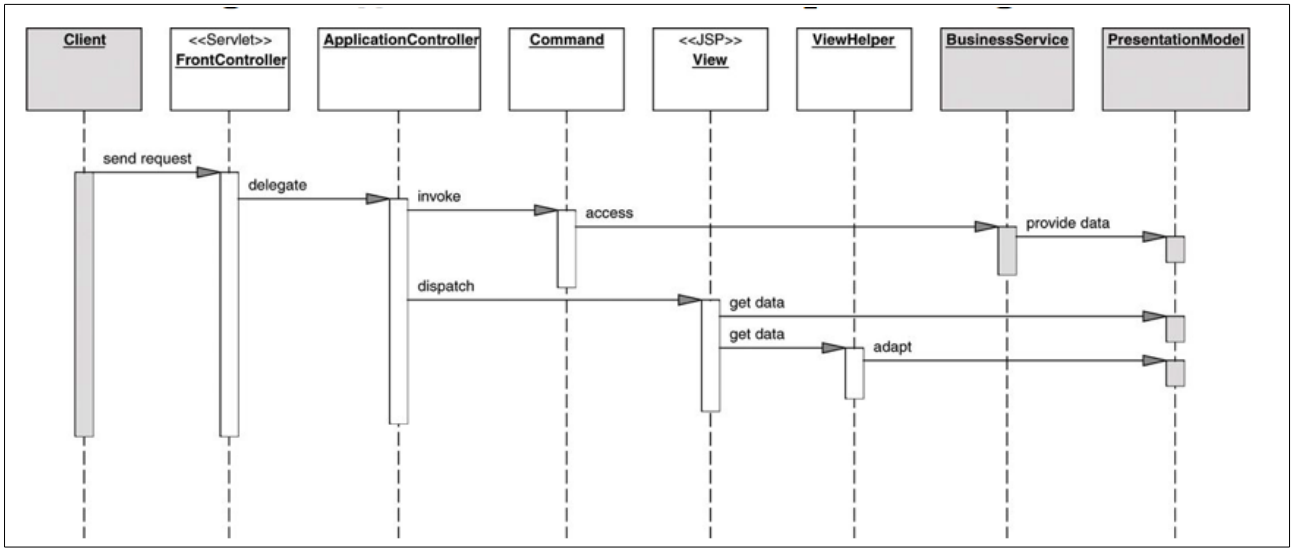
Struts2 genera sus vistas mediante páginas JSP con etiquetas para generar los distintos componentes con los que interactúa el usuario. Además uso POJOs para almacenar los valores de estos componentes a los que accede a través de expresiones OGNL. Por tanto establece una separación evidente entre las vista encargadas de la presentación y los datos que estas almacenas encapsulados en objetos de tipo Java. Se sigue por tanto el patrón View Helper con la estrategia JavaBean.



Img 24: Diagrama de secuencia del patrón View Helper.

2.9.5 . - Service To Worker.

Struts2 ante una petición delegará en las clases acción para invocar cualquier lógica de negocio y sólo después cuando la acción devuelva el código `result` cederá el control a la generación de la vista determinada por el `result`.



Img 25: Diagrama de secuencia del patrón Service to Worker.

3 . - JSF

Al igual que hicimos con Struts2 vamos a ir analizando el framework de JSF a medida que implementamos el componente login. E

3.1 . - Configuraciones previas.

Creamos un proyecto Maven con Eclipse sin arquetipos. En el POM realizamos las siguientes configuraciones:

- Indicamos la versión de java para compilar (1.8)
- Añadimos como dependencias las librerías necesarias para implementar un proyecto JSF.
  - `jsf-api` (API de JSF)
  - `jsf-impl` (Implementación de JSF)
  - `servlet-api` (Servlets)
  - `cdi-api` (para usar beans administrados)
- Añadimos el plugin maven de wildfly para desplegar en nuestro servidor.

Img 26: Dependencias para un proyecto web JSF.

3.2 . - Archivos de configuración.

JSF dispone de un servlet que actuará como controlador de entrada gestionando todas las peticiones. Debemos de configurarlo en el descriptor web.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://jav
3      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xm
4    <display-name>login-jsf</display-name>
5    <servlet>
6      <servlet-name>Faces Servlet</servlet-name>
7      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
8      <load-on-startup>1</load-on-startup>
9    </servlet>
10   <servlet-mapping>
11     <servlet-name>Faces Servlet</servlet-name>
12     <url-pattern>*.xhtml</url-pattern>
13   </servlet-mapping>
14   <welcome-file-list>
15     <welcome-file>index.xhtml</welcome-file>
16   </welcome-file-list>
17 </web-app>
```

Img 27: Descriptor web de una aplicación JSF.

Además dispone de un fichero de configuración propio `faces-config.xml` donde podemos:

- Establecer reglas de navegación.
- Indicar la localización de mensajes.
- Sobre escribir la anotaciones de los beans administrados.

De momento crearemos el archivo sólo con la cabecera.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <faces-config
3     xmlns="http://xmlns.jcp.org/xml/ns/javaee"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-f
6     version="2.2">
7 </faces-config>
```

Img 28: Fichero de configuración faces-config.xml

Ambos ficheros están localizados dentro de la carpeta WEB-INF.

3.3 . - Vistas Facelets y Beans Administrados.

Para la creación de las vistas JSF usa páginas XHTML con etiquetas propias para añadir componentes. En la siguiente tabla vemos las 4 librerías disponibles y el nombre de espacio donde se encuentran localizadas.

Librería	**Funcionalidad**	**Nombre de espacio.**
HTML	Componentes de entrada formulario.	`xmlns:h="http://xmlns.jcp.org/jsf/html"`
Core	Acciones personalizadas.	`xmlns:f="http://xmlns.jcp.org/jsf/core"`
Facelets	Crear plantillas.	`xmlns:ui="http://xmlns.jcp.org/jsf/facelets"`
Composite	Crear componentes compuestos.	`xmlns:cc="http://xmlns.jcp.org/jsf/composite"`

3.3.1 . - Etiquetas.

Veamos las principales etiquetas de la librería HTML

Etiquetas Entrada	
Etiqueta	**Funcionalidad**
`h:inputHidden`	Permite al autor de un página incluir una variable escondida.
`h:inputSecret`	Campo estándar de contraseña. Acepta una única línea de texto sin espacio y muestra asteriscos a medida que se introducen los caracteres.
`h:inputText`	Campo estándar de entrada. Acepta una única línea de texto.
`h:inputTextarea`	Campo estándar multilínea. Acepta múltiples líneas de texto.

Etiquetas Salida	
Etiqueta	**Funcionalidad**
`h:outputFormat`	Muestra un mensaje formateado.
`h:outputLabel`	Muestra un mensaje formateado.
`h:outputLink`	Muestra un enlace a otra página sin generar un evento de acción.
`h:outputText`	Muestra una línea de texto.

Etiquetas Acción	
Etiqueta	**Funcionalidad**

`h:commandButton`	Muestra un botón.
`h:commandLink`	Muestra un enlace.

Etiquetas Contenedor	
Etiqueta	**Funcionalidad**
`h:form`	Representa un formulario usando sus componentes anidados.
`h:panelGroup`	Agrupa todos sus componentes anidados.
`h:panelGrid`	Representa sus componentes anidados en forma de tabla.

Etiquetas Componentes de selección	
Etiqueta	**Funcionalidad**
`h:selectBooleanCheckBox`	Muestra una casilla de verificación.
`h:selectOneRadio` `h:selectManyRadio`	Muestra un conjunto de opciones en forma de botones.
`h:selectOneMenu` `h:selectManyMenu`	Muestra un menú desplegable de opciones.
`h:selectOneListBox` `h:slectManyListBox`	Muestra una lista de opciones.

La diferencia entre `One` y `Many` permitirá al usuario elegir una o más de una opción.
Para indicar las opciones disponemos de las siguientes etiquetas:

- `f:selecItem` – Representa una única opción.
- `f:selectItems` – Representa un conjunto de opciones.

Ambas tendrán como atributos `itemLabel` e `itemValue` para distinguir la forma en la que se muestran las opciones y en la que se guardan.
Un Item puede ser cualquier Objeto aunque JSF proporciona la clase `SelectItem`. Además el conjunto de opciones debe ser un objeto de tipo `Collection`.

Todas las etiquetas disponen de los siguientes atributos generales:

- `binding` – identifica una propiedad de un bean administrado con el que se vincula el componente.
- `id` – Identificador único del componente.
- `immediate` – Valor booleano para indicar si cualquier evento, validación o conversión asociada al componente se procesa en la fase temprana de aplicación de los valores de los parámetros.
- `rendered` – Especifica una condición para indicar si se genera o no el componente en la página.
- `style` – Especifica una hoja de estilo CSS para el componente.
- `styleClass` – especifica una clase CSS que define el estilo del componente.
- `value` – Indica el valor del componente en forma de una expresión de valor.

3.3.2 . - Beans administrados.

En JSF cada página XHTML se conecta con beans administrados que le sirven de respaldo. Desde la página podemos acceder a las propiedades y métodos del bean usando El (Expression Language).

EL nos permite:

- Lectura y escritura de datos almacenados en las propiedades del bean.
- Invocar métodos del bean.
- Realizar operaciones aritméticas, booleanas y de cadenas de String.

- Construcción y manipulación de colecciones de objetos.

Para el acceso tan sólo debemos usar la siguiente sintaxis:

- `#{bean.property}` → para leer o escribir la propiedad property del bean Bean.class.
- `#{bean.method}` → para invocar el método method del bean Bean.class

Para indicar que un POJO es un bean administrado debemos realizar las siguientes anotaciones en su clase.

- @Named
- @SessionScoped

Además debe implementar la interfaz `Serializable`.

Vemos que que EL usa el mismo nombre de la clase del POJO pero en minúscula. Si quisiéramos usar otro nombre se lo indicaríamos en la anotación `@Named`.

En nuestro ejemplo para implementar la vista `login.xhtml` necesitaríamos las siguientes etiquetas:

- `h: ipuntText` para el usuario.
- `h: inputSecret` para la contraseña.
- `h:commandButton` para el botón de entrada.
- `h:panelGrid` para mostrar los componentes en tabla.
- `h: form` para agrupar los componentes del formulario.

Además para almacenar los valores de usuario y contraseña crearemos el bean administrado `LoginBean`.

```
<h:form>
<h:panelGrid columns="2">
    <h:outputLabel for="userId">Usuario:</h:outputLabel>
    <h:inputText id="userId" value="#{loginBean.user}" />

    <h:outputLabel for="passId">Contraseña</h:outputLabel>
    <h:inputSecret id="passId" value="#{loginBean.pass}"/>

    <h:commandButton value="Acceder" action="#{loginBean.acceder}" />
</h:panelGrid>
</h:form>
```

Img 29: Formulario Login con JSF.

```
12 @Named
13 @SessionScoped
14 public class LoginBean implements Serializable {
15
16     private static final long serialVersionUID = -8584780627456838114L;
17
18     private String user;
19     private String pass;
```

Img 30: Bean administrado LoginBean para dar respaldo a la vista login.xhtm

En la vista registro necesitamos dos etiquetas de selección:

- `h:selectOneMenu` para mostrar una lista desplegable para seleccionar el IES donde trabaja el profesor.
- `h:selectManyCheckBox` para mostrar una lista de casillas de selección para indicar los cursos que tiene el profesor.

Para la primera opción se decide usar `f:selectItem` indicando en el propio formulario la etiqueta y descripción de cada IES.

```
<h:selectOneMenu id="schoolId" value="#{person.school}">
    <f:selectItem itemValue="1" itemLabel="IES Güímar"></f:selectItem>
    <f:selectItem itemValue="2" itemLabel="IES PuntaLarga"></f:selectItem>
    <f:selectItem itemValue="3" itemLabel="IES Santa Ana"></f:selectItem>
</h:selectOneMenu>
```

Img 31: Ejemplo de selecItem en el formulario login.

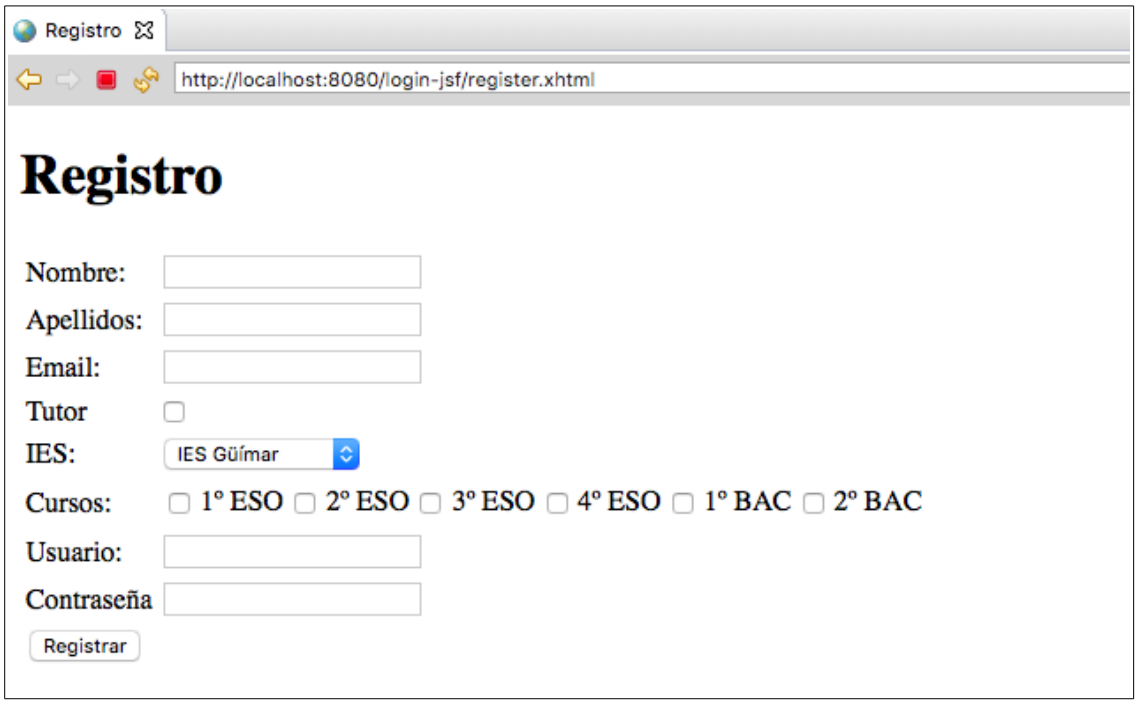
Para la segunda opción se opta por `f:selectItems` indicando las opciones mediante el bean administrado `registerBean` que da respaldo a la vista `register.xhtml` y contiene un lista de objetos `Level` con el código y la descripción de los distintos cursos.

```
30        <h:selectManyCheckbox id ="levelsId" value="#{registerBean.levels}">
31            <f:selectItems value="#{registerBean.levelsAvailable}" var="l"
32                itemValue="#{l.code}" itemLabel="#{l.description}" />
33        </h:selectManyCheckbox>
```

Img 32: Ejemplo de selectItems en el formulario login.

```
28    public RegisterBean() {
29        super();
30        levelsAvailable = new Level[6];
31        levelsAvailable[0] = new Level("1E", "1º ESO");
32        levelsAvailable[1] = new Level("2E", "2º ESO");
33        levelsAvailable[2] = new Level("3E", "3º ESO");
34        levelsAvailable[3] = new Level("4E", "4º ESO");
35        levelsAvailable[4] = new Level("1B", "1º BAC");
36        levelsAvailable[5] = new Level("2B", "2º BAC");
37    }
```

Img 33: Colección de objetos Level en para indicar las opciones de cursos en RegisterBean.



Img 34: Vista Registro.

3.4 . - Implementando eventos de acción.

Para implementar la acción de `h:commandButton` o `h:commandLink` debemos usar el atributo `action` y referirnos a un método del bean administrado que ejecute la acción. Además el método devolverá un `String` que dirá a la aplicación cuál es la siguiente vista a mostrar.

En nuestro ejemplo queremos que el botón de entrada a la plataforma valide el usuario y contraseña del formulario login. En caso de que sea válido iremos a la página UOC dando la bienvenido al usuario a la plataforma. Pero si no es válido se volverá a mostrar la página login añadiendo un mensaje de error.

Implementaremos el método acceder en el bean administrado `loginBean`.

```
public String acceder() {
    if(service.validar(new Credential(this.user, this.pass)))
        return ("uoc");
    else {
        FacesContext.getCurrentInstance()
                    .addMessage(null, new FacesMessage("Usuario o contraseña no válido"));
        return("login");
    }

}
```

Img 35: Método que implementa la acción del botón acceder del formulario login.

3.5 . - Validación.

Parar validar los valores de los componentes de entrada JSF dispone de las siguientes alternativas:

3.5.1 . - Validadores por defecto.

En la siguiente tabla mostramos los principales validadores estándares disponibles.

Etiqueta	Funcionalidad
`f:validateLength`	Comprueba que el valor del componente tiene una determinada longitud.
`f:validateDoubleRange`	Comprueba que el valor del componente está en un determinado rango.
`f:validateRegex`	Comprueba que el valor del componente verifica una expresión regular.
`f:validateRequired`	Comprueba que el valor del componente no está vacío.

Para registrarlos en un componente debemos anidar la correspondiente etiqueta.

Todas etiquetas están asociadas a las correspondientes clases que deben implementar la interfaz `Validator`. Además todos ellos tienen su correspondiente mensaje de error.

En nuestro ejemplo en el formulario login registramos lo siguientes validores:

- Tamaño mínimo de 6 en los campos usuario y contraseña.
- Patrón mínimo un carácter no alfanumérico en el campo contraseña.

Para que los campos sean requeridos en lugar de validadores utilizamos el atributo `required` del componente con su correspondiente mensaje de error. Además utilizamos el atributo `label`

para saber que campo tiene errores de validación. Por último añadimos etiqueta `h:messages` para mostrar todos los errores de validación al final de la página.

```
21          <h:outputLabel for="passId">Contraseña</h:outputLabel>
22          <h:inputSecret id="passId" value="#{loginBean.pass}"  label="Contraseña"
23              required="true" requiredMessage="Introduce una contraseña">
24                  <f:validateLength minimum = "6" />
25                  <f:validateRegex pattern=".*\W.*"/>
26          </h:inputSecret>
```

Img 36: Uso de validadores estándar en el componentes contraseña.

3.5.2 . - Método de un Bean Administrado.

Podríamos implementar la validación mediante un método de un bean administrado y referirnos a él mediante EL usando el atributo `validator` en la etiqueta del componente.

El método que implementa la validación debe aceptar los siguiente parámetros:

- `FacesContext context` → Objeto que contienen toda la información de la petición.
- `UIComponent toValidate` → Componente de entrada cuyo valor queremos validar.
- `Object value` → Objeto que queremos validar.

En nuestro ejemplo para validar que hayamos seleccionado al menos un curso en el formulario registro implementaremos el siguiente método en bean administrado `registerBean`.

```
43      public void validateLevels(FacesContext context, UIComponent toValidate, Object value) {
44          if(levels == null || levels.length == 0) {
45              ((UIInput)toValidate).setValid(false);
46              context.addMessage(toValidate.getClientId(),
47                      new FacesMessage("Selecciona al menos un curso"));
48          }
49      }
```

Img 37: Método que implementa la validación de seleccinar al menos una opción.

Además lo registramos en el componente de lista de opciones cursos usando el atributo `validator`.

```
36          <h:outputLabel for="levelsId">Cursos:</h:outputLabel>
37          <h:selectManyCheckbox id ="levelsId" value="#{registerBean.levels}"
38              validator="#{registerBean.validateLevels}">
39                  <f:selectItems value="#{registerBean.levelsAvailable}" var="l"
40                      itemValue="#{l.code}" itemLabel="#{l.description}" />
41          </h:selectManyCheckbox>
```

Img 38: Uso del atriuto validator para referenciar el método de valadción.

3.5.3 . - Implementar un Validador personalizado.

Se puede crear un validador creando una clase que implementa la interfaz `Validator`.
Una implementación de la interfaz `validator` debe tener:

- Un constructor.
- Métodos de acceso (`setters/getters`) para todos los atributos que queramos que tenga la etiqueta de validación.
- Método `validate`.
- La anotación `@FacesValidator` para registrar el validador en la aplicación.

Para registralo en un componente debemos anidar la etiqueta `f:validator` y usar el atributo `binding` indicando la clase que implementa la validación. También podemos crearnos una

etiqueta personalizada usando un archivo xml que sea el descriptor de etiquetas de la aplicación.

3.6 . - Ciclo de vida de una aplicación JSF.

Desde que el cliente realiza una petición HTTP hasta que el servidor responde con una página traducida a HTML, JSF como framework de presentación debe encargarse de las siguientes tareas:

- Obtener los parámetros de la petición de entrada.
- Completar los valores de los componentes de la vista con estos parámetros.
- Realizar las conversiones y validaciones registradas en los componentes.
- Invocar la lógica de negocio..
- Localizar y generar la siguiente vista a entregar al navegador.

JSF realiza de forma casi automática cada una de estas tareas. Para ello dispone de:

- Una instancia de `FacesContext` donde que contiene toda la información necesaria para procesar la petición.
- Una estructura de árbol de componentes llamada view que le permite dar soporte a las vistas.

JSF realiza estas tareas en un orden estricto pasando la aplicación por una serie de fases qie detallamos a continuación.

3.6.1 . - Fase de restauración de la vista.

JSF construye el árbol de componentes de la página, registra las validaciones y los eventos de cada componente y guarda la vista en una instancia de `FacesContext`.
Si la petición a la página es la inicial, la vista creada estará vacía ya que no habrá ninguna entrada del usuario ni acción a procesar. JSF llamará al método `FacesContext.renderResponse` y saltará a la fase de generación de la respuesta.

3.6.2 . - Fase de aplicación de valores de la petición.

Mediante su método `decode` cada componente extrae de los parámetros de la petición un nuevo valor que almacenan localmente.
Si algún componente tiene el atributo `immediate` con valor `true`, las validaciones, conversiones y eventos asociados a ese componente serán procesados en esta fase.

3.6.3 . - Fase de proceso de validación.

Se procesan todos los validadores registrados en los componentes del árbol mediante el método `validate` usando el valor almacenado localmente. Se completan todas las conversiones de los componentes `input` que no tenían el atributo `immediate` con valor `true`.

3.6.4 . - Fase de actualización del modelo de datos.

Se actualizan las propiedades de los beans administrados con los atributos `value` de los componentes `input`. Si el valor local del componente no puede ser convertido al tipo de datos de la propiedad del `bean`, se avanza a la fase generación donde se genera la misma página mostrando los errores de conversión.

3.6.5 . - Fase de invocación.

Se gestiona los eventos a nivel de aplicación como pueden ser un `submit` a un formulario o enlazar a una página.

3.6.6 . - Fase de generación de la respuesta.

Si se trata de una petición inicial se añaden a la vista los componentes referenciados por las etiquetas de la página.
Se delega la autoridad al recurso apropiado para generar la página. En caso de que se hayan producido errores en las fases de ejecución se vuelve a generar la misma página. Si la página contiene las etiquetas `h:message` o `h:messages` todos los errores que se hayan puesto en cola en las fases anteriores se mostrarán.
Después de que la vista haya sigo generada, el estado de la respuesta es salvado para que las siguientes peticiones puedan acceder a ella.

El ciclo de vida normal puede verse alterado de dos formas:
Llamando al método `FacesContext.renderRespone()` que hará avanzar el ciclo directamente a la fase de generación. Este método puede ser invocado por medio de:

- El método `decode` de algún componente.
- Un `listener` asociado a un evento registrado en algún componente.
- El método `validate` de algún componente cuando falla la validación.
- Un convertidor registrado en un componente si falla la conversión.
- El valor de un atributo de un componente no puede ser convertido al tipo de una propiedad del `bean` administrado.

3.7 . - Patrones empleados por JSF.

3.7.1 . - Front Controller.

JSF utiliza la clase `javax.faces.webapp.FacesServlet` como punto central de gestión de entrada de todas las peticiones.

3.7.2 . - Context Object.

JSF dispone de la clase FacesContext que contiene toda la información de estado relacionada con una petición.

3.7.3 . - View Helper.

JSF utiliza beans administrados para dar soporte a sus vistas, pudiendo sus etiquetas acceder a las propiedades del bean gracias al uso de EL. Además son estos beans los que pueden acceder a la lógica de negocio por medio de alguno de sus métodos. Se sigue por tanto el patrón View Helper con la estrategia JavaBean.

3.7.4 . - Service To Worker.

JSF delega en los métodos de los beans administrado para invocar la lógica de negocio. Estos métodos devuelven un String con el nombre de la vista a generar. Se sigue por tanto el patrón de sólo ceder la generación de la vista una vez invocada la lógica de negocio. Además su resultado determinará la vista a presentar.

4 . - Vaadin.

4.1 . - Arquitectura Vaadin.

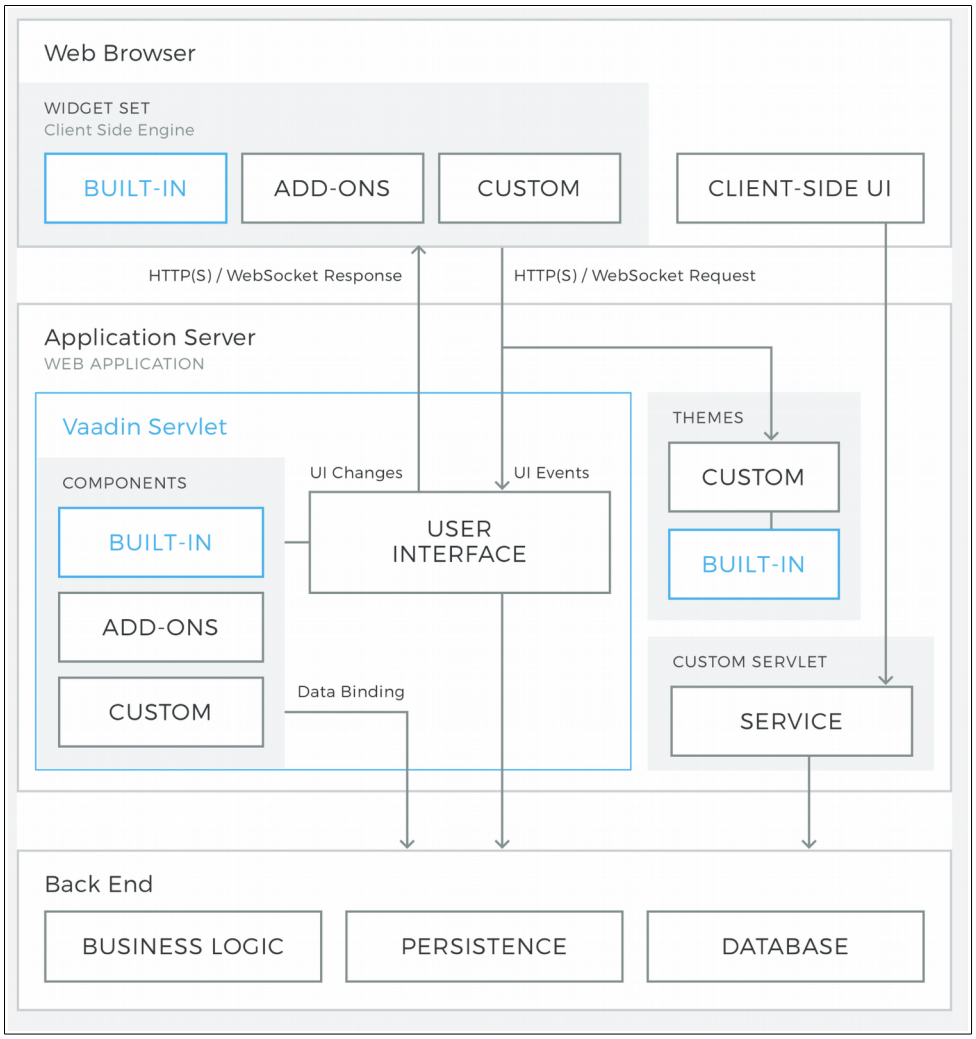
Vaadin proporciona dos modelos para el desarrollo de aplicaciones web:

- Modelo lado servidor.

Permite aplicaciones desarrolladas sólo en el lado del servidor usando el motor cliente Vaadin basado en Ajax que genera la interfaz de usuario en el navegador.

- Modelo lado cliente.

Permite el desarrollo de widgets y aplicaciones Java que son compiladas en JavaScript y ejecutadas en el navegador.



Img 39: Arquitectura de Vaadin.

El framework de Vaadin consiste en una serie de APIs, una multitud de componentes/widgets en ambos lados, temas para controlar la apariencia y un modelo de datos que permite vincular los componentes directamente con los datos.

Las aplicaciones en el lado del servidor se ejecutan en un contenedor web. `VaadinServlet` es la clase `servlet` encargada de servir las peticiones HTTP que las interpreta como eventos de una sesión de usuario. Los eventos están asociados con componentes UI (Interfaz de Usuario) y se entregan a los correspondientes oyentes de la aplicación. Si la lógica de la aplicación hace cambio en los componentes UI, el `servlet` los genera en el navegador como respuesta. El

motor cliente que corre en el navegador recibe la respuesta y hace los cambios necesarios en la página del navegador.

Los principales elementos de la arquitectura y sus funcionalidades son las siguientes:

4.1.1 . - UI (Interfaz de usuario).

La interfaz de usuario se implementa extendiendo la clase `com.vaadin.ui.UI`. Su principal tarea es la de crear la interfaz inicial, añadiendo los componentes UI y estableciendo los oyentes de eventos que gestionan las entradas de usuario.

4.1.2 . - Componentes/Widgets UI.

Los componentes son clases que implementan parte de la interfaz. Cada componente tiene su pareja `widget` en el lado cliente encargado de generarlo en el navegador para que el usuario interaccione con él. Estos componentes usan los eventos para realizar la lógica de la aplicación.

4.1.3 . - Motor del lado cliente.

Es el encargado de generar la interfaz en el navegador usando los widgets correspondientes de los componentes. Comunica la interacción del usuario al servidor y generan los cambios producidos en los componentes. Estas comunicaciones se realizan usando HTTP ó HTTPS asíncrono.

4.1.4 . - Servlet de Vaadin.

El `Servlet` está implementado por la clase `VaadinServlet`. Recibe las peticiones de diferentes clientes, determina a que sesión de usuario pertenece y delega la petición a esa sesión.

4.1.5 . - Temas.

Vaadin distingue claramente entre la apariencia y la estructura de un componente UI. Mientra la lógica se implementa en código Java, la presentación se define con temas con `CSS` o `Sass`. Vaadin proporcionan una serie de temas por defecto que incluyen hojas de estilo, plantillas HTML que definen distribuciones personalizadas y otros recursos como imágenes y fuentes.

4.1.6 . - Eventos.

La interacción del usuario con la interfaz crea eventos, que son primeramente procesados por los widgets del lado cliente para luego pasar a través del `servlet` de Vaadin a los componentes UI que delegan en los oyentes correspondientes de la aplicación.

4.1.7 . - Push del servidor.

Vaadin soporta push (empuje) del servidor. Esto significa que cualquier cambio en los componentes UI son directamente trasladados al cliente sin necesidad de ningún evento o petición cliente.

4.1.8 . - Vinculación de datos.

Vaadin proporciona un modelo de datos para vincular directamente valores de los componentes campo con las fuentes de datos. De esta manera se pueden actualizar los datos sin necesidad de ningún código de control.

4.1.9 . - Aplicaciones lado cliente.

Vaadin también soporta módulos de aplicaciones en modelo cliente que corre en el servidor. Pueden hacer uso de los widgets, temas y back-end. Son útiles si necesitamos aplicaciones con un alto grado de respuesta.

4.1.10 . - Back-end.

La lógica del negocio suele ejecutarse en el mismo `servlet` que el código UI aunque separado por al menos un API Java (posiblemente EJBs) o distribuida en un servicio back-end. El almacenamiento de datos se distribuye en gestores de base de datos a los que se accede mediante una solución de persistencia como JPA.

4.2 . - Aplicaciones del lado del servidor.

Las aplicaciones Vaddin del lado del servidor se ejecutan en contenedor de `servlets`. La interfaz de usuario se implementa con la clase UI que es la encargada de crear y gestionar todos los componentes de la interfaz. Las interacciones de los usuarios se gestionan con oyentes de eventos y es posible también vincular los componentes directamente con los datos. El estilo visual de la aplicación se define con temas como CSS ó Sass. Los iconos, imágenes y ficheros descargables se gestionan como recursos que pueden ser externos o servidos por la aplicación o el propio servidor.

Los principales elementos de una aplicación Vaadin son los siguientes:

4.2.1 . - UI.

La clase UI representa un fragmento de HTML dentro de la página web. Normalmente la aplicación Vaadin se desarrolla extendiendo la clase UI y añadiéndole contenido.

4.2.2 . - Page.

Representa tanto la página web como la ventana del navegador en la que se ejecuta UI.

4.2.3 . - VaadinSession.

Representa una sesión de usuario con una o más UIs abiertas en la aplicación.

4.2.4 . - Componentes UI.

La interfaz de usuario consiste en componentes que son creados por la aplicación. Se distribuyen jerárquicamente usando componentes de tipo layout, con un contenido raíz en la cima de la jerarquía. La interacción de los usuarios con estos componentes causan eventos que la aplicación gestiona.

4.2.5 . - Eventos y oyentes.

Vaadin sigue el paradigma de programación dirigido por eventos. Los oyentes deben gestionar los eventos que se disparan por la interacción del usuario con los componentes. Además se hace uso del push del servidor para trasladar al cliente los cambios que la lógica de la aplicación realiza en los componentes.

4.2.6 . - Recursos.

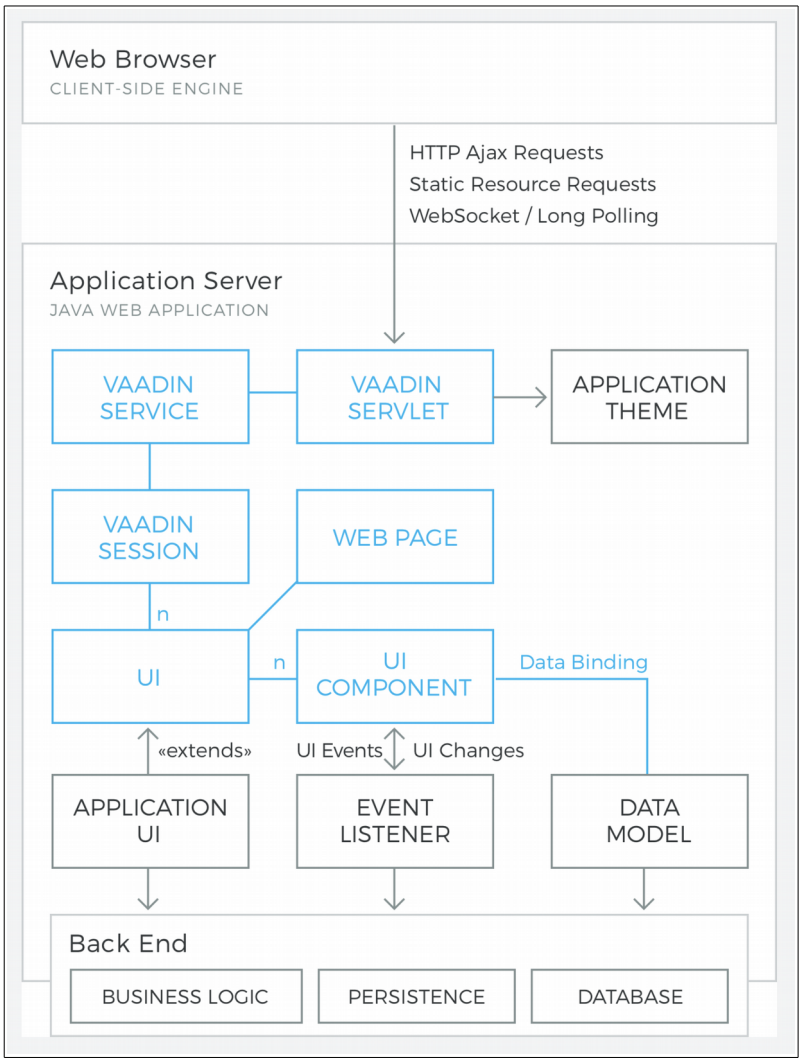
Las imágenes, los enlaces a páginas web así como los documentos descargables son gestionados por los recursos.

4.2.7 . - Temas.

La lógica y la presentación de la interfaz de usuario está claramente separada. Mientras la lógica es implementada por el código Java, la presentación se define en temas como CSS o Sass. Vaadin incluye temas incorporados pero el usuario también puede definirlos usando hojas de estilos, plantillas HTML con distribuciones personalizadas e incluyendo recursos como imágenes.

4.2.8 . - Vinculación de datos.

Los componentes campo son esencialmente vistas de datos. Con el modelo de datos de Vaadin los componentes obtienen y actualizan sus valores de forma directa sin necesidad de ningún código de control.



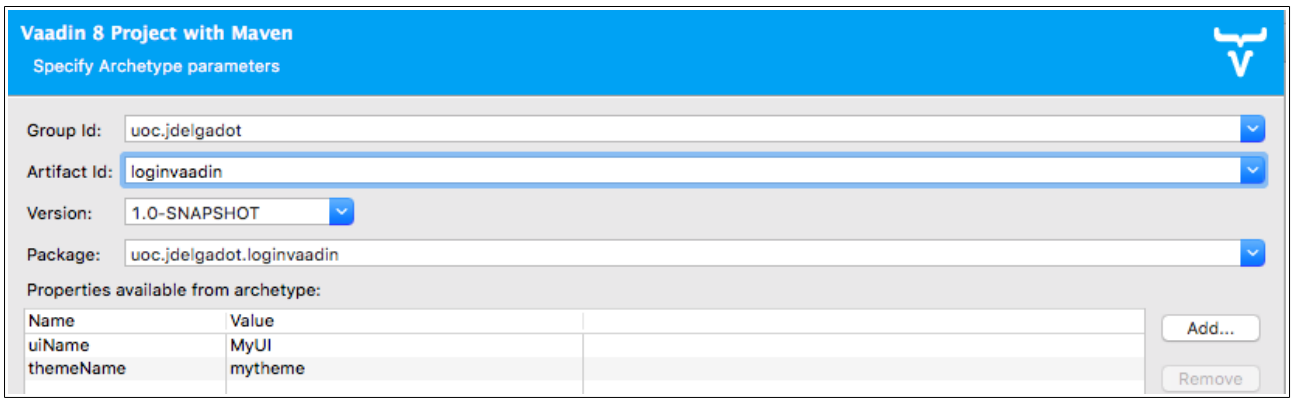
Img 40: Arquitectura aplicación.

4.3 . - Configuraciones previas.

En esta ocasión crearemos el proyecto a partir del plugin Vaadin para Eclipse. Al observar el POM vemos que son necesarias las siguientes librerías:
Observamos las siguientes librerías que necesita en el POM:

- `vaadin-server`: principal librería del framework.
- `Vaadin-client-compiled`: el motor precompilado de Vaadin del lado del cliente que incluye el widget por defecto.
- `vaadin-themes`: Incluye los ficheros fuentes SCSS y los precompilados CSS para temas Vaadin.

- `vaadin-push`: Realiza llamadas al servidor para refrescar el cliente.
- `javax-servlet-api`: Funcionalidades de servlet que proporcionará el servidor de aplicaciones.



Img 41: Creación de un proyecto con el plugin de Vaadin para Eclipse.

Además vemos en la sección `build` que:

- Al empaquetar como war se excluyen las clases `widgetset` de vaadin y no es necesario un descriptor web.
- Se ejecutan fases al compilar propias de vaadin para los `widgetset` y los temas.

Al echar un vistazo a la única clase Java que nos da el arquetipo vemos que es el punto de entrada a la aplicación ya que extiende la clase `UI`. Dispone además del método `init` que iniciará la petición.

```
22 @Theme("mytheme")
23 public class MyUI extends UI {
24
25     @Override
26     protected void init(VaadinRequest vaadinRequest) {
```

Img 42: Clase UI de un proyecto Vaadin.

En la parte final vemos la clase que extiende el servlet de Vaadin.

```
42
43     @WebServlet(urlPatterns = "/*", name = "MyUIServlet", asyncSupported = true)
44     @VaadinServletConfiguration(ui = MyUI.class, productionMode = false)
45     public static class MyUIServlet extends VaadinServlet {
46     }
```

Img 43:

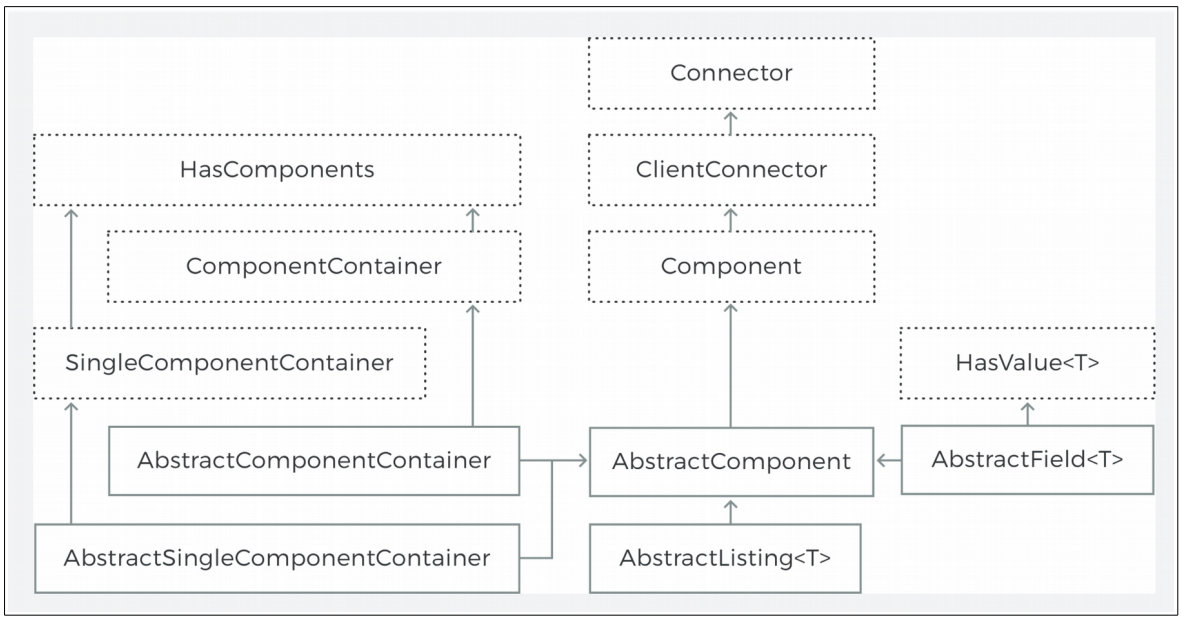
4.4 . - Componentes UI.

4.4.1 . - Interfaces y abstracciones.

Los componentes UI de Vaadin se construye sobre un esqueleto de interfaces y clases abstractas que definen e implementan las funcionalidades comunes a todos los componentes y la lógica básica para serializar su estado entre el servidor y el cliente.

En la cima de esta estructura se encuentra la interfaz `Component` y la clase abstracta `AbstracComponent` que implementa todos los métodos de la interfaz. A partir de ella tenemos clases abstractas más especializadas como:

- `AbstractComponentContainer` para componente contenedores.
- `AbstarctField` pata componentes que almacenan algún tipo de dato.



Img 44: Interfaces y abstracciones de los componentes Vaadin.

4.4.2 . - Características comunes.

La clase base de componente y su interface define un número de propiedades que podemos obtener y manipular a partir de los `getter` y `setter`. Veamos en la siguiente tabla la funcionalidad de cada característica.

Característica	Funcionalidad
`caption`	Leyenda del componente que se muestra encima, a la izquierda o dentro del componente.
`description`	Descripción que se muestra como ventana emergente al pasar el ratón sobre el componente.
`enable`	Controla si el usuario puede actuar sobre el componente.
`icon`	Etiqueta gráfica que se muestra encima. a la izquierda o dentro del componente.
`locale`	Indica el país y lenguaje usado por el componente.
`readOnly`	Define si el valor del componnete puede ser cambiado. Sólo es aplicable a los componentes que implementen la interfaz HasValue.
`styleName`	Define el nombre del estilo personalizado para el componente.
`visible`	Permite esconder el componente en el navegador.
`size`	La interfaz `Sizeable`, permite manipular el ancho y el alto de los componentes en unidades tanto absolutas o relativas. El tamaño se establece con los métodos `setWidth` y `setSize` que requiere de dos parámetros (el valor del tamaño y su unidad). También es posible indicar un único parámetro en formato `String` (“400px” ó “100%”).

4.5 . - Componentes layouts.

Son aquellos componentes que determinan la colocación de los demás componentes en la interfaz de usuario. Deben de tener una estructura de árbol. Por tanto debemos empezar por crear un componente `Layout` que será la raíz de nuestra clase UI. Para ello hacemos uso del método `setContent`. Posteriormente mediante `addComponent` iremos añadiendo el resto de componentes.
Veamos en algo más de detalle alguno de estos componentes:

4.5.1 . - VerticalLayout / HorizontalLayout.

Permiten disponer los componentes de forma horizontal o vertical. Mientras `VerticalLayout` tiene un % de ancho por defecto y aun alto indefinido; `HorizonatlLayout` tiene ambas dimensiones indefinidas.
Los principales atributo son:

- `comoponentAligment` – Indica el tipo de alineación entre componentes (`left, center, right, top, middle, bottom`).
- `spacing` – indicador booleano para saber si queremos espacio entre los componentes.
- `margin` - indicador booleano para saber si queremos un margen.
- `expandRation` – indica el ratio de expansión de un componente respecto al total.

4.5.2 . - FormLayout.

Dispone los componentes y sus leyendas en dos columnas, con los indicadores de campos requeridos y errores para ser mostrados en cada campo.

4.5.3 . - GridLayout.

Dispone los componentes en una rejilla definida por el número de filas y columnas. El número de fila y columna sirven de coordenadas para colocar un componente que puede ocupar más de una fila y/o columna.

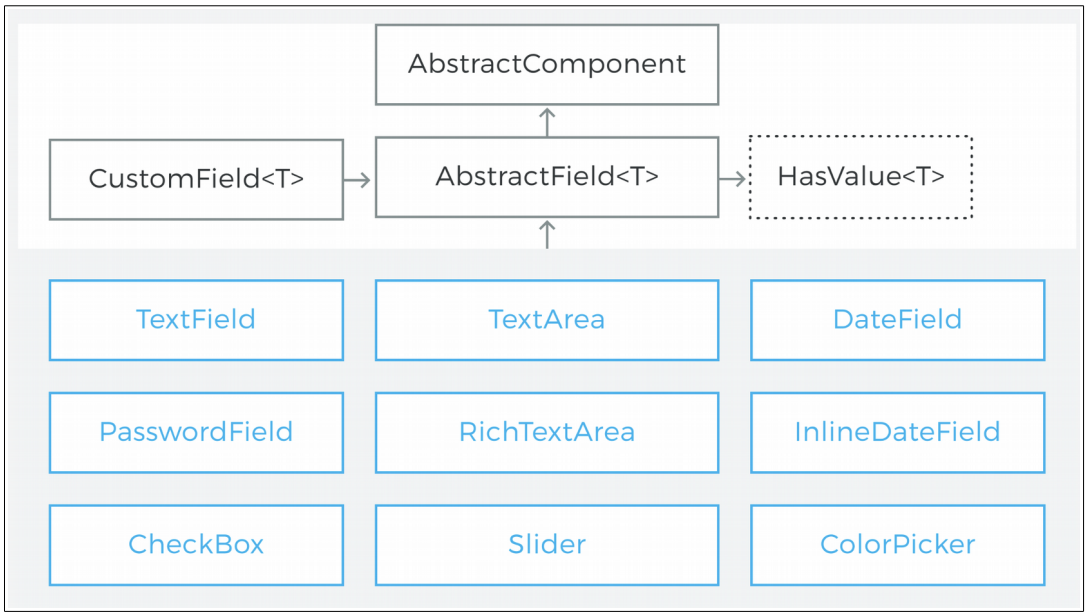
4.5.4 . - Panel.

Sólo puede contener un único componente (que se añade con el método `setContent`) que se muestra como un marco alrededor del contenido. Opcionalmente puede tener una leyenda y un icono que son gestionado por el propio Panel.

En nuestro ejemplo implementaremos los formularios Login y Register extendiendo la clase `FormLayout`.

4.6 . - Componentes Field.

Los componentes `Fields` son aquellos que tienen un valor que el usuario puede cambiar a través de la interfaz.
Todos los componentes `Field` se construyen a partir de la interfaz `HasValue` y la clase abstracta que implementa todos sus métodos `AbstractField`.



Img 45: Estructura de las clases componentes de campo.

La interfaz `HasValue` marca al componente como editable por parte del usuario. El tipo de parámetro en la interfaz es el tipo de valor que el componente está editando. Podemos establecer el valor con el método `setValue()` y leerlo con `getValue()`.

Además esta interfaz define las siguientes propiedades accesible con `getters` y `setters`:

- `redOnly` → establece el valor del componente como no editable.
- `requiredIndicatorVisible` → muestra un asterisco para indicar que el campo no puede estar vacío.
- `emptyValue` → El valor inicial de un campo vacío.
- `clear` → Borra el valor y lo deja con el valor vacío.

La interfaz `HasValue` proporciona el método `addValueChangeListener` como oyente que procesa los cambios en los valores de los componentes.

En la siguiente tabla mostramos la funcionalidad los principlaes compoentes de campo Vaadin.

Clase componente	Funcionalidad
`TextField`	Entrada de texto mediante teclado.
`TextArea`	Versión multilínea de `TextField`.
`PasswordField`	Variante de `TextField` que esconde la entrada con asteriscos.
`DateField`	Introduce y muestra fecha con un campo de entrad numérico y un calendario desplegable.
`CheckBox`	Casilla de verificación. Se trata de un componente que almacena un valor booleano.

Para establecer la longitud máxima del campo de texto tenemos el método `setMaxLegth()`.

Para gestionar los valores nulos disponemos de los siguientes métodos:

- `setNullSettingAllowed()` - Permite los valores nulos.

- `SetNullRepresentation()` - Indica cómo mostrar el valor nulo.

Para crear el formulario Login añadimos los siguientes componentes:

- `Label` – Para el título del formulario.
- `TextField` – Para la entrada de usuario.
- `PasswordField` – Para la entrada de la contraseña.

Añadimos indicadores requeridos e iconos a los campos.

```
19      public Login() {
20          this.setMargin(true);
21          label = new Label("ACCESO A PLATAFORMA DE ENTRADA");
22          user = new TextField("Usuario");
23          pass = new PasswordField("Contraseña");
24          user.setRequiredIndicatorVisible(true);
25          user.setIcon(VaadinIcons.USER);
26          pass.setRequiredIndicatorVisible(true);
27          pass.setIcon(VaadinIcons.PASSWORD);
28          this.addComponents(label, user, pass);
29
30      }
```

Img 46: Constructor de la clase que implementa el formulario Login.

4.7 . - Componentes selección.

Son aquellos componentes que permiten seleccionar entre una o más ítems. Todos ellos basados en las clases `AbstractSingleSelect` ó `AbstractMultiSelect`.

Veamos en la siguiente tabla las principales clases y funcionalidades de los componentes selección.

Clase componente.	**Funcionalidad.**
`ComboBox`	Lista desplegable con un campo de texto donde el usuario puede escribir para encontrar ítem y seleccionar uno. El usuario puede entrar nuevos ítems.
`ListSelect`	Lista vertical para seleccionar uno o múltiples ítems.
`NativeSelect`	Usa el componentes de selección propios del navegador. Normalmente es una lista desplegable para selección única y lista multilínea para selección múltiple.
`RadionButtonGroup`	Muestra los ítems agrupados verticalmente como botones para seleccionar uno.
`CheckBoxGroup`	Muestra los ítems agrupados verticalmente como casillas de verificación para seleccionar uno o varios.
`TwinColSelect`	Muestras dos columnas donde el usuario puede ir seleccionando los ítems y moverlo de un a otro lado mediante botones de control.

Los ítems son normalmente `String`, en cuyo caso pueden ser usados como leyenda de las opciones, pero pueden ser cualquier tipo de objeto.
Disponemos de los siguientes métodos para establecer las opciones (forma en la que se guardan los ítems) y sus leyendas (forma en la que se muestran):

- `setItemCaptionGenerator(ItemCaptionGenerator<T> itemCaptionGenerator)`

Establece un generador de leyendas de ítems para producir `String` que muestren las opciones del componente.

- `setItems(Collection<T> items)`

Establece los ítems del componente mediante una colección de objetos.

En nuestro formulario Register tendremos dos campos de selección:

- `ComboBox` → Lista desplegable para seleccionar el IES donde trabaja el profesor.
- `CheckBoxGroup` → Grupo de casillas para seleccionar los niveles que imparte el profesor.

Para el `ComboBox` usamos un `array` de `String` para los ítems (nombres de los IES). Para el `CheckBoxGroup` tendremos una creada una lista de objetos de tipo nivel que tiene dos propiedades (código y descripción). Generaremos las leyendas de las opciones con la propiedad descripción.

```
private static String [] schools = {"IES Güímar", "IES Punta Larga", "IES Santa Ana" };

private static List<Level> levelsAvailable = new ArrayList<>();
static {
    levelsAvailable.add(new Level(1, "1º ESO"));
    levelsAvailable.add(new Level(2, "2º ESO"));
```

Img 47: Items para los componentes de selección del formulario Register.

```
42          school = new ComboBox<>("IES");
43          school.setItems(schools);
44          levels = new CheckBoxGroup<>("Cursos", levelsAvailable);
45          levels.setItemCaptionGenerator(Level::getDescription);
```

Img 48: Componentes de selección del formulario Register.

4.8 . - Eventos y oyentes.

Vaadin ofrece un modelo de programación orientada a eventos para gestionar la interacción del usuario. Cuando un usuario hace algo en la interfaz, como cliquear un botón o seleccionar un ítem, la aplicación necesita saberlo. Vaadin usa el patrón evento – oyente (también conocido como observador) para comunicar la entrada del usuario a la lógica de la aplicación.
El diseño de este patrón observador se fundamenta en dos tipos de elementos:

- Eventos.
- Oyentes.

Los eventos (de distinto tipo) los emiten los componentes según la interacción que hagan los usuarios con la interfaz.

Los oyentes son interfaces que se asocian a tipos de eventos que deben implementar la lógica de la aplicación. Estos oyentes deben registrarse en el componente que genera el evento mediante el método `add*Listener` (donde `*` es el tipo de evento).

Tenemos varias formas para implementar el oyente y registrarlo en el componente:

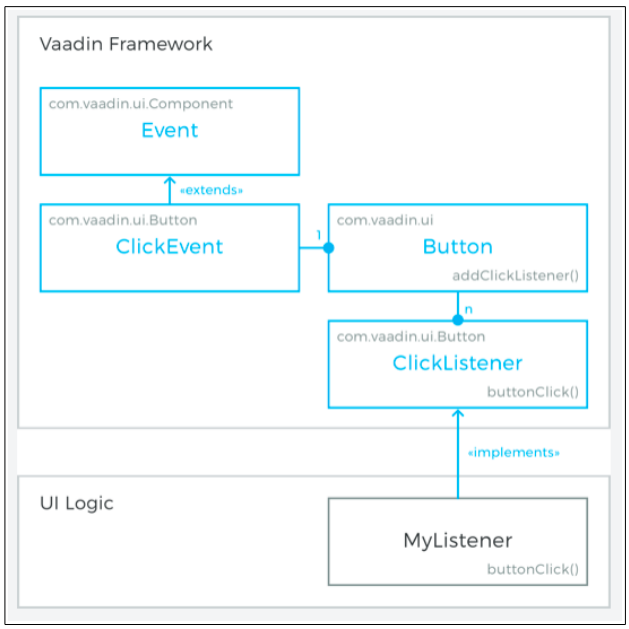
- Crear una clase que implemente la interfaz y registrala en el componente mediante el método `add*Listener`.
- Crear una clase anónima que pasamos al método `add*Listener` mediante parámetro.
- Crear una clase anónima que pasamos a través del constructor del componente.

4.8.1 . - Componente Botón.

El componente `Button` normalmente se utiliza para generar alguna acción, como la finalización de entrada de un formulario.
Cuando un botón se pulsa se dispara el evento `Button.ClickEvent` que es gestionado por el oyente `Button.ClickListener` que debe implementar el método `buttonClick`. Normalmente el oyente será una clase anónima que incluso podemos poner como parámetro en el constructor del componente o bien añadiéndola con el método `addClicListener`.

En la siguiente ilustración vemos la relación de clases e interfaces para:

- El componente Botón.
- El evento `ClickEvent` que puede generar.
- El oyente `ClickListener` que debe implementar la lógica del evento clic.



Img 49: Diagrama de clase del oyente click de un botón.

Como vimos en la sección anterior podemos registrar un oyente de click en el componente botón de las siguientes maneras:

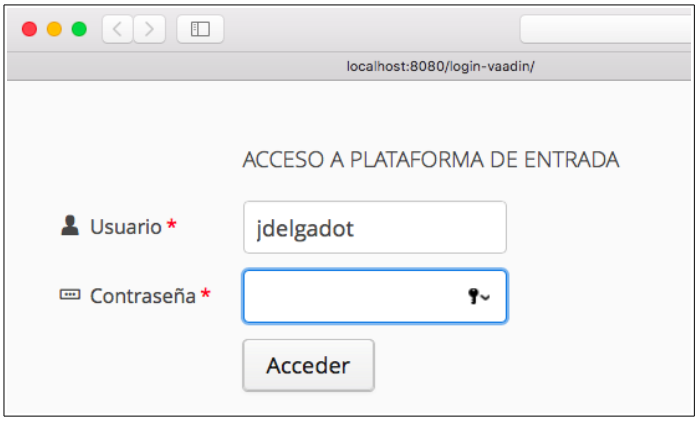
a) Clase anónima de oyente pasada como parámetro del constructor.

```
Button button = new Button("Click Me",
     new Button.ClickListener() {

                public void buttonClick(ClickEvent event) {
                Notification.show("Button Pressed");
           }

     });
```

b) Mediante el método `addClickListener` y usando expresiones lambda de java 8.

```
     Button button = new Button("Click Me");
     button.addClickListener( e -> {
          Notification.show("Button pressed");
               });
```

Añadimos los botones a los formularios Login y Register.



Img 50: Interfaz del formulario Login.

4.8.2 . - Componente Compuesto.

La facilidad para crear componentes de interfaz de usuario es una de las principales características de Vaadin. Basta con combinar componentes ya existentes en el framework para obtener un componente compuesto. Para crear un componente compuesto basta con:

- Extender un componente `layout`.
- Usar un componente compuesto (clases `Composite` ó `CustomComponent`).

La segunda opción envuelve un componente de tipo layout, con lo que conseguimos principalmente encapsulación, escondiendo los detalles de la implementación de la composición.

Para crear un componente compuesto basta con:

- Extender la clase `Composite` o `CustomComponent`.
- Establecer el componente raíz en el constructor con el método `setCompositionRoot`.

La diferencia entre las dos clases de composición es que `CustomComponent` introduce una capa más en el DOM en el navegador, que puede ser utilizada para dar estilo y personalizar el tamaño. mientras Composite es una versión más ligera que no tiene representación visual en el navegador y es reemplazada exactamente por su contenido.

En nuestro ejemplo los formulario Login y Register son ya componentes compuestos ya que extienden la clase FormLayout.

Si quisiéramos esconder este layout podríamos encapsularlo con la clase `CustomComponent`. En ese caso crearíamos una instancia FormLayout para que sea la raíz de nuestro componente.

```
17 public class Login extends CustomComponent {
18
19     private FormLayout form;
```

Img 51: Creación del componente Login.

Establecemos el componente raíz y damos tamaño al componente.

```
49         form.addComponents(label, user, pass, loginButton);
50         setCompositionRoot(form);
51         setSizeUndefined();
```

Img 52: Establecimiento de la raíz de un componente compuesto.

4.9 . - Vinculación de componentes.

La mayoría de las aplicaciones web permiten a los usuarios rellenar un formulario de entrada de datos. Estos se representan en una instancia de un objeto de negocio (bean).
Vaadin proporciona la clase `Binder` para vincular los valores del objeto de negocio con los campos que se muestran en la interfaz de usuario. La clase `Binder` gestiona la lectura de las propiedades del objeto y la conversión de tipos de datos entre el objeto y el componente.

Por ejemplo si quisiéramos registrar los datos d euna persona (modelizada con la clase `Person`) y quisiésemos vincular el campo de entrada del formulario del nombre (`namField`) con la propiedad name de la clase `Person procederemos` de la siguiente forma:

Instanciar la clase `Binder` parametrizada con el tipo de bean a vincular.

```
Binder<Person> binder = new Binder<>;
```

Con el método `forField` indicamos la instancia del componente que queremos vincular. Añadimos con el método bind la propiedad de la instancia del bean los `callbacks` de `set` y `get`.

```
TextField nameField = new TextField("Nombre");
binder.ForField(nameField)
         .bind(Person::getName, Person::setName);
```

Una vez que tengamos vinculado los componentes la clase `binder` puede cargar los valores desde un instancia del objeto a los componentes (con el método `readBean`), dejar que el usuario los edite y salvarlo (`writeBean`) a la instancia del objeto vinculado.

En nuestro ejemplo vamos a vincular los formulario `Login` y `Register` con los bean `Credential` y `Register`.

4.10 . - Validación.

La misma clase `Binder` nos permitirá añadir restricciones a cada campo vinculado. El validador se ejecutará cada vez que el usuario realice cambios en el valor del campo y cuando se salve a la instancia del objeto.

Los validadores se definen entre los métodos `forField` y `bind` con el método `withValidator` cuyo argumento puede ser:

- Una expresión lambda para la restricción y un mensaje de error.
- Una instancia de Validador (implementando la interfaz `validator` o usando algunos de los validadores que incorpora Vaadin).

El estado de la validación se muestra por defecto gracias al método `setComponentError`. En caso de error el componente tendrá un borde rojo, con el indicador de error (signo de exclamación) y el mensaje aparece como ventana emergente. Este comportamiento puede personalizarse de varias formas:

- Con el método `withStatusValue` → para que el mensaje de error aparezca como etiqueta del componente.
- Con el método `withValidationStatusHandler` → gestión más flexible y dinámica.

En nuestro ejemplo implementaremos las restricciones de los campos del formulario Login de la siguiente manera:

- Tamaño mínimo de 6 caracteres, mediante expresión lambda.
- Patrón de contraseña con una instancia del validador `PatternValidator`.

```
40        binder.forField(pass)
41            .withValidator(user -> user.length()>=6, "La contraseña debe tener al menos 6 caracteres")
42            .withValidator(new RegexpValidator("La contraseña debe tener un caracter no alfanumérico",
43                    ".*\\W.*"))
44            .bind(Credential::getPass, Credential::setPass);
```

Img 53: Validadores registrados en el campo contraseña con Vaadin.

Img 54: Muestra de error de validación con ventana emergente.

4.11 . - Navegación y vistas.

Normalmente las aplicaciones Vaadin se ejecutan en una única página web aprovechando que usa Ajax para modificar solo una parte de la interfaz. Aún así, puede ser necesario disponer de distintas vistas y navegar entre ellas. Para eso Vaadin dispone de la clase `Navigator` que es la encargada de gestionar las distintas vistas de una aplicación.

4.11.1 . - Navegador.

La clase `Navigator` es la que gestiona en Vaadin una colección de vistas que implementan la interfaz View. Estas vistas pueden ser registradas de antemano o adquiridas por un proveedor. Al registrase, las vistas deben tener un nombre identificador y ser añadidas al navegador con el método `addView()`. Podemos registrarlas en cualquier momento y una vez hecho podemos navegar hacia ellas con el método `navigateTo()`.

El navegador gestiona la navegación dentro de un componente contenedor que puede ser:

- `ComponenteContainer` (como la mayoría de `layouts`)
- `SingleComponentContainer` (`UI`, `Panel` ó `Window`)

A su vez el componente contenedor es gestionado por un monitor (`ViewDisplay`). Al instanciar el navegador debemos indicar la UI al que vinculamos el navegador y su componente contenedor. Normalmente dejaremos que el propio navegador cree de firma interna su monitor.

4.11.2 . - Proveedores de vistas.

Podemos crear vistas de forma dinámica usando un proveedor de vista que implementa la interfaz `ViewProvider`. Este proveedor se registra en el navegador con el método `addProvider()`.
Vaadin proporciona dos tipos de proveedores:

- `ClassBasedViewProvider` → crea nuevas instancias de vistas vinculando el nombre de una vista con una clase que implemente View.
- `StaticViewProvider` → devuelve una instancia de vista basada en su nombre. El método `addView()` del navegador es una forma sencilla de crear un proveedor de vista estático para cada vistas registrada.

4.11.3 . - Implementando una vista.

Las vistas pueden ser cualquier objeto que implemente la interfaz View. Cuando se llame al método `navigateTo` del navegador o la aplicación se abra con la URI asociada al avista, el navegador cambiará a la vista y se ejecutará el método `enter`.

4.11.4 . - Gestionando los parámetros.

Por defecto las URLs gestionadas a través del navegador tienen un fragmento URI que contiene el identificador de la vista. Este fragmento URI se separa del resto de la URL mediante el carácter #.
La URI puede contener parámetros adicionales para pasar a las vistas. Estos parámetros son parte de la URI y están después el identificador de la vista separados por el carácter /. Estos parámetros junto con el identificador forman el estado de navegación.

En nuestro ejemplo convertiremos nuestros componentes en vistas. Tan solo debemos implementar para ambos la interfaz View.
En nuestra UI crearemos un navegador donde registramos las vistas.

```
navigator =new Navigator(this, this);
navigator.addView("login", login);
navigator.addView("register", register);
navigator.navigateTo("login");
```

Img 55: Registro de las vistas en el navegador.

Añadimos un botón en la vista Login para registrase. Añadimos un clickListener que navega a la vista Register.

```
43          registerButton = new Button("Crear Cuenta", e ->{
44              UI.getCurrent().getNavigator().navigateTo("register");
45          }) ;
```

Img 56: Botón registrar que navega a la vista Register.

5 . - CDI Weld.

CDI (Inyección de Contexto y Dependencia) nos permite que nuestro objetos tengan sus dependencias proporcionadas de forma automática en lugar de crearlas y recibirlas como parámetros, CDI manejará el ciclo de vida de estas dependencias.

Los principales servicios que proporciona CDI son los siguientes:

- Contexto: Este servicio permite vincular el ciclo de vida de componentes con estado a ciclos de vida de contexto extensibles y bien definidos.
- Inyección de dependencia: Este servicio permite inyectar componentes en una aplicación en forma de tipo seguro y elegir en el momento del despliegue que implementación particular de la interface inyectar.

5.1 . - Beans.

Casi cualquier clase Java que tienen un constructor sin parámetros o (constructor con la anotación @Inject) es considerado por CDI como bean. Gracias a CDI, el contenedor creará y destruirá las instancias de nuestros beans asociándolos a un contesto determinado, inyectándolos en otros beans, usándolos en expresiones EL, especializándolos con anotaciones de calificadores incluso añadiendo interceptores y decoradores sin modificar el código existente. Ta sólo necesitaremos añadir algunas anotaciones.

Un bean debe tener los siguientes atributos:

- Un conjunto no vacío de tipos.
- Un conjunto no vacíos de calificadores.
- Un scope.
- Un nombre EL (opcional)
- Un conjunto de interceptores vinculados.
- Una implementación del bean.

Veamos con un poco más de detalle alguno de estos atributos:

5.1.1 . - Tipo.

Es la clase o interfaz que define el bean. En caso de EJB de sesion incluirían las interfaces y clases locales.

5.1.2 . - Calificadores.

Un calificador es una anotación @Qualifier que nos permite evitar la ambigüedad en los puntos de inyección. Por ejemplo, si tenemos varias implementaciones de una interfaz para decidir cual inyectar usaremos el calificador inyectando la interfaz. Si un bean no tiene calificador automáticamente adquiere el calificador @Default.

5.1.3 . - Scope.

Define el ciclo de vida y la visibilidad de las instancias. Los principales scope que proporciona CDI están representados por las siguientes anotaciones:

- `@RequestScoped` – vinculado a una única petición.
- `@SessionScoped` – vinculado a una sesión de usuario y compartido por todas las peticiones que se ejecutan en el contexto de esta sesión.
- `@ApplicationScoped` – vinculado al ciclo de vida de la petición.
- `@ConversationScoped`
- `@Dependent` – vinculado al objeto en el que se inyecta.

5.1.4 . - Nombre EL

Para referenciar el bean mediante expresiones EL usaremos la anotación `@Named` y entre paréntesis el nombre del bean.

5.2 . - Métodos productores.

- No siempre podremos inyectar beans con la anotación `@Inject`. Podría ocurrir: El tipo de instancia a inyectar solo lo conocemos en el momento de ejecución.
- El bean se obtiene a partir de un servicio.

Un método productor es un método que actuá como fuente de instancias de beans. La declaración del método describe el bean y el contenedor lo invocará para tener una instancia del bean en caso de que no exista ninguna en el contexto especificado.

Un método productor se declara mediante la anotación `@Produces`. Si el método productor tiene parámetros, el contenedor buscará los beans que satisfacen el tipo y calificador de cada parámetro y los pasará de form automática (otra forma de inyección de dependencia).

5.3 . - Campos productores.

Es un alternativa al método productor. En esta ocasión la anotación `@Produces` irá en la definición del campo de la clase bean.

5.4 . - Puntos de inyección

La anotación `@Inject` define el punto en el que se inyecta una instancia de bean Puede ocurrir por tres mecanismos diferentes:

- En un constructor con parámetros de un bean.
- En un método set inicializador de un campo de una clase.
- En la definición de un campo de una clase.

La inyección de dependencia siempre ocurre cuando el contenedor instancia por primera vez el bean. Simplificando un poco, las cosas ocurren de la siguiente manera:

- Primero, el contenedor llama al constructor de bean (el constructor por defecto o el anotado con `@Inject`) para obtener una instancia del bean.
- Luego, el contenedor inicializa todos los valores de los campos inyectados en el bean.
- Luego, el contender llama a todos los métodos inicializadores del bean.
- Finalmente, ejecuta los métodos con anotación `@PostConstruct`.

6 . - Vaadin CDI

Vaadin-CDI es la forma más sencilla de utilizar la inyección de contento y dependencia (CDI) en aplicaciones Vaadin. Proporciona dos tipos adicionales de scopes para UIs y vistas.
Para preparar la aplicación vaadin para Cdi daremos los siguientes pasos:

- Eliminamos la librería servet-api.
- Añadimos la librería vaadin-cdi como dependencia en el POM.
- Crearemos un archivo beans.xml dentro del directorio WEB-INF
- Eliminamos la clase que extiende `VaadinServlet` ya que disponemos de `VaadinCDIServlet` que se desplegará automáticamente.

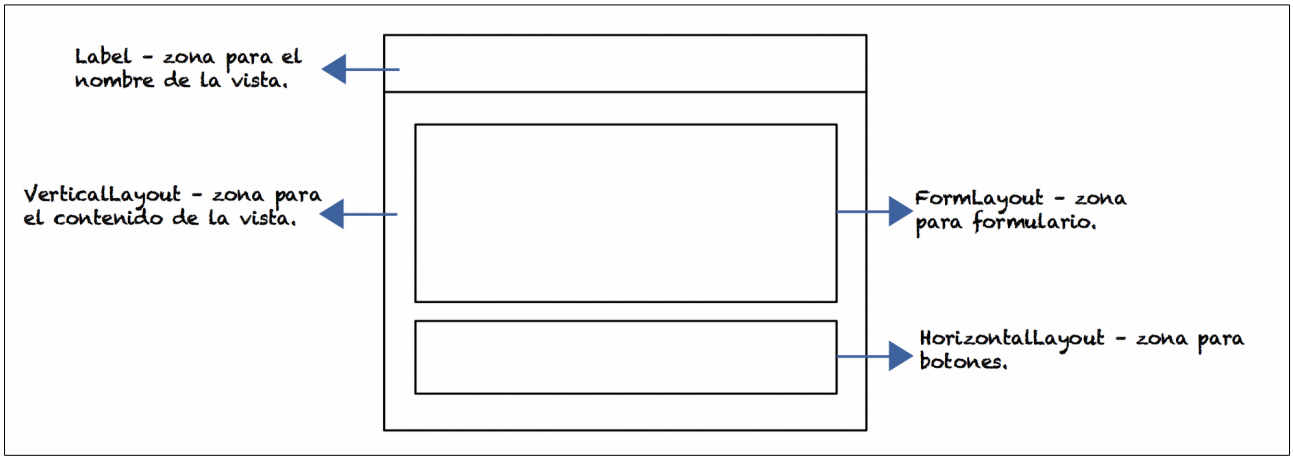
- Anotaremos con @CDIUI la clase UI que será instanciada al dar la URL. Podemos indicarle como parámetro parte del camino de la URL o dejarla vacío para indicar la raíz de la aplicación.
- Por último anotaremos las clases vistas con @CDIView indicando como parámetro el nombre de la vista.
- Inyectamos las vistas en la clase UI en lugar de instanciarlas.

7 . - Diseño e implementación Login y Register.

Los componentes Login y Register serán componentes compuestos (`CustomComponents`) a partir de una plantilla y encapsulados en componentes Vistas (`Views`).
Por tanto para cada uno de ellos podemos distinguir tres niveles.

7.1 . - Nivel layout.

Diseñamos una única plantilla para ambos componentes con estructura de panel (Panel). En la etiqueta del Panel indicaremos el nombre y en el contenido incluiremos dos zonas una con disposición específica para formularios y otra con disposición horizontal para ls botones que necesite cada componente.

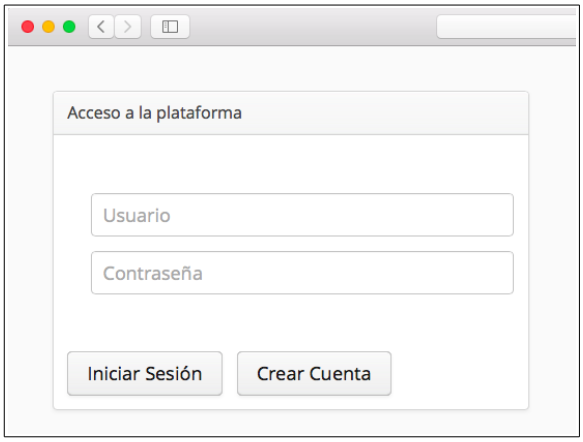


Img 57: Plantilla que usarán los componentes Login y Register.

7.2 . - Nivel componente.

En este nivel insertaremos los componentes UI en cada una de las zonas. Creamos clases que extiende `CustomComponent` con un campo `layout` que será la raíz del componente.
Para el componente Login:

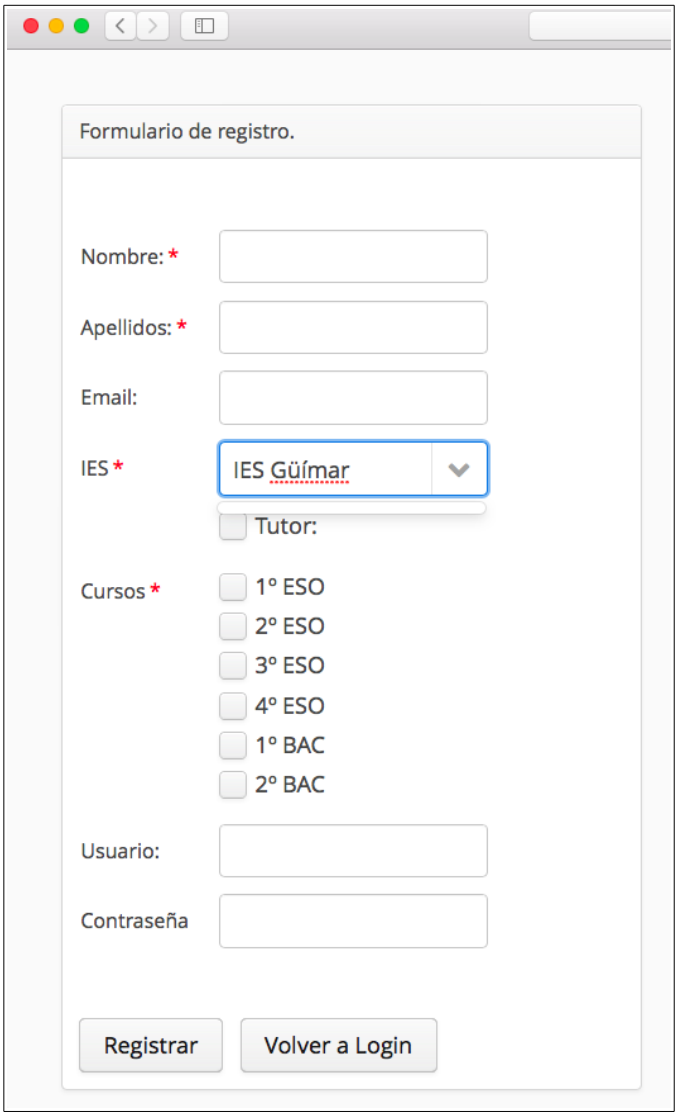
- Dos campos de entrada: uno de texto paras usuario y otro de tipo contraseña.
- Dos botones: uno para validarse y otro para ir al formulario registro.



Img 58: Componente Login.

Para el componente Register:

- Campos de entrada para nombre, apellidos, email, usuario y contraseña.
- Casilla de verificación para indicar si el profesor es tutor.
- Lista desplegable para seleccionar el centro de trabajo.
- Grupo de casilla de opciones para indicar los cursos que imparte el profesor.
- Dos botones: uno para registrarse y otra para volver a la vista Login.



Img 59: Componente Registro.

Tanto la lista desplegable como el grupo de casilla de verificación tienen ya asociados sus ítems. Para el caso del IES se trata de un array de `String`, mientras que para los cursos es una `List` de objetos `Level` (encapsulan el código y la descripción de los cursos). Estos datos han sido incluidos de forma estática.

```
// Datos con todos los posibles centros de trabajo.
private static String [] schools = {"IES Güímar", "IES Punta Larga", "IES Santa Ana" };

// Datos para indicar los cursos que puede impartir un profesor
private static List<Level> levelsAvailable = new ArrayList<>();
static {
    levelsAvailable.add(new Level(1, "1º ESO"));
    levelsAvailable.add(new Level(2, "2º ESO"));
```

Img 60: Items para los campos de selección.

7.3 . - Nivel Vista.

En este nivel para cada componente:

- Se vincula los campos con la instancia Person.
- Se registran los validadores en los campos.
- Se registran e implementan los oyentes de los botones.

7.4 . - Vinculación campos con el modelo y registro de validadores.

Para vincular los campos de la interfaz con el modelo de datos que en nuestro caso lo representa la entidad `Person`, usamos la clase `binder`.
Vamos a vincular de dos formas:

- Por campo con el método `forField` para registrar validadores personalizados.
- Por formulario con el método `bindInstancesFields` para que queden registrados las validaciones bean.

```
95      private void vinculaPass() {
96          // Vinculación del campo de contraseña pass con la propiedad pass del modelo de datos
97          binder.forField(pass)
98              // Registramos el validador
99              .withValidator(new PassValidator(capital, digit, alfa, size))
100             // Personalizamos la salida de errores por defecto
101             .withValidationStatusHandler(status -> {
102                 setStatus(status, errorPass, pass);
103             })
104             .bind("pass");
105     }
```

Img 61: Vinculación del campo contraseña y registro de su validador.

Además hemos personalizado el comportamiento por defecto de los validadores usuario y contraseña. Cada vez que falle, el campo se pondrá en borde rojo con el indicador ! y aparecerá una etiqueta en la parte superior del componente con el mensaje de error.

7.5 . - Implementación de validadores personalizados.

Hemos implementado dos validadores personalizados implementando la interfaz `Validator<T>`. Para ello debemos el método `ValidationResult apply(Sytring value, ValueContext contex)` que contendrá la lógica de la validación y devuelve el resultado encapsulado en un objeto de tipo `ValidationResult`.

```
41      @Override
42      public ValidationResult apply(String value, ValueContext context) {
43          if(value.length() < size) {
44              return ValidationResult.error("al menos " + size + " caracteres");
45          }
46          if(beta) {
47              if(! Pattern.compile("\\w*[.@-_]\\w*").matcher(value).matches()) {
48                  return ValidationResult.error("al menos . @ - _");
```

Img 62: Implementación validador de usuario.

Se han utilizado los siguientes parámetros para configurar de forma flexible los validadores contraseña y usuario:

- `capital` – true si exigimos al menos una mayúscula en la contraseña.
- `digit` – true si exigimos al menos un dígito en la contraseña.
- `alfa` – false si exigimos al menos un carácter no alfanumérico en la ocntraseña.
- `beta` – true si exigimos al menos un carácter [-_@.] en el usuario.
- `size` – indica le tamaño mínimo para usuario y contraseña.

7.6 . - Implementación y registro de oyentes en botones.

Realizamos el registro con le método `addClickListener` con expresiones lambda en el argumento para la implementación.
Los botones `registerButton` de Login y `backButton` de `Register` navegan entre las dos vistas. Luego la implementación del oyente tan sólo es obtener la instancia del navegador y usar el método `navigateTo` para ir a la vista correspondiente.

```
84    private void oyenteRegisterButton() {
85        this.registerButton.addClickListener(e -> {
86            UI.getCurrent().getNavigator().navigateTo("register");
87        });
88    }
```

Img 63: Implementación del oyente click del boton registerButton en la vista Login.

El botón `loginButton` de la vista Login debe comprobar que el usuario y contraseña son válidos. Para usamos una clase de service del paquete backend que dispone de los métodos necesarios para realizar consultas a la bases de datos `persondb` de todas las personas registradas.

```
63  private void oyenteLoginButton() {
64      this.loginButton.addClickListener(e -> {
65          if(binder.validate().isOk()) {
66              boolean valido = service.isValid(user.getValue(), pass.getValue());
67              if(!valido) {
68                  errorUser.setVisible(true);
69                  errorUser.setValue("Usuario o contraseña no válida");
70                  loginButton.setEnabled(false);
71              } else {
72                  errorUser.setVisible(false);
73                  errorUser.setValue("");
74                  Notification.show("Usuario validado");
75              }
76          }
77      });
```

Img 64: Implementación del oyente click del botón loginButton en la vista Login.

Por último el botón `registerButton` de la vista Register debe comprobar que el usuario introducido no está registrado y almacenar en la base de datos una instancia de persona con los datos del formulario.

7.7 . - Diseño del backend.

En nuestra aplicación implementaremos un backend con unos pocos personas dadas ya de alta para testar la aplicación.

7.7.1 . - Entidad.

Nuestra entidad `Person` extiende la clase abstracta `AbstractEntity` que contiene:

- Un campo con clave primaria generada automáticamente.
- Métodos sobrescritos `equals` y `hashCode`;

En esta entidad tenemos todos los campos necesarios para registrar a las personas en la plataforma.

7.7.2 . - Unidad de persistencia.

Configuramos la unidad de persistencia en el archivo `persintence.xml` dentro de la carpeta `src/main/resources/META-INF`. En el indicamos:

- El nombre de la unidad, en nuestro caso `personbd`.

- El tipo de transacción, en nuestro caso JTA (Java Transaction API) dejando que el servidor las gestione.
- Propiedades por defecto del comportamiento JPA (nivel de logging, generación de la base de datos, …).

No necesitamos indicar la fuente de base de datos ya que el servidor nos generará una para testear la aplicación.

7.7.3 . - EntityManager.

Produciremos un gestor de Entidades usando la anotación `@PersistenceContext` e indicando como parámetro la unidad de persistencia.

```
16 public class EntityManagerProducer {
17
18     @Produces
19     @Dependent
20     @PersistenceContext(unitName="personsdb")
21     public EntityManager entityManager;
22
23 }
```

Img 65: Productor de Entity Manager

7.7.4 . - Deltaspike.

Apache deltaspike es una colección de extensiones CDI portables. Consta de un módulo principal y de una serie de módulos opcionales que proporcionan distintas funcionalidades a nuestras aplicaciones. Nosotros vamos a emplear el módulo data.

El módulo data nos permite implementar patrones de repositorios. Estos patrones simplifican las consultas a la base de datos y nos permiten centranos en la lógica de la consulta reduciendo la duplicación de código y mejorando las testabilidad.

Disponemos de tres posibilidades para generar nuestras consultas:

- Declarar un método que ejecute una consulta simplemente trasladando el nombre y los parámetros.
- Declarar un ,método que automáticamente ejecute una consulta dándola en JPQL.
- Declarar un método que automáticamente ejecute una consulta dada por su nombre.

```
    List<Person> findByAgeBetweenAndGender(int minAge, int
maxAge, Gender gender);

    @Query("select p from Person p where p.ssn = ?1")
    Person findBySSN(String ssn);

    @Query(named=Person.BY_FULL_NAME)
    Person findByFullName(String firstName, String lastName);
```

Img 66: Ejemplos de patrones de consulta.

Si nos fijamos en el repositorio de nuestra aplicación vemos:

- La anotación `@Repository`
- Un método que usa la primera de las opciones vistas anteriormente. El nombre del método permite traducir a la siguiente consulta:

```
select p from Person where p.name = filter and p.pass = pass.
```

La implementación de los métodos se hace automáticamente por esta extensión del Cdi. Sólo debemos declarar la dependencia en la interfaz.

```
15 @Repository
16 public interface PersonRepository extends EntityRepository<Person, Long> {
17
18     public List<Person> findByUserAndPass(String filter, String pass);
19 }
```

Img 67: Repositorio para las entidades Person.

7.7.5 . - PersonService.

En esta clase definimos los métodos que usarán los oyentes de los botones registrar y acceder ocultando de esta manera la capa de JPA.
Además añadimos a modo de prueba unos datos de personas.

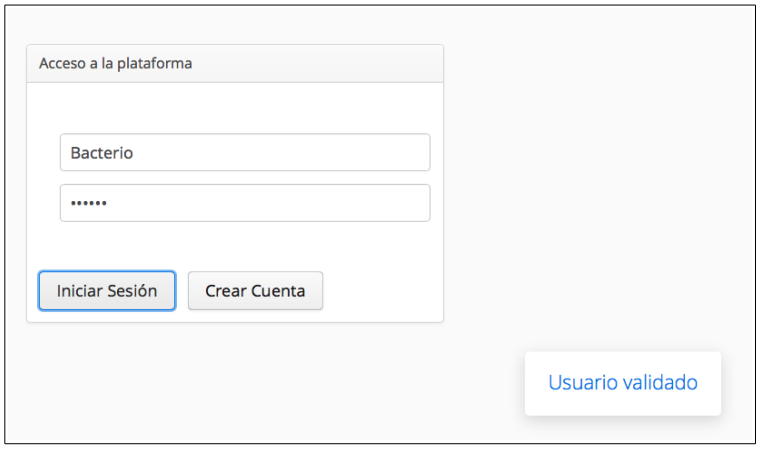
8 . - Juego de pruebas.

8.1 . - Caso 1: Validarse usuario registrado.

En PersonService tenemos tres personas ya registradas cuyos usuarios y contraseñas son:

Usuario	**Contraseña**
Mortadelo	Pass$0
Filemón	Pass$1
Bacterio	Pass$2

Probaremos con el tercer usuario.



Img 68: Usuario validado.

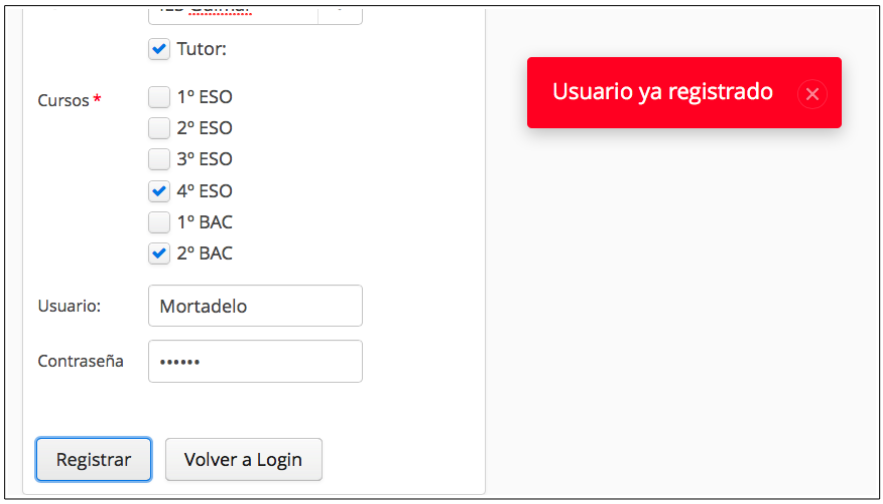
8.2 . - Caso 2: Validarse usuario no registrado .

Probamos con jdelgadot como usuario y Pass$3 como contraseña

Img 69: Usuario no validado.

8.3 . - Caso 3: Crear cuenta usuario existentes.

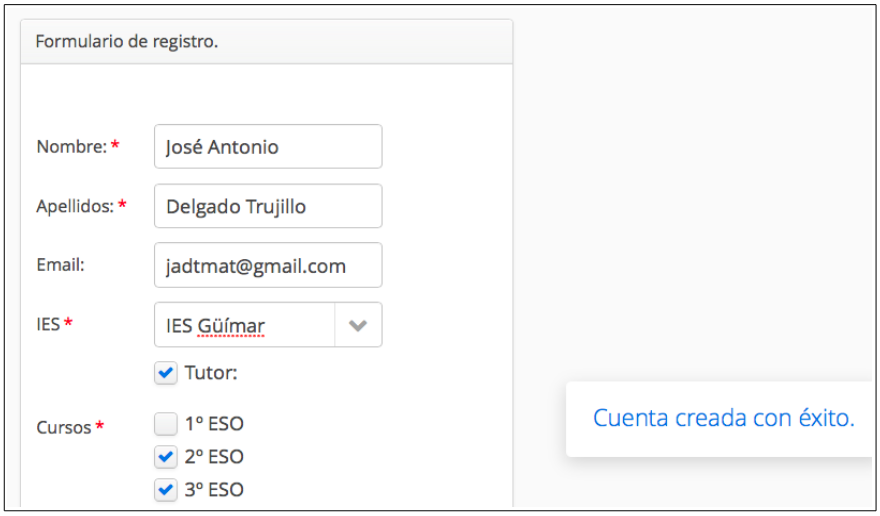
Crearemos una cuenta usando el usuario Mortadelo.



Img 70: Registro de usuario ya registrado.

8.4 . - Caso 4: Crear cuenta usuario inexistente.

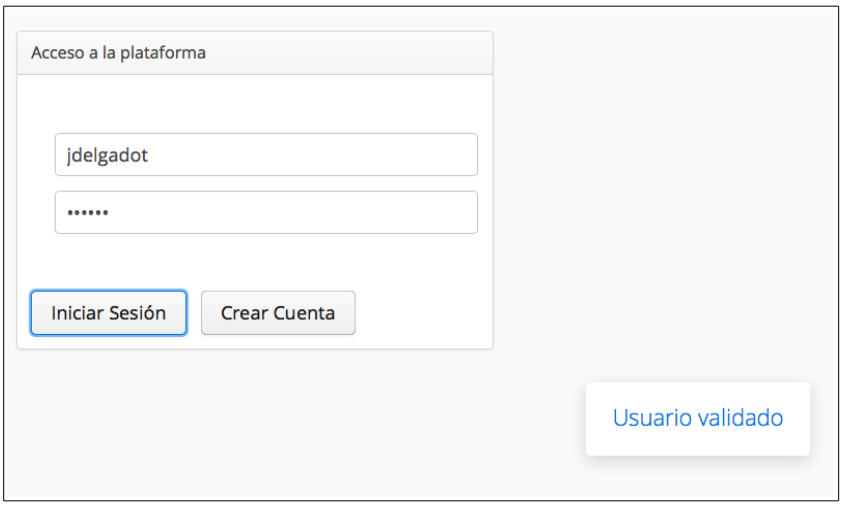
Crearemos una cuenta con usuario jdelgadot y Pass$3 como contraseña.



Img 71: Registro de nuevo usuario.

8.5 . - Caso 5: Validarse con la cuenta creada.

Comprobamos que el anterior que en el caso2 no podía validarse ahora si puede.



Img 72: Validación del nuevo usuario.

9 . - Bibliografía

- **Deepak Alur; Dan Malks; John Crupi** (2003). Core J2EE Pattern:Best Practice and Design Strategies (2nd Edition)(Prentice Hall Core Series).

- **Dave Newton** (2009). Apache Struts 2 Web Application Development. Birmingham : Packt Publishing.

- **Raghuram Bharathan** (2015). Apache Maven CoockBook. Birminfham. Pack Publishing.

- **Nicolas Fränkel** (2013). Learning Vaadin 7. Birmingham: Pack Publishing.

- Core Developers Guide.
  https://struts.apache.org/core-developers/

- Struts2 core API.
  https://struts.apache.org/maven/struts2-core/apidocs/index.html

- GitHub Struts examples.
  https://github.com/apache/struts-examples

- Javaee Part III – The web Tier.
  https://javaee.github.io/tutorial/partwebtier.html#BNADP

- JSF API (Mojarra implementation).
  https://javaserverfaces.github.io/docs/2.3/index.html

- GitHub javaee 8 examples.
  https://github.com/javaee-samples/javaee8-samples

- Documentación Vaadin 8.
  https://vaadin.com/docs

- GitHub Vaadin.
  https://github.com/vaadin

- Webinar Vaadin – JPA Best Practice.
  https://www.youtube.com/watch?v=f1VD3y2PWZs&list=PLcRrh9hGNalnmA1mbDS0NBuq6N3Mnw2u1&index=16

- Javaee Part V – Context and Dependency Injection for Java EE.
  https://javaee.github.io/tutorial/partcdi.html#GJBNR

- Weld – CDI Reference Documentation.
  http://docs.jboss.org/weld/reference/latest-master/en-US/html/

- Javaee Part IV – Bean Validation.
  https://javaee.github.io/tutorial/partbeanvalidation.html#sthref1322

- Introduction to the Java Persistence API.
  https://javaee.github.io/tutorial/persistence-intro.html#BNBPZ

- DeltaSpike Documentation.
  https://deltaspike.apache.org/documentation/

- WildFly Documentation.
  https://docs.jboss.org/author/display/WFLY10/Documentation

- Maven Documentation.
  http://maven.apache.org/guides/index.html

- Maven repository search.
  https://mvnrepository.com

- The Regular Expression Tools.
  https://regex101.com/r/tUxrBG/1

- User Guide Eclipse.
  http://help.eclipse.org/oxygen/index.jsp?nav=%2F0