UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

PROYECTO FINAL DE CARRERA

"Diseño e implementación de Evaltics utilizando Google Web Toolkit"

CÓDIGO: **DISCA-127**
Titulación: **II**

AUTOR: **Jaime Serrano Cartagena**

DIRECTOR: **Dr. Lenin Guillermo Lemus Zúñiga**
Profesor Titular de Universidad
Universitat Politècnica de València

Septiembre 2012

Resumen

La presente memoria corresponde al proyecto final de carrera de la titulación de Ingeniería Informática en la Universitat Politècnica de València. El proyecto consiste en la adaptación de un software de gestión de evaluaciones a la tecnología `Google Web Toolkit`.

En las siguientes secciones se detallan la metodología utilizada, las tecnologías usadas y las conclusiones de su utilización.

La memoria está estructurada en los siguientes puntos:

- Inicialmente se establecen los objetivos y la metodología para alcanzar dichos objetivos
- Se presentan las tecnologías que se han utilizado para la realización del proyecto
- Seguidamente se muestra la configuración del entorno de desarrollo paso por paso
- Se muestra detalladamente el patrón utilizado para la realización de la parte cliente de la aplicación
- Se detalla así mismo como crear la parte servidora
- A continuación se describe la capa de persistencia y como se integra en las otras partes de la aplicación
- Una vez teniendo todas las partes de la aplicación, se muestra como se pueden validar los módulos
- Se pasa a una explicación de como se implementado la seguridad e integrado en `GWT`
- Finalmente se muestran los pasos necesarios para instalar la aplicación en un entorno de producción.
- Para terminar podemos encontrar una serie de reflexiones finales y un listado de referencias.

Tabla de contenidos

1.Introducción....5
2.Objetivo y motivaciones....5
3.Metodología a seguir....6
4.Entorno tecnológico....7
Entorno de desarrollo: Eclipse....7
GWT....7
Maven....7
Spring Framework....8
Hibernate....9
MySQL....9
JSON....10
Apache Tomcat....10
Comparación con otras tecnologías....10
5.Configuración del entorno de desarrollo....12
Instalación de Eclipse....12
Instalación de plugins GWT....12
Instalación del plugin para la integración de Maven en Eclipse....13
Creando el proyecto Evaltics....14
Probando el proyecto....19
6.Desarrollando el cliente de la aplicación....23
Modelo MVP....24
Generando la estructura de una aplicación MVP....25
Creación de actividades y vistas....27
7.Desarrollo de la parte servidora....31
Servicios RPC....34
Servicios REST....37
8.Desarrollo de la vista. Estructura de la aplicación....43
Casos de uso....44
Profesores....44
Alumnos....45
Estructura de la aplicación....45
Información Académica....46
Gestión de evaluaciones....48
Panel de alumnos....49
9.Implementando persistencia en base de datos....52
Configuración de Hibernate....55
Modificación de los DAO para el acceso a datos....59
10.Integrando la aplicación web con la capa de persistencia....62
Integración por medio de llamadas RPC....62
Integración de los servicios REST....63
11.Validando aplicaciones GWT....65
Casos de prueba con JUnit....65
12.Seguridad....70

Lanzando excepciones hacia el cliente de GWT....72
Roles en Evaltics....73
Ofuscación de la clave en base de datos....75
13.Instalación y despliegue de la aplicación....77
Instalación de Apache Tomcat....77
Instalación de MySQL....78
Creación e instalación de la aplicación....79
14.Conclusiones y mejoras....82
15.Referencias y recursos....83

1. Introducción

En el marco de la adaptación de las carreras universitarias a la nuevas regulaciones europeas que intentan homogeneizar el sistema educativo de la Unión Europea, los profesores se enfrentan al reto de mejorar la calidad de la enseñanza y desde la Escuela Técnica Superior de Ingeniería Informática se ha creado un programa de software para el seguimiento y evaluación de la acción formativa, llamado `Evaltics`.

`Evaltics` es una herramienta diseñada para mejorar la calidad de la enseñanza mediante evaluaciones periódicas vía web. La idea general es que los profesores propongan una serie de preguntas a los alumnos antes de empezar una unidad didáctica y sean capaces de evaluar el grado de aceptación del conocimiento al final de la misma mediante evaluaciones posteriores. De esta manera los profesores pueden guiarse y adaptarse para reforzar ciertos áreas donde se puedan detectar lagunas.

La herramienta se ha desarrollado inicialmente con el software `Joomla`, y este proyecto surge como una evaluación de las posibles mejoras que pueden obtenerse al utilizar una plataforma para la construcción de interfaces gráficos avanzados en web.

2. Objetivo y motivaciones

El proyecto se realiza con el objetivo de evaluar la tecnología `GWT` como uso para realizar aplicaciones ricas para la web que proporcionen un interfaz de usuario más dinámico e intuitivo para la gestión de datos, en contraposición a las aplicaciones tradicionales enfocadas a la web y basadas en un modelo de envío de formularios web.

También se ha realizado el proyecto con el objetivo de utilizar tecnologías relativamente novedosas como JSON y de como realizar la integración de distintas tecnologías disponibles dentro de un proyecto Web, desde el uso de sistemas de mapeo objeto-relacional a la integración de distintas tecnologías para la parte servidora.

3. Metodología a seguir

La metodología utilizada para la consecución del objetivo a sido:

- Obtener información sobre las herramientas necesarias para el desarrollo de la aplicación.
- Elección de las tecnologías a utilizar tanto en la parte cliente como en la parte de servidor.
- Instalación de los programas necesarios para el desarrollo.
- Creación de un entorno de desarrollo adecuado y efectivo para la creación de aplicaciones en `GWT`. Esto incluye el uso de herramientas que ayuden a proporcionar una buena gestión de un proyecto de software durante el ciclo de vida del mismo.
- Establecimiento de los requisitos funcionales de la aplicación
- Pruebas de concepto para validar que las tecnologías de cliente y servidor son compatibles
- Desarrollo de las distintas partes de la aplicación
- Establecimiento de mecanismos de seguridad a la aplicación
- Creación de validaciones para asegurar la calidad del software
- Pruebas de validación a nivel de usuario
- Despliegue de la aplicación en un entorno simulado de producción para comprobar su correcto funcionamiento antes de la distribución

4. Entorno tecnológico

A continuación se describen las tecnologías utilizadas para la realización del proyecto:

Entorno de desarrollo: Eclipse

Entorno de desarrollo integrado de código abierto multilenguaje y multiplataforma que puede ampliar sus funcionalidades a través de plugins.

Provee herramientas para la mayoría de tareas necesarias para la realización de un proyecto software: editores de código, herramientas de compilación, depuración, integración con sistemas de control de versiones, etc...

Para el proyecto se ha utilizado el IDE Eclipse, en su versión 3.7.

GWT

GWT es una tecnología para la creación de interfaces de usuario que se ejecutan en un navegador y por tanto corresponde a la parte cliente de una aplicación cliente-servidor. La parte cliente se comunica con la parte servidor por medio de llamadas a procedimientos remotos (RPC) en su versión básica, pero GWT no impone restricciones a la tecnología usada en la parte servidora.

GWT es una plataforma de código abierto desarrollado por Google y lanzado en mayo del 2006 bajo licencia Apache 2.0.

GWT permite escribir aplicaciones AJAX en el lenguaje de programación Java, que son compiladas posteriormente, traduciendo la parte del cliente a lenguaje de programación JavaScript + HTML + CSS, generando código JavaScript más eficiente que el escrito a mano.

Proporciona un conjunto de módulos como DOM, XML, JSON, I18N, RPC, Widgets, MVP, etc. amplio para el desarrollo de las aplicaciones cliente.

Las ventajas que proporciona son:

- No es necesario aprender un nuevo lenguaje para crear aplicaciones AJAX, puesto que se utiliza el ampliamente difundido lenguaje JAVA.
- Los errores de tipos propios de un lenguaje como Javascript se detectan durante la compilación
- Abstrae de incompatibilidades entre navegadores
- Se utiliza una API para el acceso a los elementos de la página web y no es necesario el conocimiento de la API DOM.
- Integración con test JUnit
- Proporciona herramientas para la internacionalización
- Se puede depurar en tiempo real.

Maven

Maven es una herramienta que se orienta a la gestión de las tareas más básicas de un proyecto de software en Java, desde la creación de la estructura inicial de un proyecto hasta la gestión de dependencias a su compilación y empaquetado o la generación de documentación.

Su utilización se basa en una configuración basada en XML (fichero `pom.xml`) para el proyecto donde se definen el nombre, las dependencias de primer nivel y otras configuraciones. `Maven` se encarga de gestionar las dependencias, descargando los paquetes necesarios para satisfacer todas las dependencias de primer y siguientes niveles desde un repositorio central donde se encuentran la mayoría de librerías de software públicos u otros que pueden ser configurados.

`Maven` está orientado a plugins, habiendo plugins para las más diversas tareas: gestión de la documentación, ejecución de tests o automatización de tareas.

`Maven` es una herramienta de línea de comandos, pero existen soluciones para la integración de la herramienta en distintos entornos de desarrollo como `Eclipse`, facilitando la utilización y haciendo la mayor parte de procesos transparente. Para el proyecto no será necesaria la utilización de la línea de comandos. En caso de ser necesaria, su instalación requiere la descarga de un paquete desde `http://maven.apache.org/` y la configuración de una variable de entorno al directorio `bin` dentro del directorio donde se haya descomprimido.

Dentro del proyecto `Evaltics`, `Maven` es utilizado para la gestión automática de dependencias y para la generación del paquete final listo para ser desplegado en el servidor de *Servlets*.

El ciclo de desarrollo de un proyecto de software con `Maven` sigue las siguientes fases:

1. Validación de las dependencias y que el estado es correcto.
2. Compilación del código del proyecto
3. Pruebas del código compilado sin que sea necesario el empaquetado o despliegue de la aplicación.
4. Empaquetado. El código se empaqueta para ser distribuido en librerías *jar* o aplicaciones web *war*
5. Tests de integración. Procesa y despliega el paquete en un entorno donde se pueden ejecutar las pruebas.
6. Verificación del nivel de calidad
7. Instalación en repositorios locales para ser utilizado como dependencias
8. Despliegue de la librería en repositorios públicos o compartidos.

Spring Framework

`Spring` es un *framework* ligero para construir aplicaciones empresariales. `Spring` puede ahorrarnos mucho trabajo ya que puede coordinar todas las partes de la aplicación. Siendo muy modularizado, nos permite usar solo las partes que necesitamos, sin tener la carga de los que no usemos.

El núcleo de `Spring` está basado en un principio o patrón de diseño llamado "*Inversión de Control*" (`IoC` por sus siglas en inglés). Las aplicaciones que usan el principio de `IoC` se basan en su configuración (que en este caso puede ser en archivos XML o con anotaciones) para describir las dependencias entre sus componentes, esto es, los otros objetos con los que interactúa. En este caso "*inversión*" significa que la aplicación no controla su estructura; permite que sea el *framework* de `IoC` (en este caso `Spring`) quien lo haga.

Algunas de las características de `Spring` son:

- Simplificación de la programación orientada a aspectos.
- Simplificación del acceso a datos.
- Simplificación e integración con `JEE`
- Soporte para planificación de trabajos.

- Interacción con lenguajes dinámicos (como BeanShell, JRuby, y Groovy).
- Soporte para acceso a componentes remotos.
- Manejo de Transacciones.
- Su propio framework MVC.
- Su propio Web Flow.
- Manejo simplificado de excepciones.
- Soporte para Servicios Web REST
- Validación de *beans*

Spring es una plataforma para estructurar una aplicación

Hibernate

Hibernate es una tecnología ORM (mapeo de objetos – relacional) para realizar el mapeo de objetos en Java con tablas de una base de datos relacional. Hibernate se encarga de transformar en consultas a la base de datos a partir de la información contenida en el código Java mediante una configuración en XML o en anotaciones dentro de las clases.

Hibernate abstrae la capa de persistencia con una *API* propia de forma que sea transparente el cambio de motor de base de datos en la aplicación.

MySQL

MySQL® es un servidor de base de datos SQL (*Structured Query Language*) muy rápido, multi-threaded, multi usuario y robusto. El servidor MySQL está diseñado para entornos de producción críticos, con alta carga de trabajo así como para integrarse en software para ser distribuido. MySQL es una marca registrada de MySQL AB.

El software MySQL tiene una doble licencia. Se puede elegir entre usar el software MySQL como un producto Open Source bajo los términos de la licencia GNU General Public License o en una versión comercial distribuido por la empresa MySQL AB.

Las principales características de este gestor de bases de datos son las siguientes:

- Escrito en C y en C++
- Aprovecha la potencia de sistemas multiprocesador, gracias a su implementación multihilo.
- Soporta gran cantidad de tipos de datos para las columnas.
- Dispone de API's en gran cantidad de lenguajes (C, C++, Eiffel, Java, Perl, PHP, Python, Ruby, y Tcl.).
- Gran portabilidad entre sistemas, puede trabajar en distintas plataformas y sistemas operativos.
- Soporta hasta 32 índices por tabla.
- Tablas hash en memoria, que son usadas como tablas temporales.
- Proporciona sistemas de almacenamiento transaccionales y no transaccionales.
- Gestión de usuarios y contraseñas, manteniendo un muy buen nivel de seguridad en los datos.
- Fácil instalación

JSON

JSON (JavaScript Object Notation) es un f**ormato para intercambiar datos sencillos**, entendiendo como sencillos el texto, los números y los valores lógicos, pudiendo estar organizados en estructuras.

JSON es un subconjunto de la notación literal de objetos de Javascript pero no requiere el uso de Javascript. Proporciona una manera sencilla de convertir matrices, objectos o variables en general desde cualquier lenguaje a Javascript mediante una representación en texto plano la variable que luego es interpretada por el motor Javascript.

A continuación se muestra un ejemplo de una cadena en formato JSON:

```
{ "id": 1 , "nombre": "Asignatura Básica", "centro" : { "id" : 2, "nombre" : "Centro Especializado" }, "creditos" : 20 }
```

Apache Tomcat

Apache Tomcat es un servidor HTTP y un contenedor de *servlets* que implementa la especificación de *servlets* y de JSP. Es software libre gestionado por la fundación Apache.

Puede funcionar como un servidor HTTP independiente o conectado a otro servidor como el servidor Apache y es multiplataforma.

En el proyecto Evaltics, Tomcat es utilizado para el despliegue e instalación en un entorno de producción aunque pueden utilizarse perfectamente otros servidores web como Jetty o servidores propietarios que implementen la especificación de *servlets*.

Se ha utilizado la versión 6.0.x, que implementa la especificación *servlet* 2.5 y *JSP* 2.1.

Los Servlets son programas escritos en Java y que dan una respuesta alternativa a la programación Web con CGI, ampliando su funcionalidad. Se ejecutan en un servidor Web dentro de un contenedor de servlets (por ejemplo Apache Tomcat) y se utilizan para construir páginas Web. Actúan como una capa intermedia entre las peticiones que se reciben de un navegador Web y el acceso a la información en el servidor, por ejemplo, a bases de datos.

Java Server Pages (JSP) es una tecnología que nos permite mezclar HTML estático con HTML generado dinámicamente.

Comparación con otras tecnologías

GWT se engloba dentro de las tecnologías para la creación de aplicaciones AJAX. Dentro de este grupo podemos encontrar otras tecnologías para el desarrollo de aplicaciones ricas:

- Flex: permite la creación de aplicaciones dinámicas que se muestran el formato Flash en el navegador. El formato Flash está perdiendo importancia dentro de la web siendo una tecnología en decadencia actualmente y no es soportado nativamente por los navegadores

- Silverlight: al igual que Flex no es una tecnología soportada nativamente por los navegadores necesitando de plugin externos que limita la compatibilidad.

- Librerías de `javascript.` Existen multitud de librerías para facilitar la creación de aplicaciones dinámicas utilizando `Javascript` como `Jquery, Prototype, YUI,` etc... estas librerías obligan a aprender un nuevo lenguaje y tienen unos entornos de desarrollo flojos.

Para la persistencia de datos se ha realizado el proyecto en una instalación tradicional en base de datos relacional. Al estar la aplicación estructurada con una clara separación de la capa de acceso a datos, sería posible utilizar otras tecnologías más innovadoras como bases de datos `NoSQL` mediante la modificación de dicha capa. Para ello podemos utilizar estándares como `JPA` que permiten el uso de librerías de integración para acceder a las más diversas fuentes de datos, tanto relacionales como de otra índole.

Para el despliegue de la aplicación se utiliza un servidor independiente. `GWT` permite la integración en `Google AppEngine` para la publicación de aplicaciones. Sin embargo esta solución presenta problemas de interoperatividad entre *frameworks*, presentando limitaciones por deberse a un entorno cerrado. Como ventaja, tenemos la rápida distribución de la aplicación en la nube.

5. Configuración del entorno de desarrollo

En esta sección se describe el proceso para poner a punto el entorno de desarrollo que servirá de marco para la realización del proyecto.

En resúmen, el proceso consta de la instalación del software de desarrollo integrado y la instalación de los plugins necesarios para la realización de un proyecto `GWT`. Una vez instalado el software necesario, crearemos el proyecto.

Instalación de `Eclipse`

Como paso previo para la instalación de Eclipse, debemos disponer de una máquina virtual de `Java` instalada. Puesto que vamos a realizar un proyecto Java, esta máquina debe ser un `JDK`. La instalación de la máquina es muy sencilla una vez descargada la última versión desde `http://www.oracle.com/technetwork/java/javase/downloads`.

Para el proyecto se ha utilizado la versión 3.7 de `Eclipse` (*Indigo*), descargable desde la web del proyecto `http://www.eclipse.org/downloads/packages/release/indigo/sr2`, en su modalidad "*Eclipse IDE for Java EE Developers*".

La instalación no requiere ningún proceso, se descomprime en un directorio adecuado y se inicia con el ejecutable `eclipse.exe`. Al iniciar nos solicita un directorio de trabajo donde se crearán los ficheros del proyecto.

Instalación de *plugins* `GWT`

Una vez arrancado `Eclipse`, seleccionaremos la opción "Install New Software..." en el menú *Help* y pondremos la dirección `http://dl.google.com/eclipse/plugin/3.7` en el campo "*Work with*:"

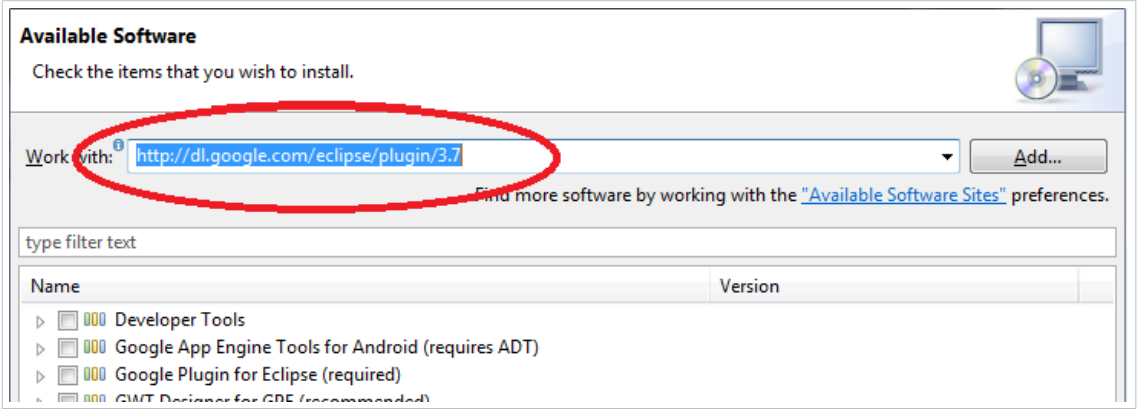


Ilustración 1: Dirección para la instalación de plugin GWT

Al aceptar la dirección nos muestra las opciones disponibles para instalar. De entre todas las opciones marcaremos las siguientes (las opciones exactas pueden cambiar según la versión del *plugin*):

- `Google Plugin for Eclipse`
- `GWT Designer for GPE`

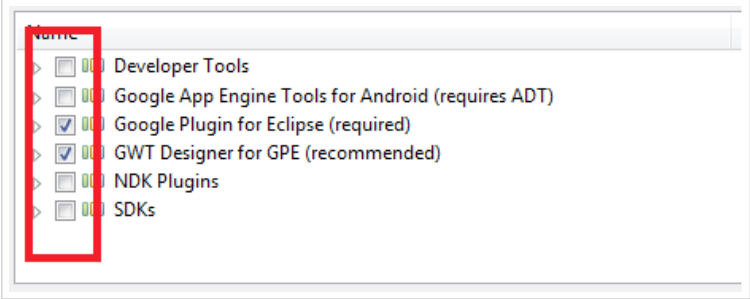


Ilustración 2: Opciones mínimas a instalar para el plugin de GWT en Eclipse

Una vez seleccionados, pulsaremos sobre el botón *"Next"* hasta aceptar los términos de licencia y proceder a la instalación cuando nos aparezca la opción *"Finish"*.

Terminada la instalación nos pedirá reiniciar el programa. Podemos hacerlo en este momento o instalar el siguiente *plugin* y reiniciar una vez instalados ambos.

Instalación del *plugin* para la integración de `Maven` en `Eclipse`

Para la instalación del *plugin* `m2e` para la integración de `Maven` con `Eclipse` procedemos de forma similar a la instalación del anterior *plugin,* utilizando como dirección de instalación `http://download.eclipse.org/technology/m2e/releases`. La opción a instalar será *"Maven Integration for Eclips*e"

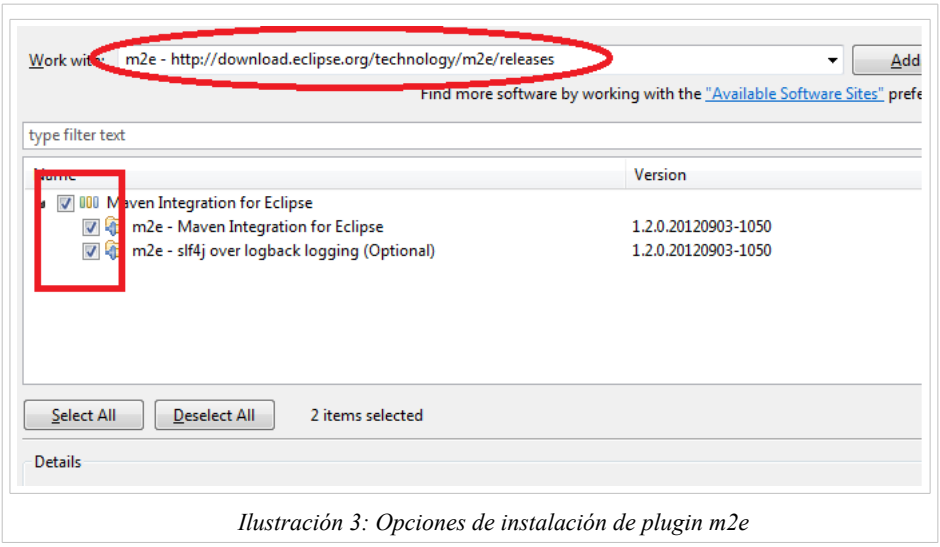


Ilustración 3: Opciones de instalación de plugin m2e

Una vez instalado el *plugin*, reiniciaremos el entorno de desarrollo cuando nos lo solicite o con la opción *"Restart"* desde el menú *"File"*.

Creando el proyecto *Evaltics*

Para la creación de la estructura inicial de la aplicación utilizaremos `Maven`, mediante el *plugin* instalado en la sección anterior. También es posible la creación de un proyecto desde las herramientas proporcionadas por el *plugin* de Google o incluso la creación de un proyecto plano al que luego se le pueden agregar las distintas opciones. En nuestro caso procedemos de la siguiente manera.

Mediante la opción *New* → *Project...* situado en el menú File, seleccionaremos crear un proyecto `Maven:`

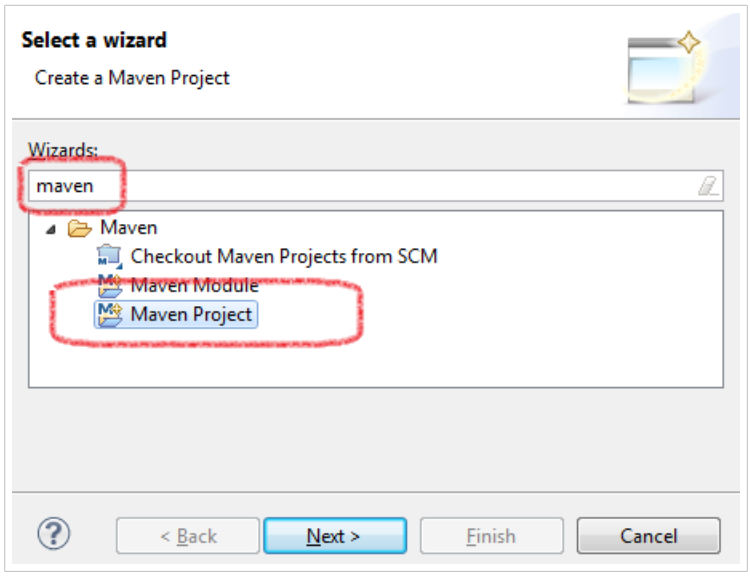


Ilustración 4: Selección de un proyecto Maven para un nuevo proyecto en Eclipse

En la siguiente pantalla aparecen opciones que podemos dejar sin modificar y la siguiente nos pide la selección de un arquetipo. Los arquetipos en `Maven` son configuraciones que definen la estructura de un proyecto. En nuestro caso seleccionamos el arquetipo `gwt-maven-plugin` en su versión 2.4.0. Para ello utilizamos el filtro que reducirá la lista de arquetipo mostrados:

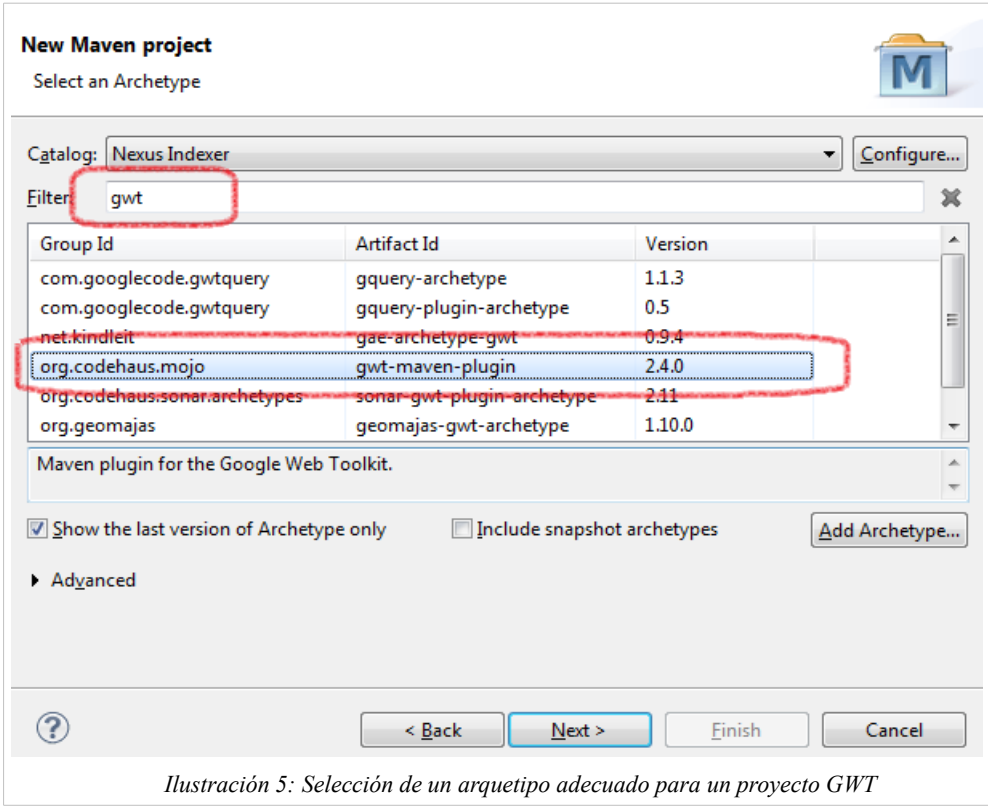


Ilustración 5: Selección de un arquetipo adecuado para un proyecto GWT

Una vez seleccionado el arquetipo y pulsando *Next,* nos muestra la pantalla donde nos pide los datos básicos del proyecto: nombre, paquete de clases `Java`, etc... Lo rellenaremos con los siguientes datos de ejemplo:

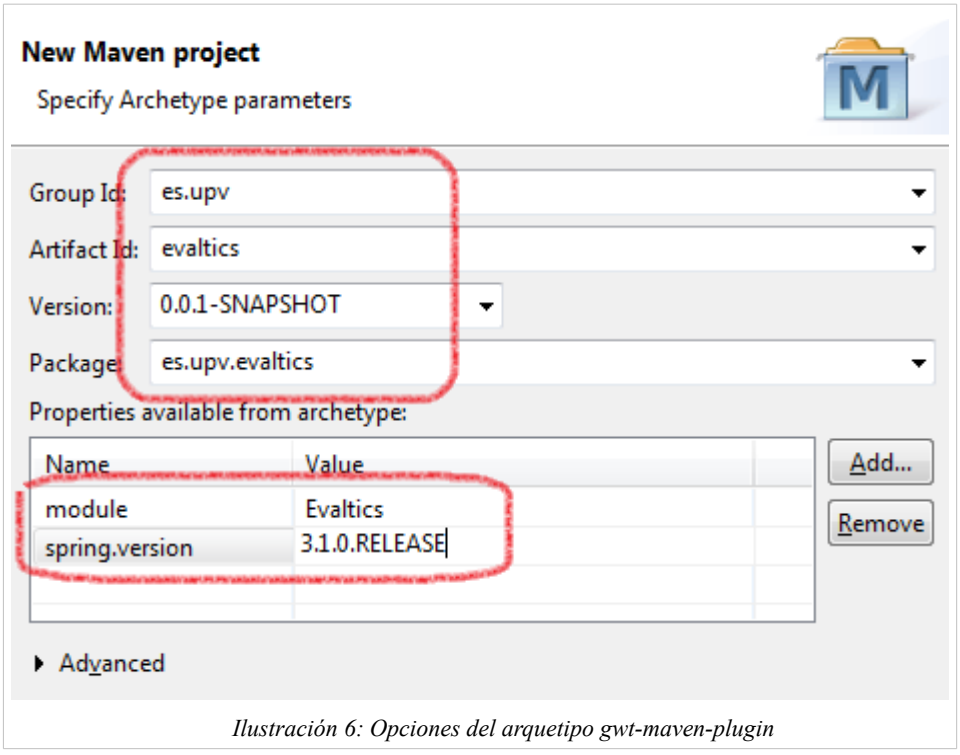


Ilustración 6: Opciones del arquetipo gwt-maven-plugin

La propiedad *spring.version* será utilizada más tarde para las dependencias de `Spring`.

Al pulsar sobre *Finish,* se crea un nuevo proyecto en el espacio de trabajo, con la siguiente estructura.

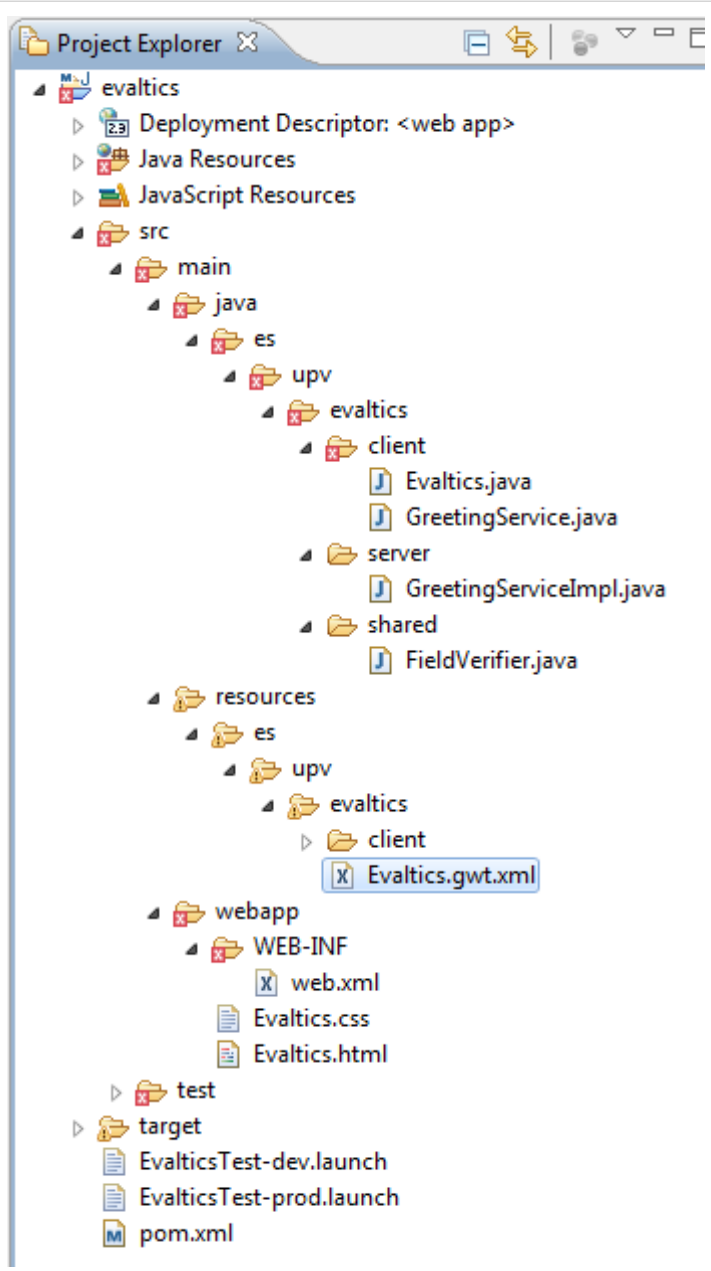


Ilustración 7: Estructura inicial del proyecto GWT Evaltics

El arquetipo contiene el esqueleto básico de la aplicación y crea un módulo *"Evaltics.gwt.xml"* junto con un código de ejemplo para poder probar la aplicación.

Es posible que aparezcan errores al inicio. Esto lo podemos ver en la pestaña *Markers* de `Eclipse:`

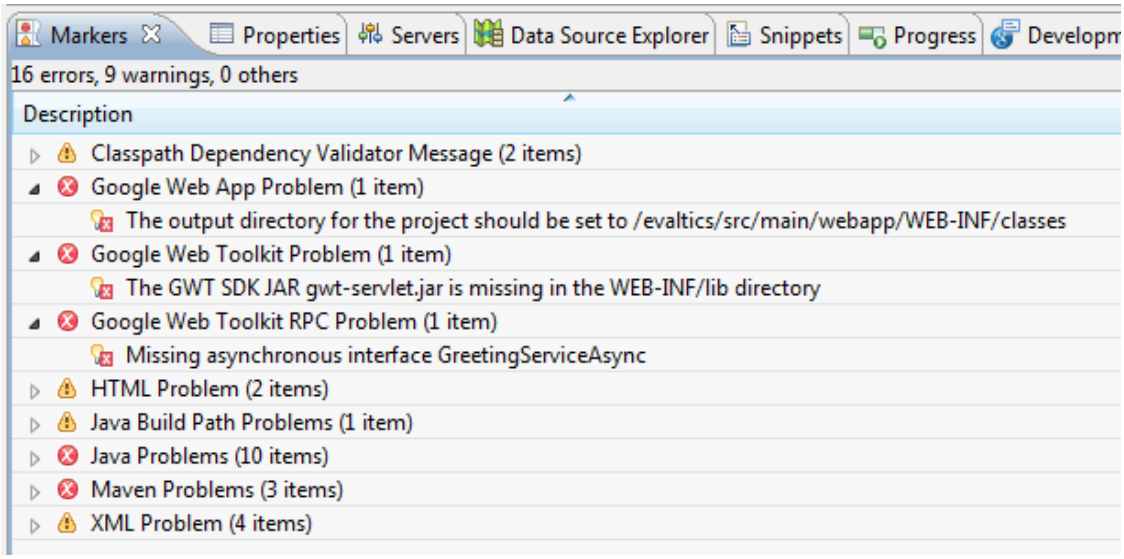


Ilustración 8: Posibles errores en la vista Markers de Eclipse al generar el proyecto inicial

Los dos primeros errores son problemas de configuración y el tercero necesita de la generación de un interfaz para el correcto funcionamiento de la aplicación.

Antes de resolver los posibles problemas nos aseguraremos que la siguiente opción de configuración está deshabilitada: en las propiedades del proyecto, accediendo mediante el menú contextual en el proyecto o en la opción *File → Properties,* seleccionamos la opción *Google → Web Application.* La opción Launch *and deploy....* debe estar desmarcada y el directorio WAR en s*rc/main/webapp* que son los directorios estándar para un proyecto `Maven`.

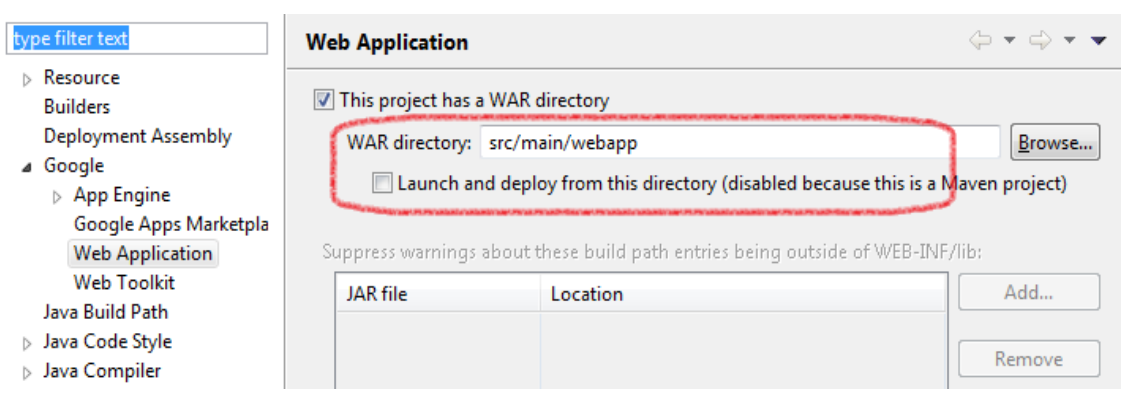


Ilustración 9: Configuración de un proyecto GWT en Eclipse

Esto soluciona los primeros errores mostrados anteriormente. Para resolver el tercer problema *(Missing asynchronous...),* pulsaremos sobre el error y en la opción *Quick Fix* (o mediante *Ctrl* + 1), nos aparece un diálogo con la opción de generar el interfaz *GreetingServiceAsync.* Pulsaremos la opción y pulsaremos *Finish* sobre este y el siguiente diálogo para mantener las opciones sugeridas.

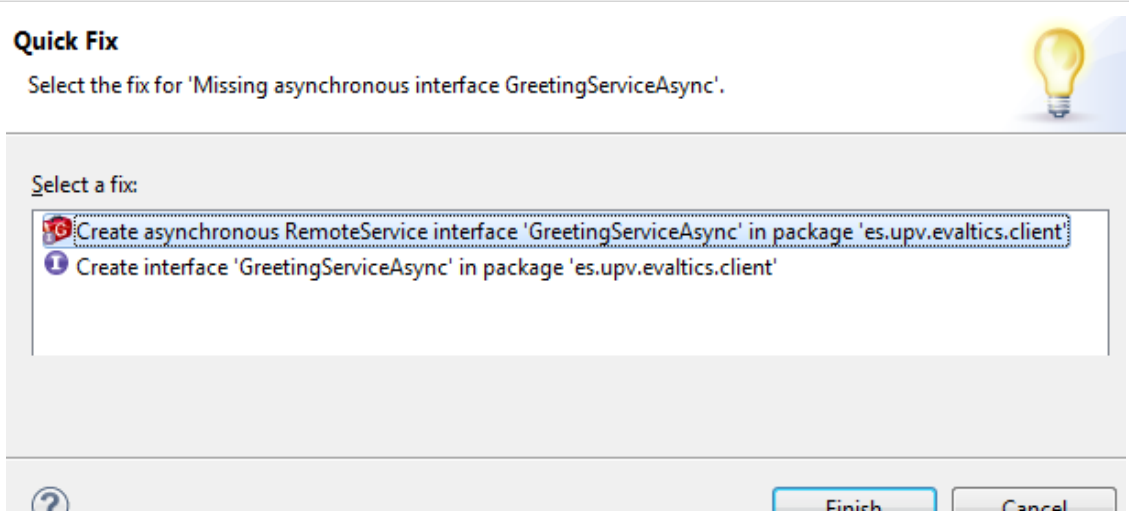


Ilustración 10: Resolución de problemas en Eclipse con los interfaces asíncronos

Esto crea una clase interfaz en el paquete es.upv.evaltics.client, con las llamadas disponibles para procedimientos remotos desde el cliente.

Aún es posible que se muestren errores en el proyecto referidos a una clase inexistente *Messages*. Esto es un componente de internacionalización. Para resolverlo ejecutaremos un objetivo de `Maven`, pulsando sobre la opción *Maven build …* en el menú *Run → Run As:*

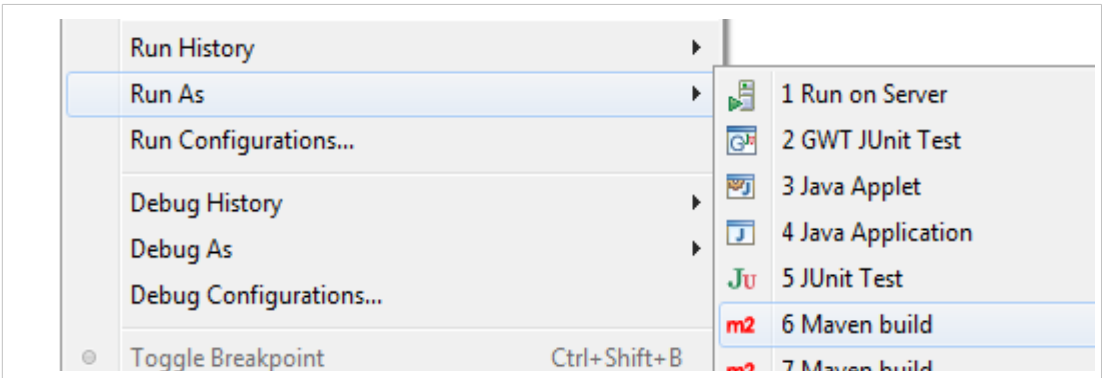


Ilustración 11: Ejecución de un objetivo Maven

En el diálogo, definiremos el objetivo a ejecutar como *gwt:i18n* y pulsaremos *Run*. Esto genera la clase automáticamente con las cadenas adecuadas para la internacionalización del proyecto.

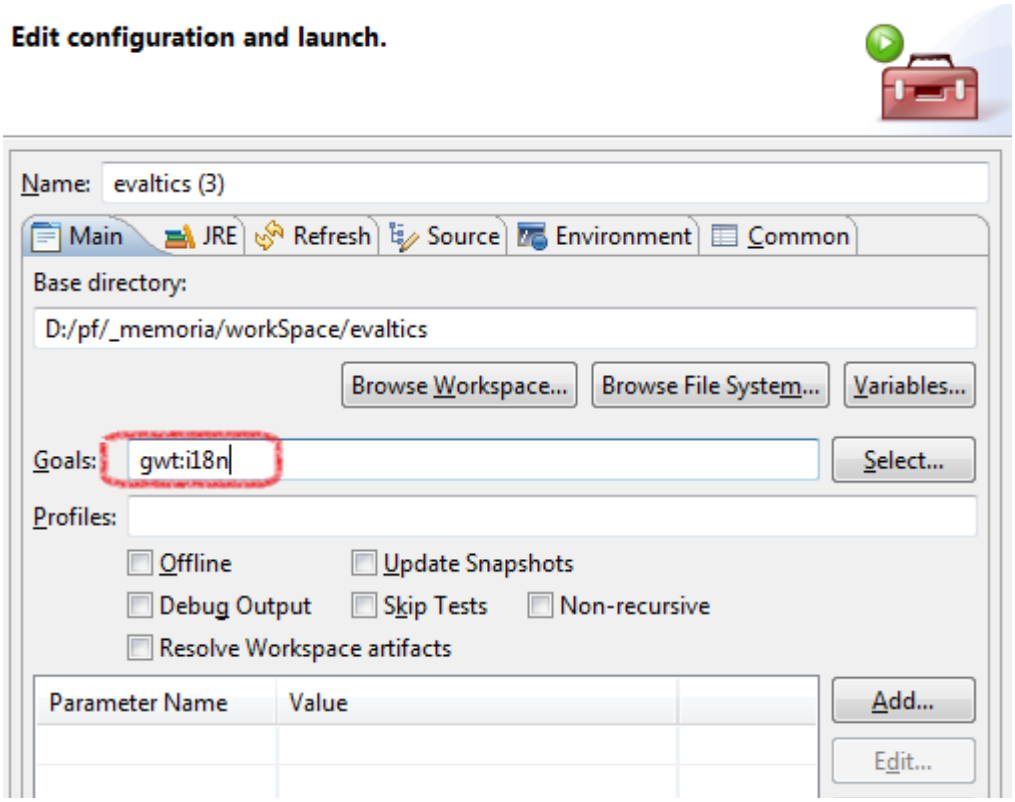


Ilustración 12: Ejecución de objetivo de internacionalización del plugin Maven de GWT

Probando el proyecto

Una vez creada la estructura inicial, probaremos el correcto funcionamiento de la aplicación y el entorno. Para ello ejecutaremos la opción *File → Run As → Web Application* desde el menú o en el menú contextual del proyecto `Evaltics`.

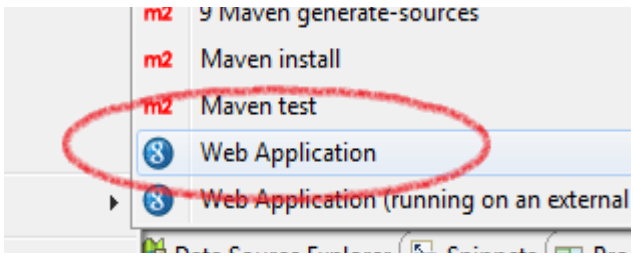


Ilustración 13: Probando la primera versión del proyecto

Esto lanzará la aplicación dentro del entorno de `Eclipse` en modo desarrollo, lo que en `GWT` se llama *"hosted"* que es una funcionalidad creada para facilitar el desarrollo de aplicaciones, puesto que este proceso no necesita del proceso de compilación y transformación del código `Java` a `Javascript` para ser probada la aplicación.

Una vez ejecutado, si todo es correcto, `GWT` lanza un servidor `Jetty` para publicar la aplicación. En la pestaña *Development Mode* de `Eclipse` aparece la dirección desde donde se puede acceder a la aplicación.

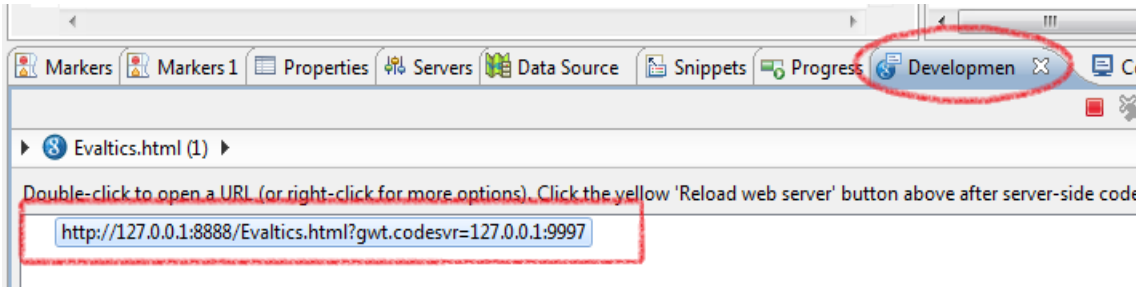
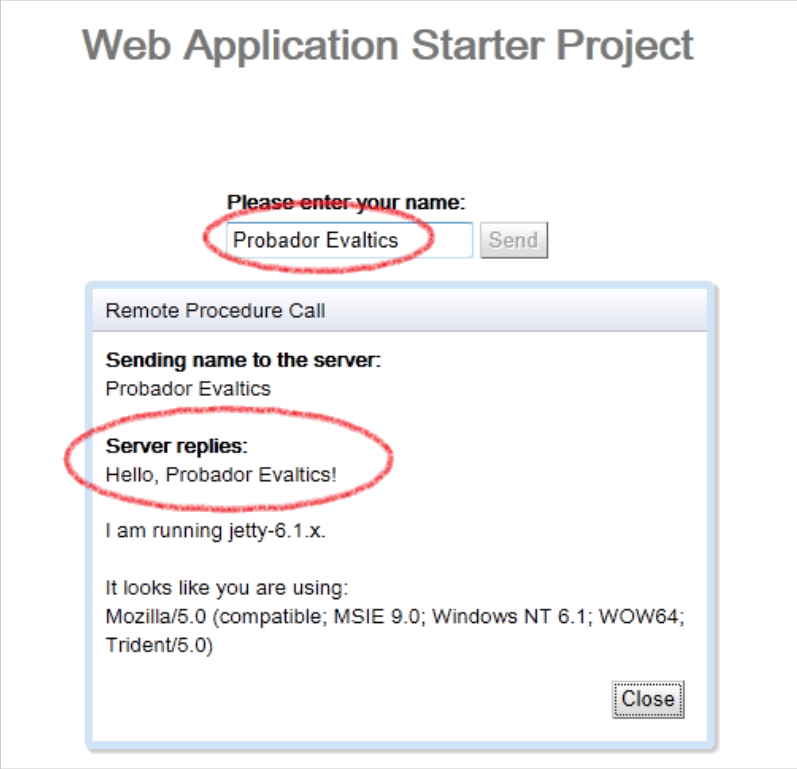


Ilustración 14: Primera prueba de funcionamiento

Pulsando sobre la dirección abrirá un navegador accediendo a la *url.* En nuestro caso de ejemplo es una aplicación que pide un nombre y al pulsar sobre *Send,* abre un diálogo de respuesta:



NOTA: al acceder a una dirección de una aplicación ejecutada en *"Hosted Mode"* por primera vez, el navegador solicitará la instalación de un *plugin.* Esto es necesario para el modo de desarrollo pero no para la ejecución de las aplicaciones una vez desplegadas en modo de producción.

Ilustración 15: Solicitud de plugin GWT para Internet Explorer

La aplicación internamente realiza una llamada a un procedimiento remoto (RPC) que simplemente devuelve el texto *"Hello,"* y el texto introducido.

Observando el proyecto, vemos como la parte servidora que responde al mensaje se encuentra en la función `greetServer`, en la clase `es.upv.evaltics.server.GreetingServiceImpl`. Esta función obtiene un argumento de tipo *String* y devuelve un resultado de tipo *String* con el mensaje finalmente mostrado. La clase está implementando el interfaz `GreetingService` que corresponde a la definición del contrato del servicio. Esta se encuentra en `es.upv.evaltics.client.GreetingService`:

```
/**
 * The client side stub for the RPC service.
 */
@RemoteServiceRelativePath("greet")
public interface GreetingService extends RemoteService {
  String greetServer(String name) throws IllegalArgumentException;
}
```

Listado 1 : Definición del contrato de un servicio en GWT

Y define el nombre del servicio con la anotación `@RemoteServiceRelativePath("greet")`.

La parte cliente se encuentra en la clase *Evaltics.java,* que es el código Java que se traducirá a Javascript para ser mostrado en el navegador. Esta clase obtiene una referencia al servicio `greet`, en la llamada

```
/**
   * Create a remote service proxy to talk to the server-side Greeting service.
   */
          private final GreetingServiceAsync greetingService =
GWT.create(GreetingService.class);
```

Listado 2 : Código cliente para obtener una referencia al un servicio remoto

y ejecuta la llamada

```
greetingService.greetServer(textToServer, new AsyncCallback<String>() {
          public void onFailure(Throwable caught) {
            // Show the RPC error message to the user
            dialogBox.setText("Remote Procedure Call - Failure");
            serverResponseLabel.addStyleName("serverResponseLabelError");
            serverResponseLabel.setHTML(SERVER_ERROR);
            dialogBox.center();
```

```
            closeButton.setFocus(true);
          }

          public void onSuccess(String result) {
            dialogBox.setText("Remote Procedure Call");
            serverResponseLabel.removeStyleName("serverResponseLabelError");
            serverResponseLabel.setHTML(result);
            dialogBox.center();
            closeButton.setFocus(true);
          }
        });
```

Listado 3 : Código del cliente para llamar a un servicio remoto

El servicio tiene un método `greetServer` que recibe una cadena y devuelve una llamada a una función a ejecutar cuando el servicio remoto responde. Los servicios siempre devuelven la respuesta en una llamada a una función de tipo `AsyncCallback` con el parámetro del tipo devuelto. Estas llamadas deben implementar los métodos `onFailure` y `onSuccess` según la llamada al servicio haya sido errónea o satisfactoria.

Finalmente, el cliente muestra un diálogo con el mensaje devuelto. Este diálogo es un control de la librería de *widgets* de `GWT.`

Como vemos, todo el código es `Java` pero en el navegador se ejecuta como código `Javascript.`

6. Desarrollando el cliente de la aplicación

GWT es una tecnología para la creación de interfaces de usuario que se ejecutan en un navegador y por tanto se centra en la parte cliente de una aplicación cliente-servidor. La parte cliente se comunica con la parte servidor por medio de llamadas a procedimientos remotos (RPC) en su versión básica, pero GWT no impone restricciones a la tecnología usada en la parte servidora, pudiéndose utilizar la más diversas de las soluciones.

La creación de la parte cliente se desarrolla utilizando el lenguaje Java y es tiene un proceso de traducción a Javascript para ser mostrado en un navegador web. Este proceso es transparente y no es necesario el conocimiento de Javascript para desarrollar una aplicación completamente funcional. En cambio la plataforma sí provee de mecanismos para ejecutar código Javascript directamente desde las propias clases Java.

GWT consiste en una serie de librerías y *toolkits* gráficos para el desarrollo de los interfaces. Estas librerías proveen de botones, etiquetas, listados o formularios que luego serán mostrados en el navegador web. Estas librerías pueden ser mejoradas creando nuevos controles. Una de las librerías más utilizadas y que incrementa la productividad de los controles por defecto de GWT es SmartGWT (http://code.google.com/p/smartgwt/downloads) siendo la utilizada en el proyecto. Es perfectamente posible la integración de diversas librerías, por ejemplo, la utilización de los controles SmartGWT con los controles estándar de la plataforma.

Básicamente, una aplicación en GWT consiste en una página HTML donde se ejecuta código Javascript que se ha traducido desde código escrito en Java. Dentro del "lienzo" que representa la página HTML se van construyendo las distintas pantallas pero todas dentro de la misma página HTML. Por tanto una aplicación en GWT puede consistir perfectamente en un único archivo HTML donde las distintas pantallas corresponden a distintos paneles que se muestran y ocultan por medio de Javascript.

Una aplicación en GWT consiste en un módulo GWT, con uno o diversos puntos de entrada desde donde se desarrolla la actividad de la aplicación. Los módulos se especifican mediante ficheros *.gwt.xml* y en ellos se especifica el nombre del módulo, las diversas librerías utilizadas, así como los paquetes Java que forman el código cliente y el servidor. Un ejemplo de fichero *.gwt.xml* es:

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='evaltics'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.  You can change       -->
  <!-- the theme of your GWT application by uncommenting          -->
  <!-- any one of the following lines.                            -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <inherits name="com.google.gwt.place.Place"/>
  <inherits name="com.google.gwt.activity.Activity"/>
  <!-- <inherits name='com.google.gwt.user.theme.standard.Standard'/> -->
  <!-- <inherits name='com.google.gwt.user.theme.chrome.Chrome'/> -->
  <!-- <inherits name='com.google.gwt.user.theme.dark.Dark'/>     -->

  <!-- Other module inherits                                      -->

  <!-- Specify the app entry point class.                         -->
  <entry-point class='es.upv.evaltics.client.EvalticsPre'/>
```

```
    <!-- Specify the paths for translatable code                    -->
    <source path='client'/>
    <source path='shared'/>

  </module>
```

Listado 4 : Especificación de un módulo GWT

El *plugin* de `Google` para `Eclipse` contiene opciones que facilitan y simplifican la creación y actualización de estos ficheros.

Para el desarrollo de una aplicación en `GWT`, se ha integrado herramientas para implementar el paradigma de programación *MVP,* Modelo Vista Presentador.

Modelo *MVP*

Este modelo, heredero del modelo-vista-controlador más conocido, impone más restricciones en la capa de la vista, obligando a que toda la lógica sea desarrollada en el presentador y la vista no tenga conocimiento del modelo de datos.

El modelo *MVP* intenta que la vista sea lo más simple posible con el objetivo de simplificar los tests de la aplicación, puesto que la vista no necesita tests (al ser tan simple) y todos los tests se concentran en el presentador, que no dependen en absoluto de la interfaz gráfica y permite realizar mejor las pruebas.

La clase presentador se coloca entre la vista (en el proyecto, la interfaz gráfica desarrollada en `GWT` con la librería `SmartGWT`) y el modelo de datos (en el proyecto, las distintas entidades). La vista contiene métodos para mostrar los datos, pero sin conocimiento de como se obtienen esos datos ni que representan. También tiene métodos para extraer los datos introducidos. Todas las acciones de la vista deben pasar por el presentador.

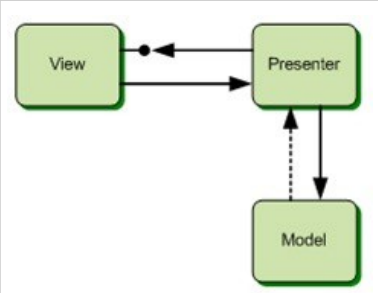


Ilustración 16: Modelo MVP

Las características resumidas son:

- La vista no conoce el modelo.
- El presentador es independiente de la tecnología de interfaz de usuario.
- Se pueden realizar pruebas sobre la vista y el presentador puesto que esta basada en un contrato.

`GWT` no impone restricciones sobre este modelo de programación. Es sólo una convención para generar mejor código. El modelo implementado en `GWT` simplifica el hecho de cambiar de sección dentro de una misma aplicación, por ejemplo, cambiar de la vista de edición de datos académicos (alumnos, matrículas, …) a la vista de gestión de evaluaciones.

En GWT el modelo se implementa mediante actividades, lugares y vistas. Una actividad *(activity)* es una acción que el usuario está realizando (un alumno realizando una evaluación). Es un concepto independiente de la vista utilizada para realizar la acción.

Un lugar *(place)* es una manera de acceder a una actividad, por ejemplo, acceder a la actividad de realizar una evaluación. Dentro del navegador se concreta por una dirección URL y un *token* para diferenciarlo (por ejemplo: http://host/Evaltics.html#panelAlumno).

La vista es lo que se muestra al usuario para realizar una actividad. Son los controles que permiten interactuar con la aplicación. Las vistas deben implementar la interfaz `IsWidget`.

En lo siguiente, se creará un esqueleto básico para la estructura *MVP* para el proyecto `Evaltics`.

Generando la estructura de una aplicación *MVP*

Para implementar el modelo *MVP* se crea una serie de clases que sirven de unión entre cada uno de las capas.

- `es.upv.evaltics.client.ClientFactory`: Es un interfaz que se utiliza para facilitar la unión de todos los componentes.

- `es.upv.evaltics.client.ClientFactoryImpl`: Es la implementación del interfaz anterior. Contiene referencias a las vistas instanciadas, así como al bus de eventos y manejador de lugares. Este manejador (`PlaceController`) se encarga de la gestión de actividades, evitando que se pase a una actividad que pueda estar en proceso.

- `es.upv.evaltics.client.mvp.AppPlaceHistoryMapper`: Esta clase declara todos los lugares disponibles que son accesibles en la aplicación. Extiende el interfaz `PlaceHistoryMapper` y se deben especificar los lugares con la anotación `@WithTokenizers`.

```
   @WithTokenizers({PanelEvaluaciones.Tokenizer.class,
PanelAlumnoPlace.Tokenizer.class })
```

- `es.upv.evaltics.client.mvp.AppActivityMapper`: Es la correspondencia entre las actividades y los lugares. Implementa el interfaz `ActivityMapper` y debe implementar la función `getActivity` donde a partir de un lugar solicitado devuelve la actividad asociada. Es la actividad la que devolverá la vista que se mostrará finalmente al usuario.

Las actividades deben implementar la interfaz `com.google.gwt.activity.shared.Activity` y contienen lógica para iniciar una vista, mostrando una interfaz de usuario. La vista se comunica con la actividad cuando recibe eventos desde el navegador web, por ejemplo cuando se pulsa un botón de guardar y es la actividad la que contiene la lógica de realizar la acción.

Las actividades inician la vista en la funcion `start` y hacen de intermediario con la vista para cambiar de actividad y mostrar otra pantalla al usuario.

Los lugares *(Places)* es un objeto que representa un estado particular de la interfaz de usuario. Un place puede convertirse a y desde un *token* de historia URL mediante la definición de un `PlaceTokenizer` para cada lugar, y el `PlaceHistoryHandler` actualiza automáticamente la URL del navegador correspondiente a cada lugar en la aplicación.

Una implementación de un lugar debe implementar el interfaz `com.google.gwt.place.shared.Place` y debe tener un `PlaceTokenizer` asociado que sabe cómo serializar el estado del place a un *token* URL.

Una aplicación en `GWT` se inicia con la llamada a la función `onModuleLoad` de todos los "*entry point*" definidos en el fichero de módulo. Los "*entry point*" o puntos de entrada son clases Java que implementan la interfaz `EntryPoint`. En el caso del proyecto, el único punto de entrada está en el fichero `Evaltics.java` como se puede ver en la definición del módulo `Evaltics.gwt.xml`:

```
<!-- Specify the app entry point class.                         -->
  <entry-point class='es.upv.evaltics.client.Evaltics' />
```

Para el funcionamiento correcto de la aplicación *MVP* se edita el código del punto de entrada `Evaltics` *(Evaltics.java)* y se modifica para relacionar todas las clases definidas anteriormente:

```
public class Evaltics implements EntryPoint {

   private SimplePanel appWidget = new SimplePanel();
   private Place defaultPlace = new PanelAlumnoPlace("Go!");

   public void onModuleLoad() {
          // Create ClientFactory using deferred binding so we can replace with
          // different impls in gwt.xml
          ClientFactory clientFactory = GWT.create(ClientFactory.class);
          EventBus eventBus = clientFactory.getEventBus();
          PlaceController placeController = clientFactory.getPlaceController();
          // Start ActivityManager for the main widget with our ActivityMapper
          ActivityMapper activityMapper = new AppActivityMapper(clientFactory);
          ActivityManager activityManager = new ActivityManager(activityMapper,
                        eventBus);
          activityManager.setDisplay(appWidget);
          // Start PlaceHistoryHandler with our PlaceHistoryMapper
          AppPlaceHistoryMapper historyMapper = GWT
                        .create(AppPlaceHistoryMapper.class);
          PlaceHistoryHandler historyHandler = new PlaceHistoryHandler(
                        historyMapper);
          historyHandler.register(placeController, eventBus, defaultPlace);
          RootPanel.get().add(appWidget);
          // Goes to place represented on URL or default place
          historyHandler.handleCurrentHistory();
   }

}
```

Listado 5 : Código del entrypoint de la aplicación con el modelo MVP

Finalmente es necesario modificar el fichero de definición de módulo `Evaltics.gwt.xml` y añadir con que clase se va a instanciar `ClientFactory` dentro del entorno cliente:

```
   <replace-with class="es.upv.evaltics.client.ClientFactoryImpl">
          <when-type-is class="es.upv.evaltics.client.ClientFactory"/>
   </replace-with>
```

Creación de actividades y vistas

Para la creación de nuevas vistas y actividades, el *plugin* de `Google` simplifica la tarea permitiendo crearlas desde el menú contextual pinchando en el paquete correspondiente al módulo cliente.

Antes de crear una actividad y un lugar asociado, crearemos el paquete `es.upv.evaltics.client.activity` y `es.upv.evaltics.client.place` mediante la opción *New → Package* pulsando sobre el paquete `es.upv.evaltics.client` y accediendo al menú contextual

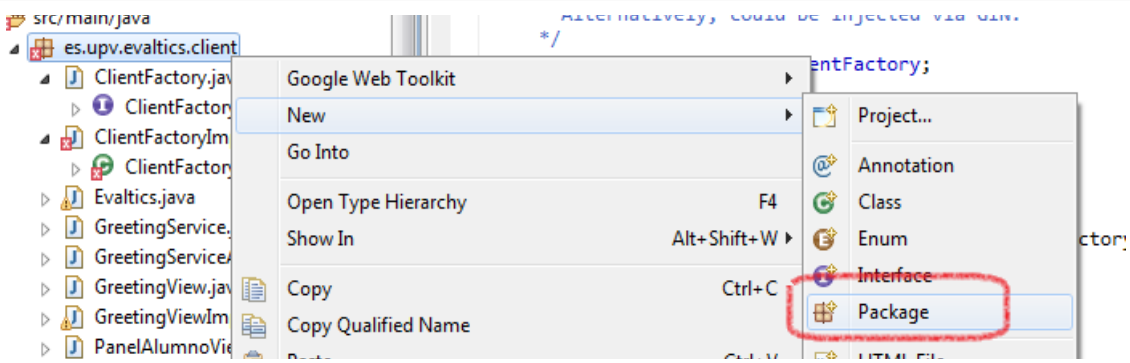


Ilustración 17: Creación de un paquete

Una vez creados los dos paquetes, tendremos la siguiente estructura de proyecto:

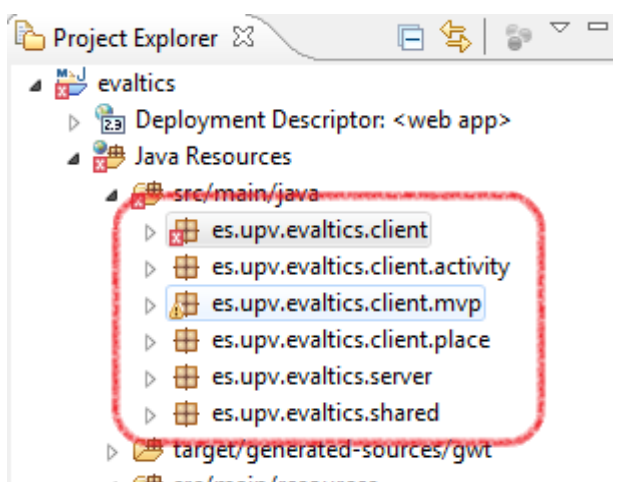


Ilustración 18: Estructura de paquetes de la aplicación

Ahora crearemos una actividad para la realización de evaluaciones por parte de un alumno. Pulsando sobre el paquete `es.upv.evaltics.client` y seleccionado la opción *MVP View*

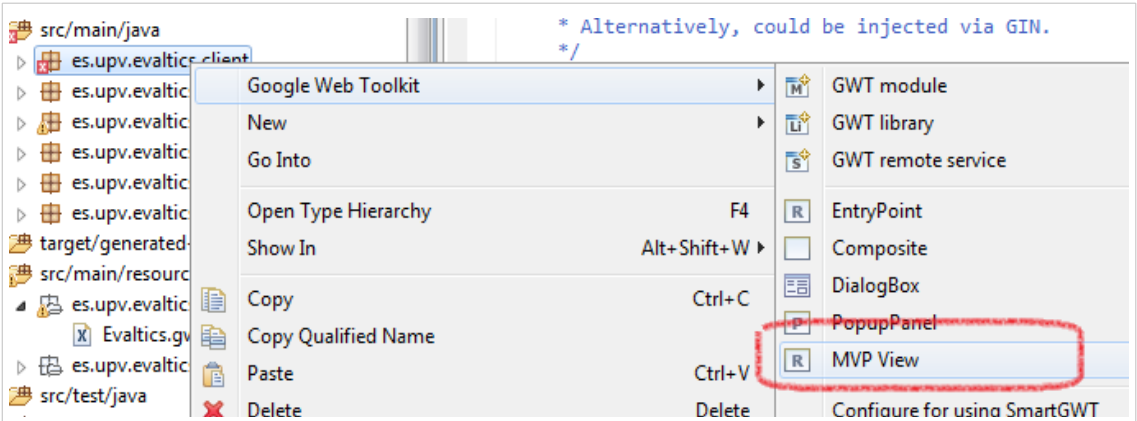


Ilustración 19: Crear un nueva vista MVP

accedemos al cuadro para crear una nueva vista *MVP*. Crearemos una nueva vista con el nombre `PanelAlumnoView` y las opciones mostradas a continuación

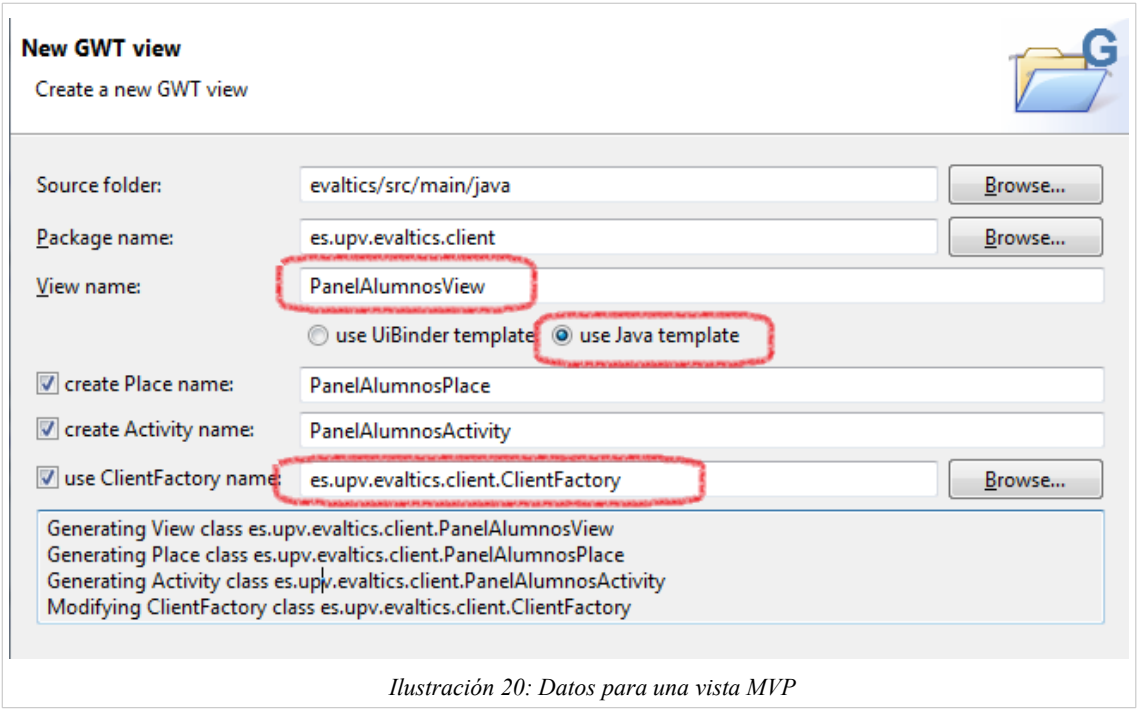


Ilustración 20: Datos para una vista MVP

Esto crea las distintas clases en el proyecto:

- **`PanelAlumnosView`** : la interfaz con la que se comunica la actividad con la vista. Contiene acciones para mostrar resultados al usuario. También se crea la implementación `PanelAlumnosViewImpl` que son los controles que se mostrarán en la página web y que se implementarán en `SmartGWT`.

  El interfaz `Presenter` dentro de `PanelAlumnosView` define acciones a realizar por el presentador (la actividad) como respuesta a acciones del usuario. Por ejemplo para el panel de alumnos de realización de evaluaciones contendría un método para la finalización de la evaluación (y otros para la respuesta de una pregunta, carga de una evaluación, etc...).

```
public interface Presenter {
        /**
```

```
             * Navigate to a new Place in the browser.
             */
            void goTo(Place place);

            /* finaliza una evaluación */
            void finalizaEvaluacion (Long evaluacionId);
    }
```

Listado 6 : Interfaz de un presentador en MVP

- **`PanelAlumnosPlace:`** Es la clase capaz de asociar el lugar de panel de alumnos con una url determinada.

- **`PanelAlumnosActivity:`** Es la actividad asociada. Responsable de mostrar la vista al usuario y de mandar acciones al usuario y captar las acciones requeridas por la vista. Implementa la interfaz `PanelAlumnosView.Presenter` y por tanto debe tener el código para la finalización de una evaluación:

```
@Override
   public void finalizaEvaluacion(Long evaluacionId) {
          /* realiza comprobaciones y llama a procedimiento remoto */

   }
```

  La implementación real de la función consistirá en una comunicación con la parte servidora para dar notificación de la finalización de una evaluación si se cumplen las condiciones adecuadas.

- **`ClienteFactory:`** se modifica para añadir métodos para obtener una instancia de la vista:

```
public PanelEvaluacionesView getPanelEvaluacionesView();
```

También es necesario modificar la interfaz `AppPlaceHistoryMapper` para añadir los nuevos lugares creados

```
   @WithTokenizers({PanelEvaluacionesPlace.Tokenizer.class,
PanelAlumnosPlace.Tokenizer.class })
```

Y modificar la clase `AppActivityMapper` para mapear la nueva actividad creada con el lugar :

```
@Override
   public Activity getActivity(Place place) {

     if (place instanceof PanelAlumnosPlace)
       return new PanelAlumnosActivity((PanelAlumnosPlace) place, clientFactory);

      return null;
   }
```

Listado 7 : Mapeo de actividades con lugares en MVP

Una vez creadas las vistas de `PanelAlumno` y `PanelEvaluación` el proyecto tiene la siguiente estructura:

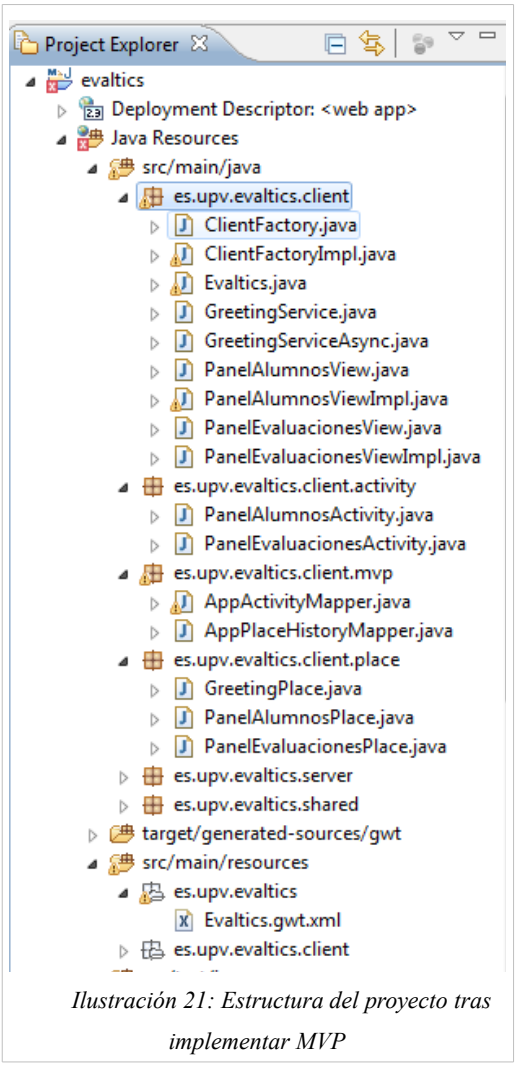


Ilustración 21: Estructura del proyecto tras implementar MVP

7. Desarrollo de la parte servidora

Aunque es posible realizar una aplicación GWT sólo con la parte cliente, generalmente las aplicaciones necesitan comunicarse con una servidor para diversas tareas como guardar datos en base datos que son imposibles de realizar desde el navegador web del cliente. Esto es porque la parte servidora se ejecuta en la máquina donde se muestra la página y no tiene acceso directo a ningún recurso del servidor.

La comunicación entre el cliente y el servidor en un entorno GWT se realiza mediante llamadas AJAX lo que permite que se intercambien datos con el servidor sin necesidad de recargar la página web.

GWT dispone de distintos protocolos para la comunicación cliente-servidor: llamadas RPC, JSON, XML, …

Para el proyecto se han utilizado las tecnologías JSON y RPC, mediante la publicación de servicios con un *servlet* basado en Spring.

Para la publicacion de servicios RPC se utiliza la librería *spring4gwt* disponible en http://code.google.com/p/spring4gwt/ que proporciona la inyección de dependencias propia de Spring a un proyecto GWT. Permite el acceso de servicios desde el cliente GWT a beans de Spring y está implementado como un *servlet* ligero sin apenas dependencias.

La librería spring4gwt debe descargarse y copiarse al directorio src\main\webapp\WEB-INF\lib del proyecto, puesto que no existe en los repositorios de Maven y no se puede gestionar las dependencias automáticamente.

Una vez copiada la librería, configuraremos el proyecto para integrar Spring. Añadiremos las dependencias al fichero *pom.xml*. Esto podemos realizarlo editando directamente el fichero *pom.xml* o mediante la opción *Maven → Add dependency* en el menú contextual del proyecto.

A continuación editaremos el fichero *pom.xml* de la siguiente forma, que ya incluyen las dependencias para Spring, Hibernate, CXF y la seguridad que serán necesarias para pasos posteriores:

Dentro del elemento dependecies del fichero *pom.xml* añadiremos:

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-orm</artifactId>
	<version>${org.springframework.version}</version>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-web</artifactId>
	<version>${org.springframework.version}</version>
</dependency>
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-core</artifactId>
	<version>${org.hibernate.version}</version>
</dependency>
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-annotations</artifactId>
	<version>${org.hibernate.version}</version>
</dependency>
```

```
            <dependency>
                   <groupId>org.hibernate</groupId>
                   <artifactId>hibernate-entitymanager</artifactId>
                   <version>${org.hibernate.version}</version>
            </dependency>
            <dependency>
                   <groupId>org.hibernate.javax.persistence</groupId>
                   <artifactId>hibernate-jpa-2.0-api</artifactId>
                   <version>1.0.0.Final</version>
            </dependency>
            <dependency>
                   <groupId>org.slf4j</groupId>
                   <artifactId>slf4j-log4j12</artifactId>
                   <version>1.5.8</version>
            </dependency>
            <dependency>
                   <groupId>org.hsqldb</groupId>
                   <artifactId>hsqldb</artifactId>
                   <version>1.8.0.10</version>
            </dependency>
            <dependency>
                   <groupId>c3p0</groupId>
                   <artifactId>c3p0</artifactId>
                   <version>0.9.1.2</version>
            </dependency>
            <dependency>
                   <groupId>org.springframework</groupId>
                   <artifactId>spring-jpa</artifactId>
                   <version>2.0.8</version>
            </dependency>
            <dependency>
                   <groupId>net.sf.ehcache</groupId>
                   <artifactId>ehcache-core</artifactId>
                   <version>2.3.1</version>
            </dependency>
            <dependency>
                   <groupId>org.hibernate</groupId>
                   <artifactId>hibernate-ehcache</artifactId>
                   <version>3.5.1-Final</version>
            </dependency>
            <dependency>
                   <groupId>org.hibernate</groupId>
                   <artifactId>hibernate-validator</artifactId>
                   <version>4.3.0.Final</version>
            </dependency>
            <dependency>
              <groupId>org.apache.cxf</groupId>
              <artifactId>cxf-rt-frontend-jaxrs</artifactId>
              <version>${cxf.version}</version>
              <exclusions>
                  <exclusion>
                      <artifactId>spring-core</artifactId>
                      <groupId>org.springframework</groupId>
                  </exclusion>
                  <exclusion>
                      <artifactId>spring-web</artifactId>
                      <groupId>org.springframework</groupId>
                  </exclusion>
                  <exclusion>
                      <artifactId>wstx-asl</artifactId>
                      <groupId>org.codehaus.woodstox</groupId>
                  </exclusion>
              </exclusions>
          </dependency>
           <dependency>
              <groupId>org.codehaus.jackson</groupId>
```

```
            <artifactId>jackson-jaxrs</artifactId>
            <version>1.9.9</version>
        </dependency>
        <dependency>
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-web</artifactId>
          <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-config</artifactId>
          <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
          <groupId>javax.servlet.jsp</groupId>
                <artifactId>jsp-api</artifactId>
                <version>2.1</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-webmvc</artifactId>
          <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
                <groupId>org.tuckey</groupId>
          <artifactId>urlrewritefilter</artifactId>
          <version>3.2.0</version>
         </dependency>
```

Listado 8 : Dependencias Maven para el proyecto

Cada uno de los elementos `dependency` representa una dependencia que utilizará el proyecto. Maven resolverá todas las distintas dependencias de segundo y siguientes niveles automáticamente.

También debemos añadir unas variables al fichero `pom.xml`

```
    <org.springframework.version>3.0.7.RELEASE</org.springframework.version>
    <org.hibernate.version>3.5.1-Final</org.hibernate.version>
    <cxf.version>2.6.0</cxf.version>
```

Dentro del elemento `<properties>`

También es posible que sea necesario ajustar las propiedades de versión de `Java` del *plugin maven-compiler*. Para ello buscamos el elemento

```
    <artifactId>maven-compiler-plugin</artifactId>
```

y establecemos las opciones de configuración para la versión de `Java` que dispongamos:

```
<configuration>
   <source>1.6</source>
   <target>1.6</target>
</configuration>
```

En estos momentos podemos ejecutar la aplicación y comprobar que se ejecuta aunque aún no hemos publicado todavía ningún servicio.

Dentro del proyecto `Evaltics` se utilizan dos tipos de comunicación entre cliente y servidor:

- Llamadas `RPC:` tipo de comunicación estándar en `GWT` permite la serialización de objetos que implementen el interfaz IsSerializable de forma transparente. Estos servicios se publican mediante el servlet `springGwtRemoteServiceServlet` definido en *web.xml* y que está implementado en la librería *spring4gwt.*

- Llamadas a servicios `REST` mediante comunicación con datos `JSON.`

En lo sucesivo vamos a ver como configurar servicios `RPC` y servicios `REST` para la aplicación `Evaltics.` Los *servlets* responsables de servir estos servicios se configuran en el fichero *web.xml.* Primeramente ajustaremos el fichero *web.xml* modificando la cabecera y quedando como:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
       version="2.5">

       <!-- Servlets -->

       <!-- Default page to serve -->
       <welcome-file-list>
              <welcome-file>Evaltics.html</welcome-file>
       </welcome-file-list>

    </web-app>
```

Listado 9 : Versión inicial del fichero de configuración de la aplicación web

Servicios RPC

`GWT RPC` hace que sea fácil para el cliente y el servidor pasar objetos Java de ida y vuelta a través de HTTP puesto que es capaz de serializar de forma transparente los objetos Java y convertirlos en objetos `Javascript` para su interactuación en el navegador web.

Para definir un servicio RPC se crea un interfaz que extiende a `RemoteService` y se declara la dirección donde será accesible por medio de la anotación RemoteServiceRelativePath. Para el proyecto `Evaltics` crearemos el servicio `EvaluacionService` que contendrá métodos para interactuar con las evaluaciones.

```
    @RemoteServiceRelativePath("springGwtServices/evaluacionService")
    public interface EvaluacionService extends RemoteService {

       public void guardaRespuesta (Long evaluacionAlumnoID, RespuestaAlumno
respuesta);

       public void finalizarEvaluacion (Evaluacion evaluacion);
    }
```

Listado 10 : Definición de un servicio RPC

El servicio lo creamos como relativo a la dirección `springGwtServices` puesto que el servicio va a ser servido por un *servlet* que atenderá a las peticiones que empiezan por `springGwtServices.` Esto lo debemos configurar en el fichero *web.xml* y añadir el código:

```
	<servlet>
		<servlet-name>springGwtRemoteServiceServlet</servlet-name>
		<servlet-
class>org.spring4gwt.server.SpringGwtRemoteServiceServlet</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>springGwtRemoteServiceServlet</servlet-name>
		<url-pattern>/evaltics/springGwtServices/*</url-pattern>
	</servlet-mapping>
```

Listado 11 : Servlet para servir servicios RPC con Spring

Donde se especifica que el *servlet* `springGwtRemoteServiceServlet` es implementado por la clase `Java org.spring4gwt.server.SpringGwtRemoteServiceServlet` y responderá a peticiones que comiencen con `/evaltics/springGwtServices.`

Hasta ahora tenemos un interfaz que nos dice la funcionalidad que será pública para el servicio. Esta funcionalidad debe ser implementada por una clase que implemente el interfaz `EvaluacionService` y que extiende a `RemoteServiceServlet.` Esta funcionalidad se ejecutará exclusivamente en la parte servidora, por tanto debe ir en el paquete `es.upv.evaltics.server.`

```
import es.upv.evaltics.client.EvaluacionService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

public class EvaluacionServiceImpl extends RemoteServiceServlet implements
EvaluacionService {

	@Override
	public void finalizarEvaluacion() {
		// TODO implementar la funcionalidad como lógica de negocio

	}
}
```

Listado 12 : Definición de un servicio RPC

Una vez definido el servicio, su interfaz y su implementación falta definir el contrato que debe obtener el cliente para utilizar este servicio. Este contrato estará formado por funciones que devuelven una función que será ejecutada por el cliente al terminar la ejecución del servicio. Esto es así, porque **las llamadas a servicios en `GWT` siempre son asíncronas**.

En Eclipse si abrimos el fichero `EvaluacionService.java` mostrará que contiene errores, puesto que no encuentra la clase `EvaluacionServiceAsync` que es la definirá las llamadas asíncronas que se hacen desde el cliente. Una forma fácil de solucionar los errores es pulsando

sobre un error `EvaluacionServiceAsync` y pulsando *Ctrl + 1* o sobre la opción *Quick Fix* del menú contextual.

Esto nos da la opción de crear la interfaz `EvaluacionServiceAsync` directamente en el paquete `es.upv.evaltics.client`.

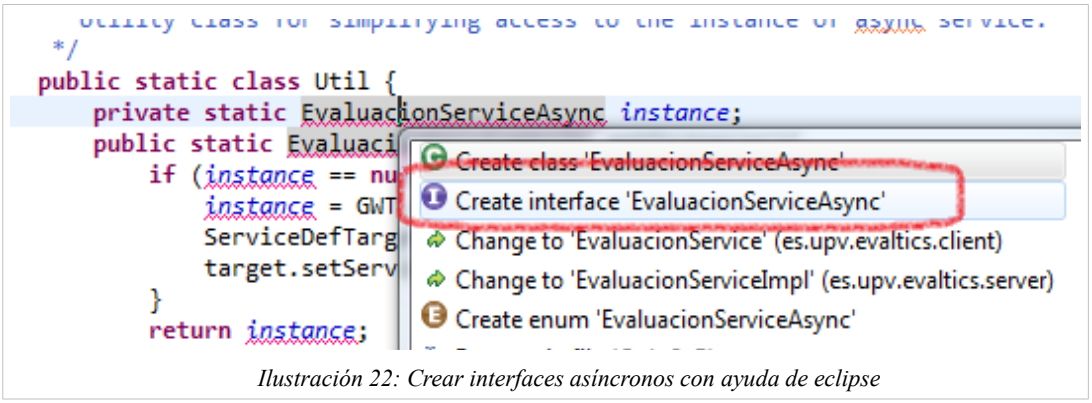


Ilustración 22: Crear interfaces asíncronos con ayuda de eclipse

Al abrir el fichero EvaluacionServiceAsync.java nos muestra errores, esto es debido a que no contiene las declaraciones de las llamadas a las funciones definidas por el servicio. Para solucionar esto pulsaremos sobre el nombre de la clase remarcado y ejecutaremos las opciones que nos sugiere Eclipse para cada uno de los métodos que falten.

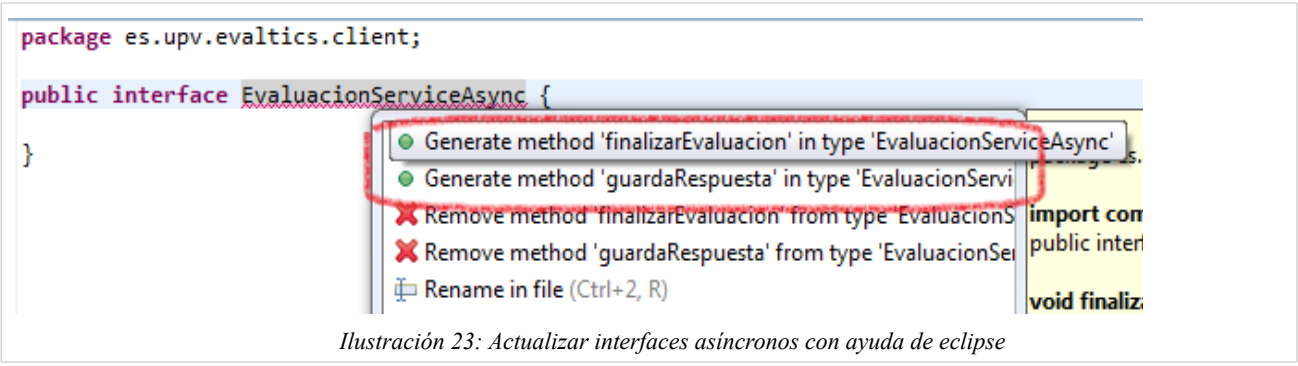


Ilustración 23: Actualizar interfaces asíncronos con ayuda de eclipse

Esto genera una declaración que contiene los parámetros de la función del servicio y un parámetro final que será la respuesta a la llamada al servicio de tipo `AsyncCallback`:

```
import com.google.gwt.user.client.rpc.AsyncCallback;

import es.upv.evaltics.shared.RespuestaAlumno;

public interface EvaluacionServiceAsync {

   void finalizarEvaluacion(AsyncCallback<Void> callback);

   void guardaRespuesta(Long evaluacionAlumnoID, RespuestaAlumno respuesta,
                AsyncCallback<Void> callback);

}
```

Listado 13 : Servicio asíncrono

El cliente obtendrá una referencia a un objeto de tipo `EvaluacionServiceAsync` y ejecutará las funciones del servicio, pasando como último argumento una función que será ejecutada al terminar la ejecución del servicio.

Servicios REST

Otra forma de realizar llamadas al servidor en una aplicación GWT es mediante la llamada a servicios `REST`. Para el proyecto `Evaltics` se han creado servicios de este tipo para la información que se considera de tipo académico como los datos de centros, asignaturas y alumnos. Un servicio `REST` es más adecuado en entorno heterogéneos generales puesto que es una tecnología más abierta que `GWT RPC` al no estar basada en protocolos propietarios. Los servicios REST se basan en la ejecución de métodos de HTTP mediante el paso de datos codificada en formato `XML` o en formato `JSON`. El formato `JSON` es un estándar para la notación de objetos `javascript` que hace que su uso en navegadores web sea muy sencilla y muy directa a partir de una cadena de texto.

Para la publicación de servicios `REST` mediante `Spring` es necesaria una librería que convierta, serialice o deserialice, objetos Java en texto `JSON` que describa el mismo objeto. Para el proyecto se ha utilizado la librería `Jackson`, conocida por su buen rendimiento.

Dentro de las dependencias anteriormente agregadas al fichero *pom.xml* estaba la de la librería `Jackson`:

```
<dependency>
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-jaxrs</artifactId>
            <version>1.9.9</version>
</dependency>
```

Listado 14 : Dependencias Maven para la librería de uso JSON

Para la publicación de los servicios se utiliza la librería `Apache CXF`. La librería `Apache CXF` es un *framework* completo, de código abierto para servicios web.

CXF frecuentemente se emplea en proyectos de infraestructura con arquitecturas orientadas a servicios `(SOA)` y contiene una API para el desarrollo de servicio web: `JAX-WS` y una *API* `JAX-RS` para el desarrollo de servicios web de tipo `RESTful` como los implementados en la aplicación Evaltics.

Las dependencias que se han introducido para la publicación de los servicios en el pom.xml es:

```
        <dependency>
              <groupId>org.apache.cxf</groupId>
              <artifactId>cxf-rt-frontend-jaxrs</artifactId>
              <version>${cxf.version}</version>
        </dependency>
```

Listado 15 : Dependencias para publicar servicios REST

Para la configuración de estos servicios se debe crear el fichero `cxf-servlet.xml` dentro del directorio `src/main/webapp/WEB-INF` y configurarlo de la siguiente manera:

```
    <beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:jaxrs="http://cxf.apache.org/jaxrs"
           xmlns:cxf="http://cxf.apache.org/core"
                       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
                http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd">

        <import resource="classpath:META-INF/cxf/cxf.xml"/>

       <bean id="jsonProvider" class="org.codehaus.jackson.jaxrs.JacksonJsonProvider"
/>

       <!-- Add new endpoints for additional services you'd like to expose -->

       <jaxrs:server address="/api">
              <jaxrs:features>
                     <cxf:logging />
              </jaxrs:features>
              <jaxrs:serviceBeans>
                     <ref bean="centroManager" />
              </jaxrs:serviceBeans>
              <jaxrs:providers>
                     <ref bean="jsonProvider" />
              </jaxrs:providers>
              <jaxrs:extensionMappings>
                     <entry key="json" value="application/json" />
                     <entry key="xml" value="application/xml" />
                     <entry key="feed" value="application/atom+xml" />
              </jaxrs:extensionMappings>
       </jaxrs:server>

    </beans>
```

Listado 16 : Configuración del ficher cxf-servlet.xml

El bean *jsonProvider* referenciado en jaxrs:providers define a Jackson como un proveedor de los datos de los servicios. Es la librería que transformará las peticiones en objetos `Java` y que devolverá los resultado en cadenas con notación `JSON`.

La dirección jaxrs:server address=*"/api"* define la base de los servicios publicados por `CXF`.

El elemento jaxrs:serviceBeans define los *beans* de `Spring` (o de un *framework* `EJB`) que se utilizarán para la publicación de los servicios. Cada uno de estos *beans* está configurado en `Spring` para la aplicación `Evaltics`.

```
              <jaxrs:serviceBeans>
                     <ref bean="centroManager" />
                     <ref bean="cursoManager" />
                     <ref bean="asignaturaManager" />
                     <ref bean="alumnoManager" />
              </jaxrs:serviceBeans>
```

Listado 17 : Definición de servicios en CXF

Vemos como crear el servicio para centros:

```
    package es.upv.server.service;
```

```
import java.util.List;

import javax.jws.WebService;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.springframework.security.access.annotation.Secured;

import es.upv.shared.model.Centro;

@WebService
@Path("/centros")
public interface CentroManager {

   @Secured("ROLE_ALUMNO")
   @Path("/list")
   @GET
   @Produces(MediaType.APPLICATION_JSON)
   List<Centro> list();

   @Path("/save")
   @POST
   @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   @Produces(MediaType.APPLICATION_JSON)
   Centro saveCentro (Centro c);

   @Path("/delete/{centroId}")
   @POST
   @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   @Produces(MediaType.APPLICATION_JSON)
   void deleteCentro (@PathParam("centroId") Long centroId);

}
```

Listado 18 : Creación de un servicio REST

Esta clase está marcada con la anotación `@WebService` que lo marca como que implementa un servicio. Se define la base para las direcciones del servicio mediante `@Path("/centros")` y para cada uno de los métodos del servicio se define la dirección a la que será accesible y su definición de parámetros.

También se debe especificar que tipo de método HTTP soporta cada uno de los métodos mediante las anotaciones `@POST, @GET`, etc...

De esta forma tenemos que el servicio list de centros será accesible en la url `/services/api/centros/list` y que devolverá una lista de objetos Centro en notación `JSON` cuando se llame mediante una llamada *GET* de HTTP.

Una vez creado el interfaz se debe crear el bean que será quien ejecute las llamadas. Mediante una anotación `Spring`, definiremos el bean `centroManager.` Para esto, creamos la clave `CentroManagerImpl` que implementa el servicio `CentroManager` dentro del paquete `es.upv.evaltics.server.service.impl` y se define así:

```
@Service("centroManager")
public class CentroManagerImpl implements CentroManager {
```

```
	CentroDao centroDao;

	public CentroDao getCentroDao () {
		return centroDao;
	}

	@Autowired
	public void setCentroDao (CentroDao centroDao) {
		this.centroDao = centroDao;
	}

	@Override
	public List<Centro> list() {
		List<Centro> list = centroDao.getAll();

		return list;
	}
}
```

Listado 19 : Implementación de servicio REST

Como vemos, la implementación hace una llamada a un objeto *DAO* que se ha obtenido mediante inyección de dependencias de `Spring` y que obtiene una lista de centro de una base de datos.

Una vez implementado el servicio, es necesario hacer que `Spring` reconozca las anotaciones y que cree los *beans* necesarios para que la librería `CXF` pueda utilizarlo. Creamos un fichero de configuración de `Spring` *applicationContext.xml* en *src/main/webapp/WEB-INF* con el siguiente contenido:

```
    <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jee="http://www.springframework.org/schema/jee"
xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:task="http://www.springframework.org/schema/task"
       xsi:schemaLocation="
                        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
                        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
                        http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-3.0.xsd
                        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
                        http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task-3.0.xsd">

       <context:annotation-config/>
       <context:component-scan base-package="es.upv" />

    </beans>
```

Listado 20 : applicationContext.xml

Las opciones de configuración

```
	<context:annotation-config/>
	<context:component-scan base-package="es.upv" />
```

le marcan a Spring que busque clases Java marcadas con la anotación @Service y que las instancia automáticamente para luego ser inyectadas. La librería CXF reconoce un *bean* con el nombre centroManager y lo utilizará para servir peticiones que lleguen a la url …/services/centros/*.

Finalmente es necesario modificar el fichero *web.xml* para que la aplicación responda a las peticiones que comiencen con services mediante el *servlet* de CXF:

```
    <servlet>
		<servlet-name>CXFServlet</servlet-name>
		<servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-
class>
	</servlet>
	<servlet-mapping>
		<servlet-name>CXFServlet</servlet-name>
		<url-pattern>/services/*</url-pattern>
	</servlet-mapping>
```

Listado 21 : Configuración del servlet CXF en web.xml

Y modificar el fichero para añadir los dos fichero de configuración que hemos creado en esta sección:

```
	<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>
			/WEB-INF/applicationContext.xml
			/WEB-INF/cxf-servlet.xml
		</param-value>
	</context-param>
```

Listado 22 : Referencia a la configuración de CXF en web.xml

Llegados a este punto tenemos una aplicación capaz de servir peticiones de servicios RPC y servicios REST. Para probar los servicios RPC necesitamos hacer llamadas desde el cliente GWT. Sin embargo podemos probar los servicios REST accediendo a la url de publicación del servicio una vez arrancada la aplicación GWT.

Para probar los servicios REST, lanzaremos la aplicación mediante Run → Run As → Web Application y accederemos a la url

http://127.0.0.1:8888/services/api/centros/list.json

que debe responder con una lista vacía o con una lista de elementos si hemos definido algún dao.

Si se produce un error al ejecutar, es posible que haya que copiar la librería cxf-rt-frontend-jaxrs-2.6.0.jar al directorio src/main/WEB-INF/lib

La estructura del proyecto debe ser la siguiente:

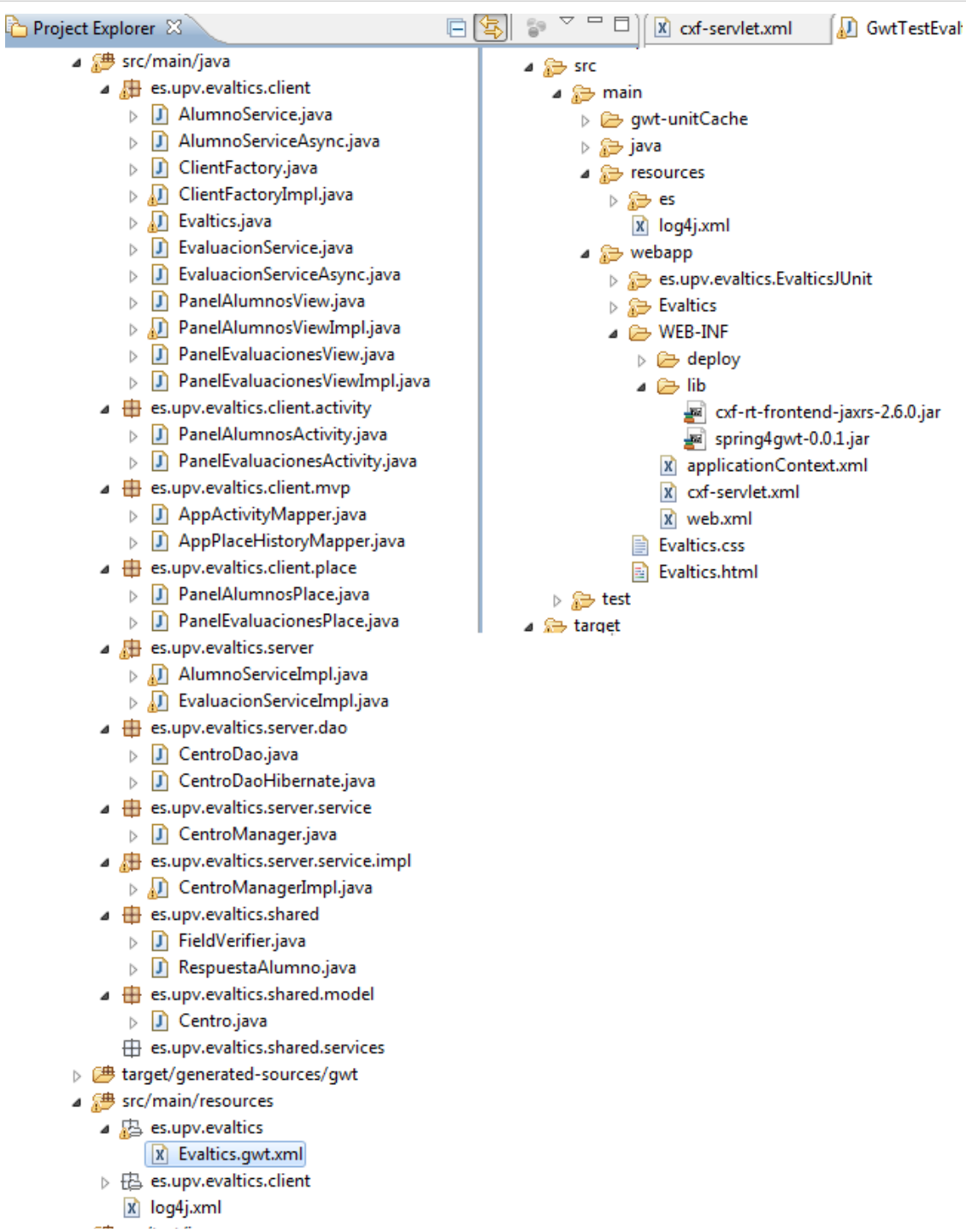


Ilustración 24: Estado del proyecto tras implementar los servicios

8. Desarrollo de la vista. Estructura de la aplicación

Para la aplicación `Evaltics` se ha diseñado las pantallas mediante el uso de las librerías `SmartGWT` que expanden y mejoran los controles por defecto de GWT.

`SmartGWT es` un conjunto de librerías que aporta a `GWT` una gran cantidad de *widgets* con funcionalidad extra y añade en muchos de ellos integración con el servidor, siendo no sólo widgets de lectura de datos, sino capaces de cualquier tipo de manipulación de los mismos de una manera rápida y sencilla. `SmartGWT` se distribuye bajo licencia GPL y está basado en `SmartClient,` que es un conjunto de librerías gráficas de tratamiento de datos cliente/servidor para el desarrollo de páginas web en JavaScript.

Para su utilización debemos configurar el modulo *Evaltics.gwt.xml.* Antes de configurarlo debemos descargar el paquete de `SmartGWT` de la web del proyecto:

http://code.google.com/p/smartgwt/downloads/list

Descargamos la versión 2.4 en un directorio y en el proyecto `Eclipse` pulsamos sobre el fichero *Evaltics.gwt.xml.* En el menú contextual seleccionamos la opción "*Configure for using SmartGWT*".

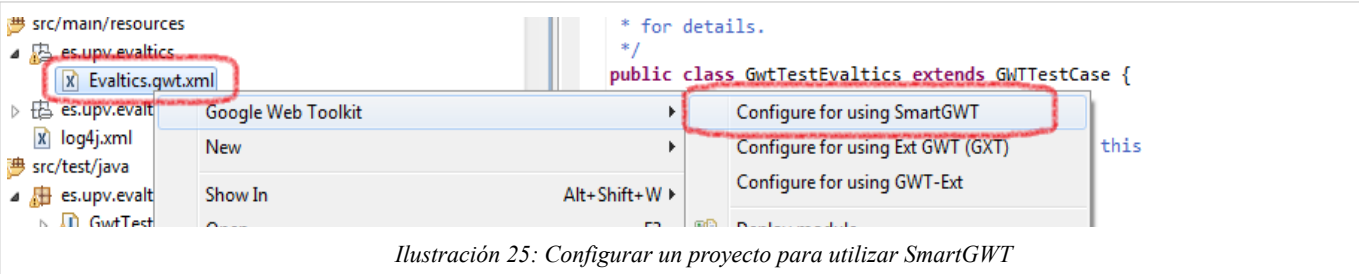


Ilustración 25: Configurar un proyecto para utilizar SmartGWT

Una vez pulsada la opción nos pedirá el directorio donde hemos descomprimido el paquete descargado y aceptaremos. Esto modifica el fichero *Evaltics.gwt.xml* añadiendo la dependencia

```
    <inherits name="com.smartgwt.SmartGwt"/>
```

Para crear un interfaz de usuario podemos usar el editor integrado. Para editar la pantalla de *PanelAlumno* pulsamos sobre el fichero `PanelAlumnosViewImpl` y en el menú seleccionamos *Open With → GWT Designer.* El editor tiene una pestaña para acceder al editor de interfaz

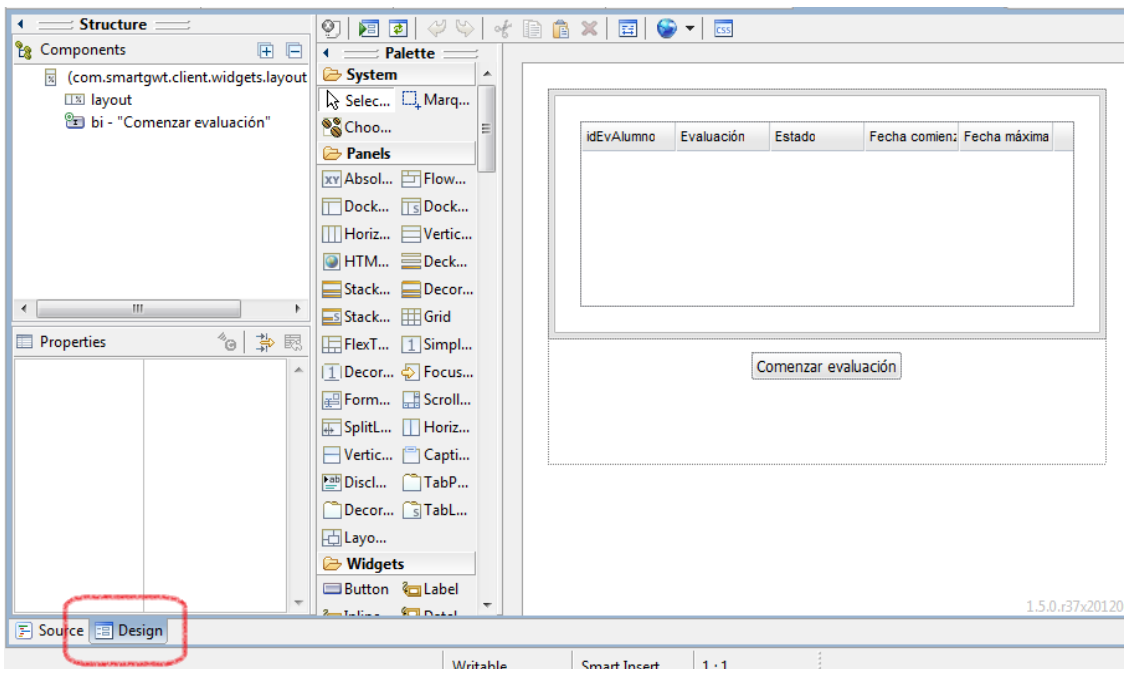


Ilustración 26: Creación de interfaces gráficas con GWT Designer

La generación del interfaz consiste en la colocación de controles y de la asignación de acciones a los eventos provocados por la interacción con estos controles.

Los interfaces dinámicos e intuitivos de `GWT` hacen que manejar la aplicación sea muy sencillo. Los formularios cuentan con validaciones para evitar que se introduzca información inválida y avisa directamente al usuario. La comunicación con el servidor se realiza de forma instantánea sin necesidad de recargar la página y la compatibilidad con los diversos navegadores está garantizada por el uso de librerías de alto nivel.

Casos de uso

La aplicación `Evaltics` tiene dos tipos de usuarios diferenciados:

- Profesores: gestionan la información académica y la información de preguntas y también publican evaluaciones.
- Alumnos: acceden a la aplicación para realizar una evaluación de alguna asignatura en la que estén matriculados.

Los descripción casos de uso para cada uno de los tipos de usuarios son los siguientes:

Profesores

- Información Académica
    - Gestión de centros
        - Ver los centros dados de alta
        - Dar de alta nuevos centros
        - Eliminar centro
        - Modificar centros

- Gestión de asignaturas
  - Dar de alta nuevas asignaturas. Las asignaturas tienen asociado un centro y un curso.
  - Eliminar y modificar las asignaturas existentes

- Gestión de alumnos y matrículas
  - Dar de alta nuevos alumnos
  - Gestionar los alumnos existentes.
    - Modificar el usuario y clave con la que se autentican los alumnos.
  - Matricular a alumnos en asignaturas

- Gestión de autores
  - Dar de alta nuevos autores
  - Gestionar los autores existentes

- Gestión de evaluaciones

- Gestión de preguntas
  - Gestionar los temas de una asignatura
  - Dar de alta nuevas preguntas asociadas a una asignatura y a un tema.
  - Modificar las preguntas existentes
  - Definir el tipo de una pregunta
  - Definir las respuestas asociadas a una pregunta de tipo “Opción múltiple”
  - Definir la respuesta correcta de una pregunta de tipo “Verdadero/Falso”
  - Definir la respuesta posible para una pregunta de tipo “Redacción”

- Evaluaciones
  - Crear evaluaciones. Para crear una evaluación se debe seleccionar preguntas de un tema de una asignatura.
  - Las evaluaciones publicadas son presentadas a los alumnos matriculados en la asignatura

- Corrección de evaluaciones
  - Permite ver las evaluaciones realizadas de una asignatura y corregir las respuestas dadas por los alumnos.

Alumnos

- Realización de evaluaciones. Un alumno tiene disponibles para realizar las evaluaciones publicadas y no pasadas de las asignaturas en las que se ha matriculado.
  - Para la realización de evaluaciones se debe contestar el total de preguntas definidas en la evaluaciones creadas por el profesor.
  - Cada tipo de pregunta se muestra según se haya definido el tipo

Estructura de la aplicación

Para el proyecto `Evaltics`, se ha creado una estructura con una pantalla inicial, desde donde se puede acceder a las distintas opciones de edición. La pantalla inicial contiene enlaces a las secciones:

Ilustración 27: Pantalla inicial de Evaltics

- Información académica : permite la gestión de centros, asignaturas, alumnos y sus matriculaciones
- Evaluaciones : permite la gestión de preguntas y la publicación de evaluaciones asociadas a una asignaturas
- Panel de alumnos: es el lugar donde los alumnos realizan las evaluaciones que se han sido publicadas y que tienen disponibles.

Información Académica

Dentro de , encontramos diversas pestañas para acceder a la gestión de los centros, de las asignaturas y de los alumnos y de añadir o quitar matrículas

La gestión de centros y asignaturas es similar. Muestran un listado de las distintas entidades dadas de alta en una tabla. Al seleccionar una fila permite su modificación en el formulario inferior. También permiten la eliminación de registros o la creación de nuevos mediante los botones correspondientes.

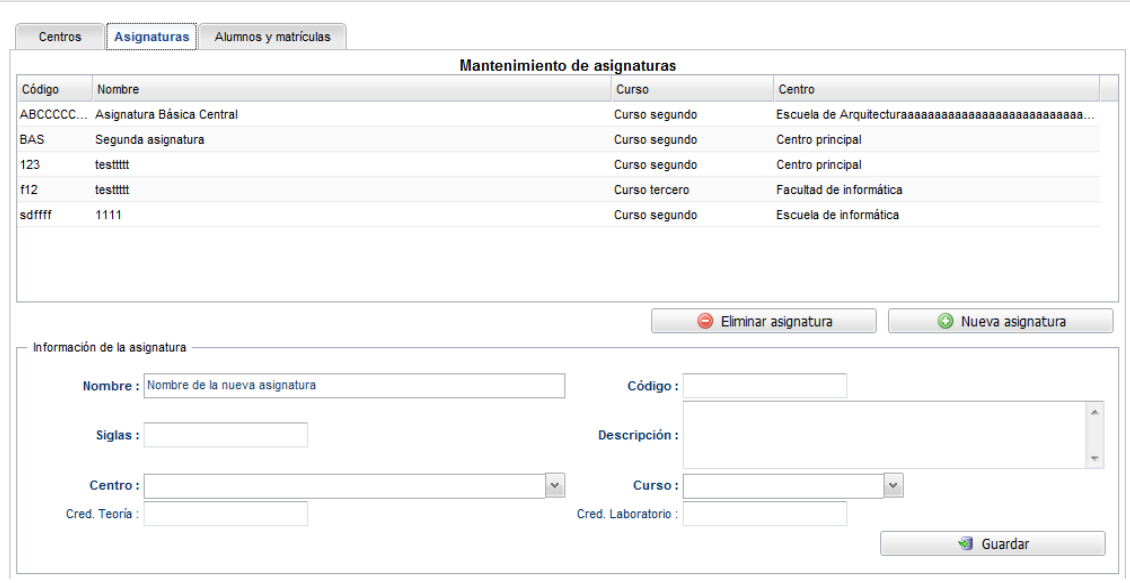


Ilustración 28: Mantenimiento de asignaturas

La gestión de alumnos tiene una apariencia similar pero permite la matriculación de asignaturas para el alumno seleccionado. Para realizar una matrícula, seleccionaremos un alumno y buscaremos la asignatura a matricular, posiblemente mediante el filtrao de curso y centro. Una vez seleccionada la asignatura pulsaremos sobre el botón "Matricular asignatura" y aparecerá en la lista de asignaturas del alumno.

Para desmatricular, pulsaremos sobre el icono de prohibido que aparece al pasar el cursor sobre una fila.

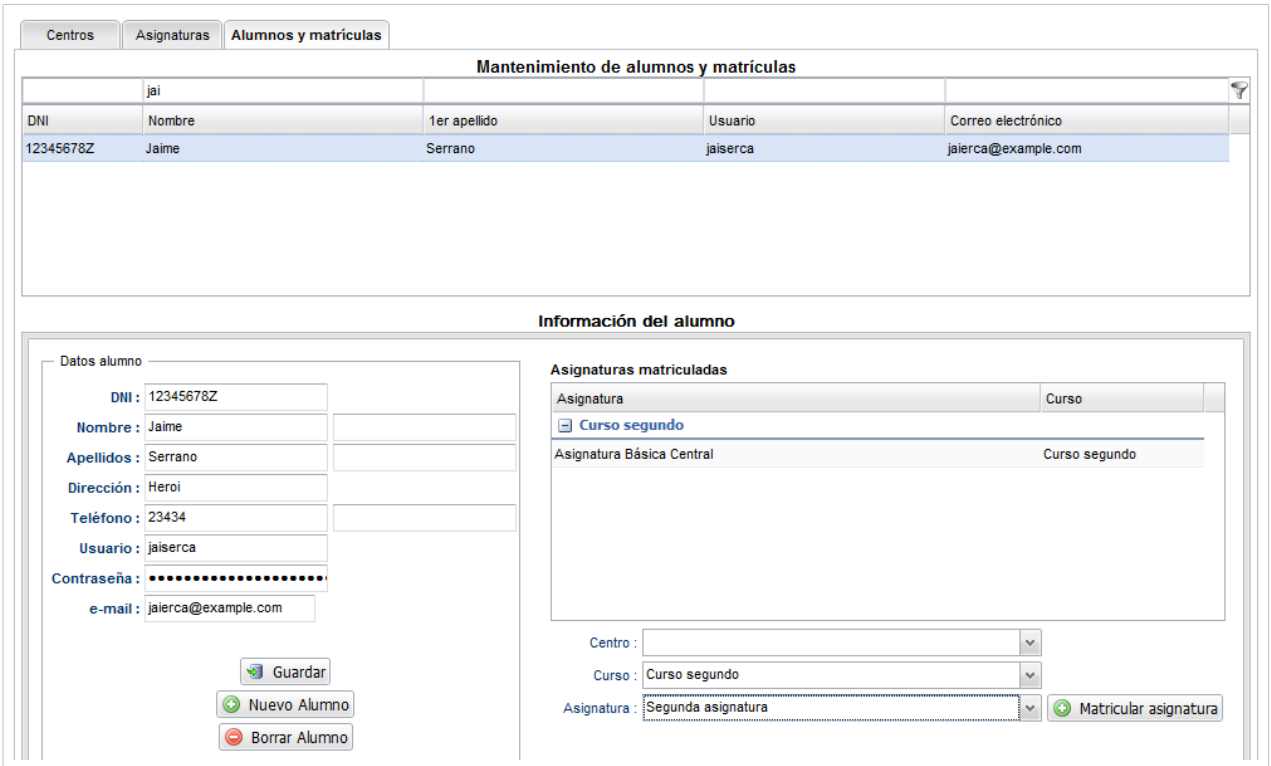


Ilustración 29: Gestión de alumnos y matrículas

Gestión de evaluaciones

En la sección de gestión de evaluaciones tenemos pestañas para :

- Gestión de preguntas: Al seleccionar una asignatura y un tema, se muestran las preguntas asociadas a ese tema, pudiéndose modificar su enunciado así como el tipo de pregunta. Podemos agregar nuevos tipos de preguntas mediante la opción situada al lado del selector de tipos.

  La pestaña Respuestas permite especificar la respuesta correcta a la pregunta, según el tipo de pregunta que se haya establecido:

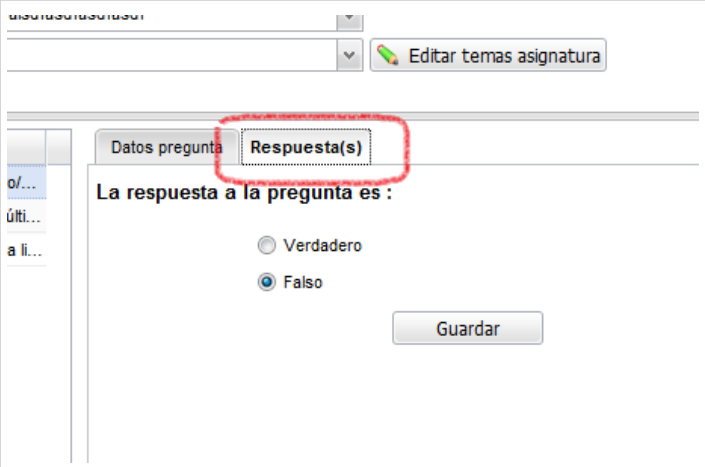


Ilustración 30: Gestión de respuesta de una pregunta de tipo V/F

  Para las preguntas de tipo *Verdadero/Falso* permite decir cual es la correcta, para las preguntas de tipo *opción múltiple,* permite definir varias respuestas y si son correctas o no y para las de *redacción libre,* permite describir una posible respuesta válida.

- La pestaña de evaluaciones permite publicar nuevas evaluaciones para que sean respondidas por los alumnos. Seleccionaremos la asignatura deseada y un tema de la asignatura.

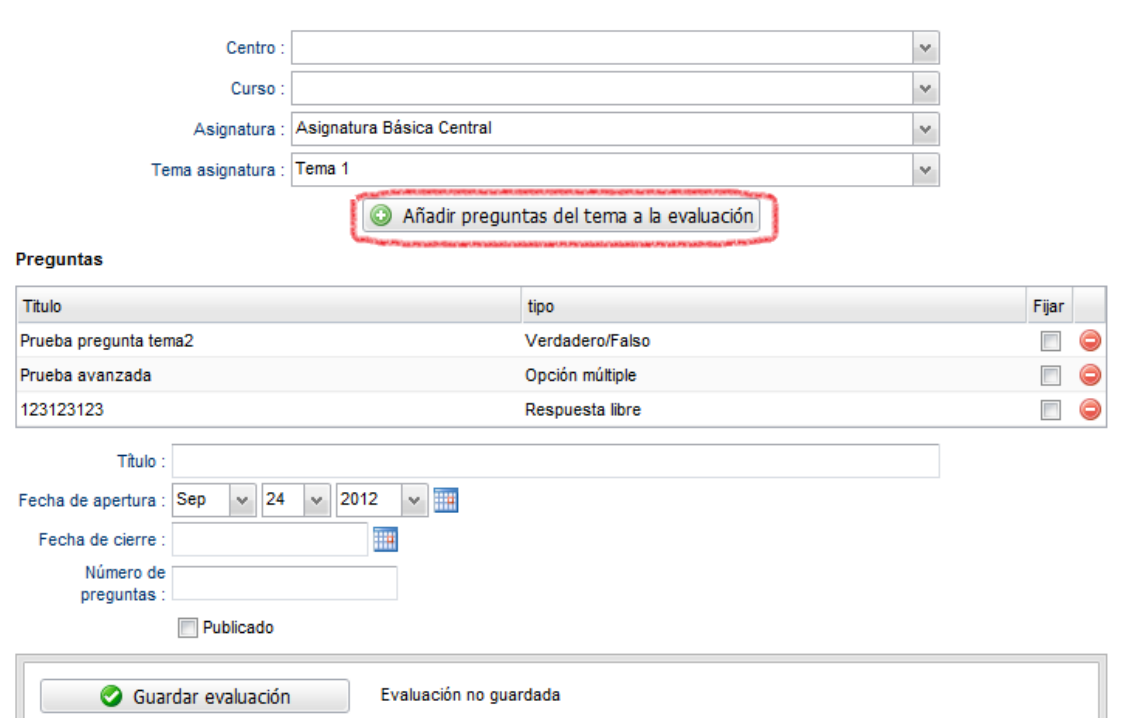


Ilustración 31: Creación de una nueva evaluación

Al pulsar sobre "*Añadir preguntas*" aparecen en el listado las distintas preguntas definidas para ese tema en la gestión de preguntas. Quitaremos las preguntas que no deben aparecer en la evaluación y tras introducir varios datos, como si debe ser publicado o la fecha máxima de realización, procederemos a guardar la evaluación. Todos los alumnos matriculados en esa asignatura tendrán disponible la evaluación a partir de la fecha marcada si la marca de publicado está seleccionada.

- La pestaña de corrección de evaluaciones permite a los profesores la validación de las evaluaciones realizadas por los alumnos. Para cada evaluación mostrada permite marcar las preguntas correctas o incorrectas y ver la puntuación final del alumno.

- La pestaña de *Tipos de pregunta,* permite la misma gestión que la opción situada al lado de la selección de tipo de pregunta en la edición de una pregunta y es un ejemplo de reutilización de componentes gráficos.

Panel de alumnos

El panel de alumnos es donde los alumnos acceden para realizar evaluaciones publicadas de las asignaturas en las que están matriculados.

Al entrar, solicita una validación de credenciales solicitando un usuario y una clave

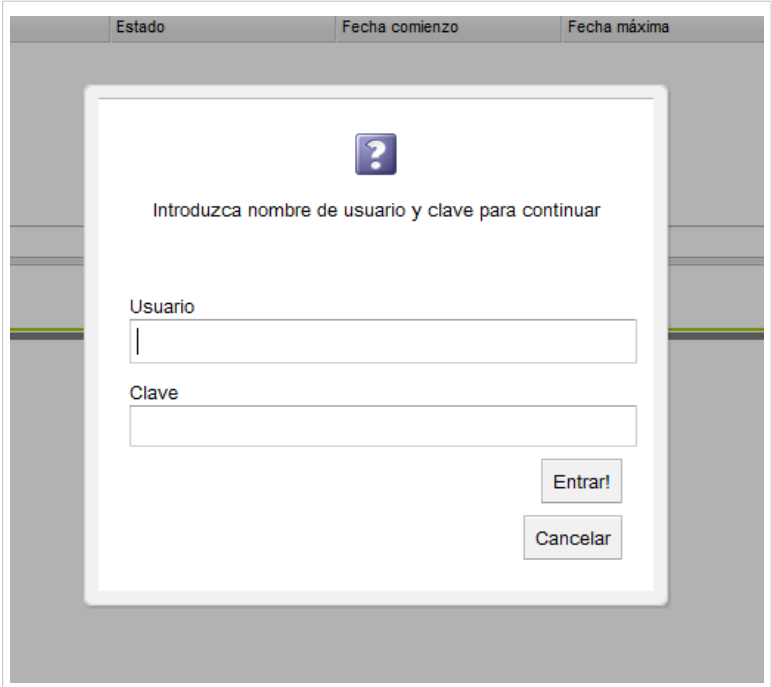


Ilustración 32: Pantalla para la validación de credenciales de un alumno

El diálogo no permite avanzar si no se introducen datos válidos. Una vez validado el acceso se muestran las evaluaciones disponibles que son las evaluaciones publicadas dentro de la fecha entre la apertura y la máxima de realización, junto con el estado *COMENZADO* / *FINALIZADO* / *NO COMENZADO* :

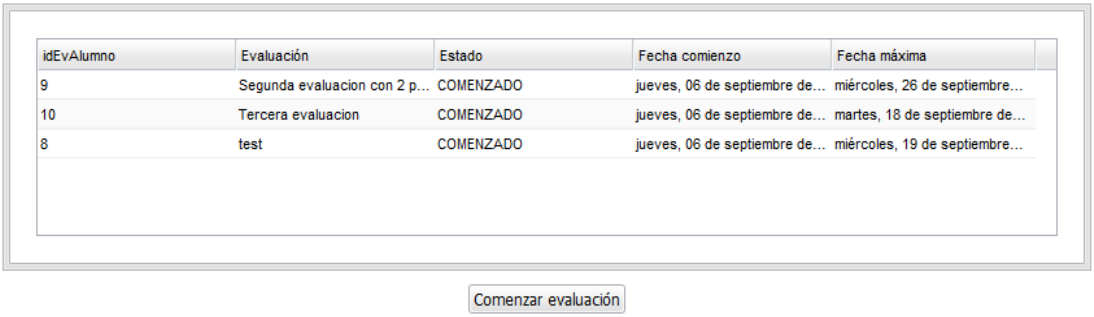


Ilustración 33: Evaluaciones disponibles para un alumno

Una vez seleccionada una evaluación, el alumno debe pulsar sobre el botón para comenzar o continuar una evaluación pendiente. Se muestra entonces el panel de respuestas donde el alumno marcará la opción correcta o escribirá la respuesta en el cuadro correspondiente, dependiendo del tipo de pregunta. Una vez respondido, debe guardar la respuesta mediante el botón inferior y puede continuar a la siguiente pregunta o revisar las anteriores mediante los botones situados a los lados. El alumno puede detener una evaluación en cualquier momento cerrando el diálogo mostrado.

La aplicación muestra un aviso cuando está en la última pregunta o en la primera.

Ilustración 34: Realizando una evaluación

9. Implementando persistencia en base de datos

En la aplicación `Evaltics`, la persistencia de datos se ha desarrollado mediante el `ORM Hibernate`.

`Hibernate` permite el mapeo de objetos `Java` con entidades de una base de datos, simplificando hasta eliminar la necesidad de creación de consultas a la base de datos. La aplicación se ha desarrollado sobre la base de datos `MySQL`, pero `Hibernate` permite la abstracción del servidor de base de datos utilizado mostrando una API general con una propia sintaxis de consulta sin relación directa con la base de datos subyacente.

A continuación se muestra el esquema de base de datos como referencia.

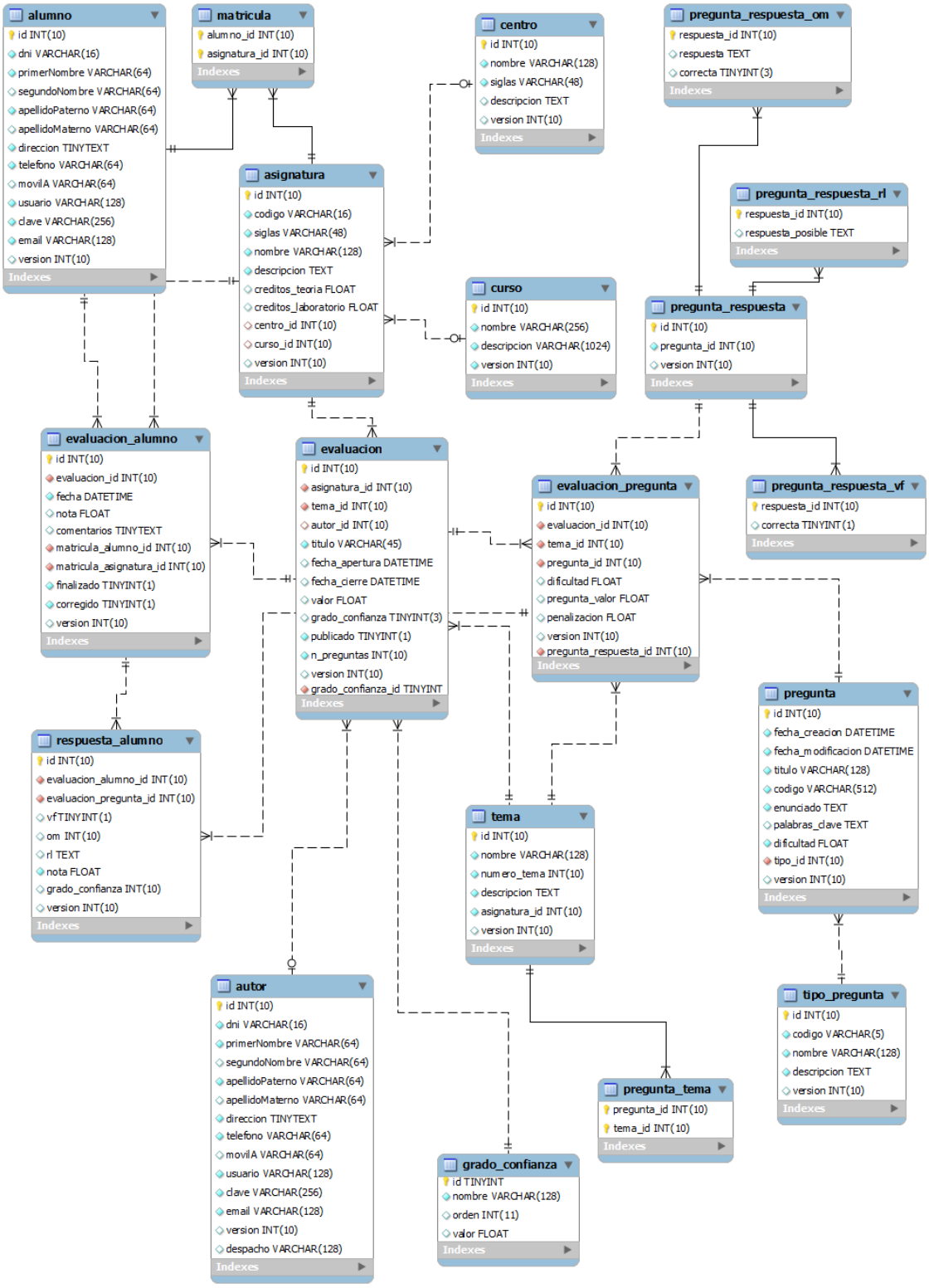


Ilustración 35: Modelo de base de datos para la aplicación Evaltics

Para la definición de las respuestas se ha utilizado las capacidades de herencia que permiten diversos `ORM`, entre ellos `Hibernate`, para adaptar el polimorfismo de las clases `Java` a la base de datos. Esto permite un uso polimórfico de llamadas a procedimientos simplificando la definición y la cantidad de métodos necesarios.

La configuración de `Hibernate` se realiza por medio de anotaciones en las clases del modelo que forman las entidades. Para cada una de las clases que deben ser mapeadas a la base de datos se debe marcar con la anotación `@Entity`.

```
@Entity
@Table(name = "centro")
@XmlRootElement(name = "Centro", namespace = "http://evaltics.upv.es")
public class Centro implements java.io.Serializable {
   private static final long serialVersionUID = -5006916269856510592L;

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   @Column(name = "id")
   private Long id;

   @Column(name = "nombre", nullable = false, length = 128)
   private String nombre;

   @Column(name = "siglas", nullable = false, length = 48)
   private String siglas;

   @Column(name = "descripcion")
   private String descripcion;

   @Version
   private Integer version;
```

Listado 23 : Anotaciones en Hibernate

La tabla de base de datos asociada a la entidad se define con la anotación `@Table.`

Para la utilización de `Hibernate` es recomendable la utilización de claves sintéticas, en lugar de claves naturales para la identificación, es decir, como clave primaria. Las claves sintéticas no tienen significado en la lógica del negocio, pero facilitan el proceso de edición de un entidad, puesto que nos aseguramos que va a ser un identificador que no va a cambiar durante todo su existencia. Las claves naturales deben identificarse, sin embargo, y ser utilizadas para la definición de los métodos `hashCode` y `equals` de las clases `Java. Hibernate` utiliza estas funciones para comparar entidades del mismo tipo.

La clave primaria se especifica con la anotación `@Id` y está marcada como que se genere automáticamente al guardar. Esto evita el tener que llevar un control de los identificadores, facilitando la inserción de registros.

Cada columna de la tabla asociada se mapea con una propiedad de la clase `Java`, de tal forma que al obtener las entidades de un tipo, `Hibernate` automáticamente generará una consulta SQL adecuada la recuperar lo datos. Igualmente generará consultas para el borrado o la actualización de entidades.

La anotación @`Version` se utiliza para la gestión de concurrencia para evitar que se guarden datos que han sido modificados por otro usuario en aplicaciones con acceso concurrente. Cuando `Hibernate` va a guardar una entidad, compara el campo `Version` que ha obtenido al leer la entidad de la base de datos con el existente. Si no coinciden no permite que se guarden los

cambios, puesto que significa que otro usuario lo ha modificado desde el momento en lo que se leyó. La aplicación debe decidir que medidas tomar, si guardar los datos de cualquier manera o avisar al usuario para que los revise.

Hibernate también permite la asociación de entidades. Por ejemplo, asignatura contiene una asociación con la entidad Curso. Esto se define con las anotaciones `@OneToMany`, `@ManyToOne`, `@OneToOne` o `@ManyToMany`:

```
@ManyToOne(fetch = FetchType.EAGER)
@Cascade (value = { org.hibernate.annotations.CascadeType.SAVE_UPDATE})
@JoinColumn(name = "curso_id", insertable = false, updatable = false)
private Curso curso;

@Column(name="curso_id")
private Long cursoId;
```

Listado 24 : Anotaciones para asociaciones en Hibernate

`Hibernate` permite la carga de entidades de forma perezosa *(LAZY)* en donde una consulta de los datos de una asignatura no obtendría los datos de su curso asociado, sino que los datos del curso se obtendrían una vez que fueran necesario (es decir, al acceder a *asignatura::curso).* En el proyecto `Evaltics` no se ha utilizado la carga perezosa, si no que se ha utilizado la instantánea, pues que las consultas no resultan muy complejas y por el propio diseño de `GWT` cliente-servidor, no es posible acceder a los datos de un curso desde el cliente de la aplicación.

Configuración de `Hibernate`

Una vez anotados las entidades que forman parte del modelo, es necesario configurar el proyecto para utilizar `Hibernate.` Las dependencias necesarias que hay que añadir al *pom.xml* si no están ya añadidas son:

```
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-core</artifactId>
	<version>${org.hibernate.version}</version>
</dependency>
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-annotations</artifactId>
	<version>${org.hibernate.version}</version>
</dependency>
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-entitymanager</artifactId>
	<version>${org.hibernate.version}</version>
</dependency>
<dependency>
	<groupId>org.hibernate.javax.persistence</groupId>
	<artifactId>hibernate-jpa-2.0-api</artifactId>
	<version>1.0.0.Final</version>
</dependency>
```

Listado 25 : Dependencias para Hibernate

Donde la variable `org.hibernate.version` hace referencia a la definida en el elemento `properties`:

```
    <org.hibernate.version>3.5.1-Final</org.hibernate.version>
```

Dentro del directorio src/main/*resources* crearemos los siguientes ficheros de configuración de Spring:

applicationContext-dao.xml

Este fichero crea el gestor de transacciones necesario para el control de sentencias en grupos y el *sessionFactory* que se encarga de mantener una sesión a la conexión de base de datos subyacente. Los *beans* creados serán luego inyectados en los *DAOs* de la aplicación.

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:context="http://www.springframework.org/schema/context"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd"
           default-lazy-init="true">

       <!-- Transaction manager for a single Hibernate SessionFactory (alternative to
JTA) -->
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
            <property name="sessionFactory" ref="sessionFactory"/>
        </bean>

        <!-- Hibernate SessionFactory      -->
        <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
            <property name="dataSource" ref="dataSource"/>
            <property name="configLocation" value="classpath:hibernate.cfg.xml"/>
            <property name="hibernateProperties">
                <value>
                    hibernate.dialect=${hibernate.dialect}
                    hibernate.query.substitutions=true 'Y', false 'N'
                    hibernate.cache.use_second_level_cache=true
                    hibernate.cache.provider_class =
org.hibernate.cache.EhCacheProvider
                    hibernate.show_sql=false
                    hibernate.format_sql=true
                </value>
                <!-- Turn batching off for better error messages under PostgreSQL -->
                <!-- hibernate.jdbc.batch_size=0 -->
            </property>
        </bean>


    </beans>
```

Listado 26 : applicationContext-dao.xml

La referencia a `${hibernate.dialect}` es una sustitución que se realizará con el fichero properties definido en el siguiente archivo de configuración.

applicationContext-resources.xml

Este fichero crea un *datasource* que es la configuración de la conexión a la base de datos que utilizará `Hibernate`.

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans                          xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:jee="http://www.springframework.org/schema/jee"
                       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                                http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

        <!-- For mail settings and future properties files -->
                                                    <bean           id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
            <property name="locations">
                <list>
                    <value>classpath:jdbc.properties</value>
                </list>
            </property>
        </bean>


       <!-- datasource con c3p0 provider -->

            <bean  id="dataSource"  class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">
            <property name="driverClass" value="${jdbc.driverClassName}"/>
            <property name="jdbcUrl" value="${jdbc.url}"/>
            <property name="user" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
            <property name="autoCommitOnClose" value="true"/>
        </bean>

    </beans>
```

Listado 27 : applicationResources.xml

No están especificados los datos en este fichero, si no en el fichero jdbc.properties. Cada variable hace referencia a una entrada de ese fichero y Spring realiza una sustitución al crear el bean *dataSource* con los valores del fichero properties

El contenido del mismo es:

```
    jdbc.driverClassName=com.mysql.jdbc.Driver
    jdbc.url=jdbc:mysql://localhost/evaltics?
createDatabaseIfNotExist=true&amp;useUnicode=true&amp;characterEncoding=utf-8
    jdbc.username=evaltics
    jdbc.password=evaltics

    hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
```

Listado 28 : Propiedades de la conexión a base de datos

Que especifica una base de datos `MySQL` en la máquina `localhost` y con usuario y clave `evaltics`.

Finalmente crearemos el fichero ***hibernate.cfg.xml*** al que se hace referencia en el *bean sessionFactory.* Este fichero contiene las entidades que se deben mapear a tablas en la base de datos, así como varias opciones de configuración de bajo nivel.

```
```

```
    <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

    <hibernate-configuration>
        <session-factory>


              <property name="connection.autoReconnect"> true</property>
              <property                    name="connection.is-connection-validation-
required">true</property>
              <property name="hibernate.c3p0.min_size">5</property>
              <property name="hibernate.c3p0.max_size">20</property>
              <property name="hibernate.c3p0.timeout">300</property>
              <property name="hibernate.c3p0.max_statements">50</property>
              <property name="hibernate.c3p0.idle_test_period">3000</property>
              <property name="hibernate.connection.provider_class">
                 org.hibernate.connection.C3P0ConnectionProvider</property>

            <mapping class="es.upv.evaltics.shared.model.Centro"/>
            <mapping class="es.upv.evaltics.shared.model.Curso"/>
            <mapping class="es.upv.evaltics.shared.model.Asignatura"/>
    <!--          …. más entidades …  -->

        </session-factory>
    </hibernate-configuration>
```

Listado 29 : hibernate.cfg.xml

El fichero especifica que se mapee la entidad *Centro, Curso* y *Asignaturas* a sus correspondientes tablas en la base de datos utilizando la configuración de las anotaciones en el código `Java`.

También es necesario modificar el fichero *web.xml* para añadir los nuevos ficheros de configuración de `Spring` agregados. El elemento `contextConfigLocation` quedaría así:

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                    classpath:/applicationContext-*.xml
                    classpath*:/applicationContext.xml
                    /WEB-INF/applicationContext.xml
                    /WEB-INF/cxf-servlet.xml
              </param-value>
      </context-param>
```

Una vez creados estos ficheros, la estructura del proyecto queda como sigue:

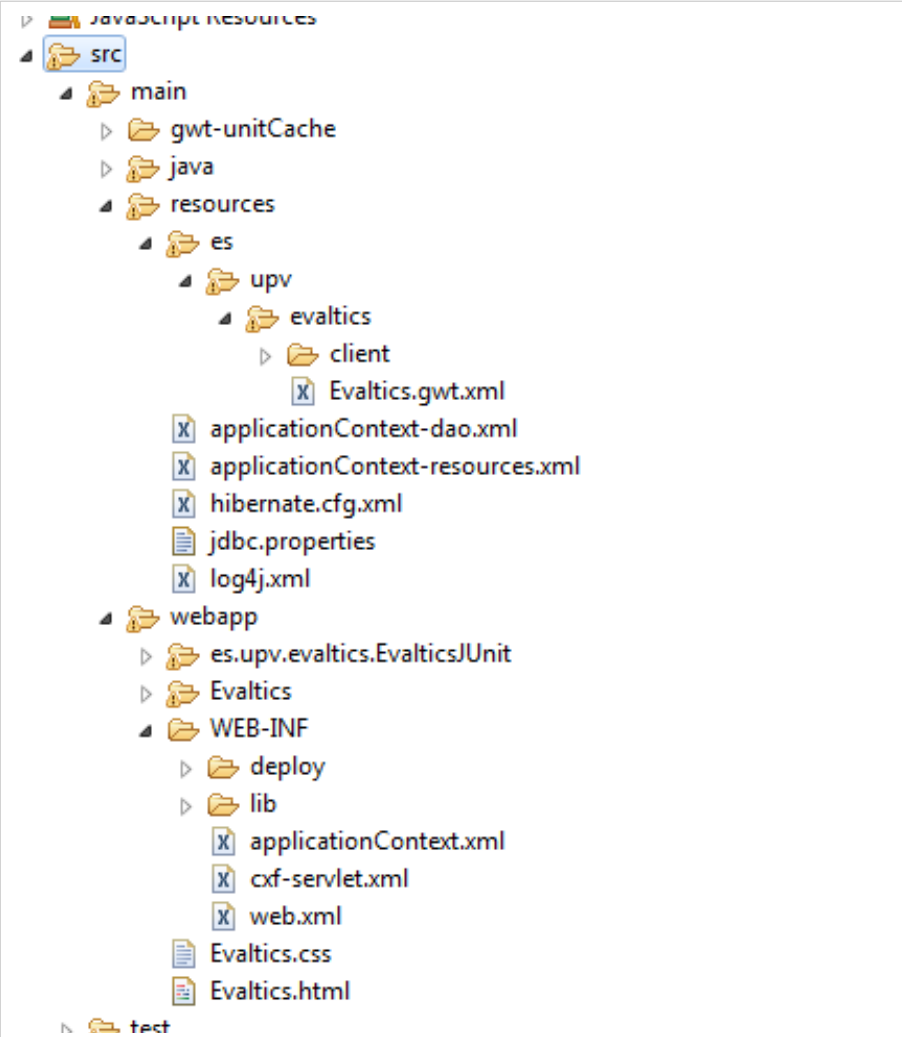


Ilustración 36: Estructura del proyecto Evaltics tras configurar Hibernate

Modificación de los DAO para el acceso a datos

Para obtener datos de `Hibernate`, los DAO deben obtener una referencia al *sessionFactory* para realizar las consultas a la base de datos por medio de `Hibernate`. Como el *sessionFactory* está configurado con `Spring`, es sólo cuestión de inyectarlo en las clases DAO y realizar las consultas.

Modificaremos el fichero *CentroDao.java* creado anteriormente para definir los métodos que se van a poder ejecutar para leer, eliminar o modificar registros de centro.

```
interface CentroDao {

  public List<Centro> getAll();

   Centro get(Long id);

   boolean exists(Long idCurso);

   Centro save(Centro object);

   void remove(Long id);
```

```
}
```

Listado 30 : Interfaz de un DAO

Estas son las funciones que el servicio podrá utilizar para el acceso a los datos de la base de datos. En la implementación del DAO, realizamos las llamadas a `Hibernate`:

```
package es.upv.evaltics.server.dao.hibernate;

import java.util.ArrayList;
import java.util.List;

import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Required;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;

import es.upv.evaltics.server.dao.CentroDao;
import es.upv.evaltics.shared.model.Centro;

@Repository("centroDao")
public class CentroDaoHibernate implements CentroDao {

    private Class<Centro> persistentClass;
    private HibernateTemplate hibernateTemplate;
    private SessionFactory sessionFactory;

    public CentroDaoHibernate() {
        this.persistentClass = Centro.class;
    }

    public HibernateTemplate getHibernateTemplate() {
        return this.hibernateTemplate;
    }

    public SessionFactory getSessionFactory() {
        return this.sessionFactory;
    }

    @Autowired
    @Required
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }


   @Override
   public List<Centro> getAll() {
          return hibernateTemplate.loadAll(this.persistentClass);
   }

   @Override
   public Centro get(Long id) {
          Centro  entity  =  (Centro)  hibernateTemplate.get(this.persistentClass,
id);

          return entity;
   }
```

```
	@Override
	public Centro save(Centro object) {
		return (Centro) hibernateTemplate.merge(object);
	}

	@Override
	public void remove(Long id) {
		hibernateTemplate.delete(this.get(id));
	}

}
```

Listado 31 : Implementacion de un DAO en Hibernate

Como vemos, al *DAO* se le inyecta el *bean* session*F*actory que crea un template de `Hibernate` con la conexión definida en el *sessionFactory.* La *API* de `Hibernate` es muy sencilla y con llamadas a `hibernateTemplate.merge` o `hibernateTemplate.delete` se realizan las operaciones que son traducidas a `SQL.`

Puesto que a la clase *centroManager* se le había inyectado el *bean centroDao,* al acceder al servicio de centros, se acaba por llamar al DAO y se realizan las consultas en base de datos por medio de `Hibernate.`

La estructura del proyecto en la parte servidora de implementación del servidor queda como:

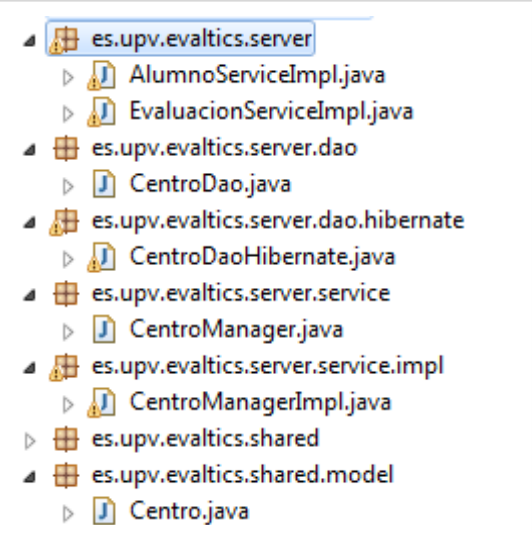


Ilustración 37: Estructura del proyecto tras introducir los DAO

10. Integrando la aplicación web con la capa de persistencia

Hasta ahora hemos visto como se realiza la implementación de los servicios que forman la parte servidora y de acceso a datos de la aplicación por una parte, y por otra parte como desarrollar interfaces gráficos que aporten experiencia visual a la aplicación. En esta sección veremos como se integran las dos partes por medio de herramientas que la plataforma `GWT` permite para la comunicación.

La manera de integrar las dos partes es distintas según se realicen llamadas a procedimientos remotos por medio de RPC o se realicen llamadas a los servicios `REST`.

Integración por medio de llamadas RPC

Las llamadas a procedimientos `RPC` desde el cliente `GWT` son más sencillos porque son el métodos estándar de uso, pero tiene el problema de que las entidades que se devuelven no son directamente utilizables en código `Javascript`, sino que hay que transformar sus propiedades en objetos `Javascript` si fuera necesario.

Desde la parte de cliente se debe obtener una referencia al servicio `RPC` y realizar una llamada asíncrona. Modificaremos la vista de evaluaciones `PanelEvaluacionesViewImpl.java` para que llame al método `finalizarEvaluación` cuando se pulse un botón.

NOTA: esto es para simplificar, con el modelo *MVP,* la vista llamaría a un método `finalizarEvaluacion` del presentador `PanelEvaluacionesActivity` para que fuera la actividad quien ejecutara la llamada al procedimiento remoto.

Modificamos la vista para añadir un manejador del evento *click* sobre un botón :

```
    button.addClickHandler(new ClickHandler() {
                    public void onClick(ClickEvent event) {

                            EvaluacionServiceAsync        evaluacionService        =
EvaluacionService.Util.getInstance();

                            evaluacionService.finalizarEvaluacion(new
AsyncCallback<Void>() {

                                    @Override
                                    public void onSuccess(Void result) {
                                            Window.alert("Finalización correcta");

                                    }

                                    @Override
                                    public void onFailure(Throwable caught) {
                                            Window.alert("No se ha podido finalizar");

                                    }
                            });
                    }
            });
```

Listado 32 : Llamada RPC desde el cliente

El código obtiene una referencia al servicio *evaluacionService* y realiza una llamada al método `evaluacionService.finalizarEvaluacion`. En este caso no acepta parámetros. La respuesta viene dada en la llamada asíncrona a un *callback* de tipo `AsyncCallback`. En este caso, si la llamada ha ido correctamente mostrara un aviso de éxito, puesto que ejecutará la función `onSuccess` del *callback,* o en cambio mostrará que ha habido algún fallo al realizar la finalización de la evaluación, ejecutando el código de `onFailure`.

En caso de recibir datos, o una lista de objetos, el cliente debería manejarlos y mostrarlos al usuario o tratarlos de la manera que sea conveniente.

Integración de los servicios REST

Los servicios `REST` no son tan directos de utilizar, sin embargo tienen la ventaja de que los objetos devueltos son directamente traducibles a código `Javascript` sin necesidad de conversión. Además los servicios `REST` son más utilizados que los servicios `GWT`, que están sólo orientados a aplicaciones `GWT`.

Para utilizar servicios `REST` en el cliente necesitamos las librerías de manejo de `JSON`. Esto es necesario configurarlo en el modulo del proyecto, el fichero *Evaltics.gwt.xml.* Para ello modificaremos el fichero para agregar las líneas

```
    <inherits name="com.google.gwt.http.HTTP" />
    <inherits name="com.google.gwt.json.JSON" />
```

Listado 33 : Modificación del módulo para añadir soporte JSON al cliente

debajo de la línea `<inherits name='com.google.gwt.user.User'/>`

Una vez realizado este cambio vamos a modificar la vista del panel alumno para obtener la lista de centros mediante la llamada a un servicio `REST`.

Modificamos `PanelAlumnoViewImpl.java` y añadimos un método `obtenerCentros`:

```
   public static void obtenerCentros() {

     Date now = new Date();
     String url = GWT.getHostPageBaseURL() + "services/api/centros/list.json?now=" +
now.getTime();

     RequestBuilder builder = new RequestBuilder(RequestBuilder.GET, url);
     builder.setHeader("Pragma", "no-cache");

     try {
       builder.sendRequest(null, new RequestCallback() {
      public void onError(Request request, Throwable exception) {

             Window.alert("Ha habido algún error " +
exception.getLocalizedMessage());

      public void onResponseReceived(Request request, Response response) {
        if (200 == response.getStatusCode()) {
          try {
                  // convertir el texto en objetos JSON
             JsArray<CentroOverlay> data =
                          CentroOverlay.arrayFromJson(response.getText());

             List<CentroOverlay> centros = new ArrayList<CentroOverlay>();
```

```
            for (int i = 0; i < data.length(); i++) {
                  centros.add(data.get(i));
            }

            Window.alert("Recibidos " + centros.size()  + " elementos" );



      } catch (JSONException e) {
            Window.alert("Ha habido algún error " );
      }
        } else {
            Window.alert("Ha habido algún error " );
        }
      }
});
} catch (RequestException e) {
            Window.alert("Ha habido algún error " );
}
}
```

Listado 34 : Llamada a un servicio REST desde el cliente

La función utiliza la *API* de `RequestBuilder` para realizar una llamada HTTP a la dirección `services/api/centros/list.json` relativa al proyecto. En este caso, es una llamada GET y la respuesta viene en un *callback* asíncrono de tipo *RequestCallback*. Este *callback* ejecutará la función `onResponseReceived` si recibe una respuesta adecuada o la función `onError` si la llamada provoca error.

Una vez recibida la respuesta, se convierte directamente a objetos `Javascript` por medio de una llamada a la función *eval* de `Javascript,` esto hace que la respuesta sea convertida a objetos que pueden tratarse desde código Java de forma transparente.

La conversión de texto `JSON` a objetos `Javascript` se realiza en una función de tipo nativo que utiliza la función nativa de los navegadores *eval* para parsear un texto en notación `JSON.` Esta función está definida de la siguiente manera:

```
public static native JsArray<CentroOverlay> arrayFromJson(String jsonString) /*-{
            return eval('(' + jsonString + ')');
   }-*/;
```

Listado 35 : Convertir un respuesta de un servicio a objetos en Javascipt

El código entre los símbolos `/*-{` y `}-*/` se ejecuta en el navegador como si código `Javascript` nativo fuera.

11. Validando aplicaciones GWT

Puesto que una aplicación GWT es una aplicación desarrollada casi en su totalidad en Java, se puede testear usando herramientas estándar y bien conocidas por los desarrolladores. Esto es una gran ventaja para la programación de aplicaciones AJAX, puesto que no existen estas herramientas de testeo o bien no son tan robustas como las existentes para el lenguaje Java.

GWT también incluye una clase especial, derivada de TestCase que es GWTTestCase que puede testear código que requiere Javascript para ejecutarse. Sin embargo el uso de estos tests son lentos y no se recomienda por la mayoría de los desarrolladores.

Puesto que GWT se utiliza para la creación de aplicaciones con interfaz gráfica y este tipo de aplicaciones siempre suponen una dificultad a la hora de testear, los casos de tests se deben centrar en la parte servidora que es más fácil de testear. Mediante el uso del patrón *MVP* para la creación de la aplicación se consigue (o intenta) mover cuanta más lógica fuera de las vistas que forman la interfaz gráfica. Esta lógica se mueve a las clases del presentador siendo más fácil testeable la lógica de la aplicación.

Distinguiremos tres tipos de pruebas en la aplicación:

1. **Pruebas unitarias**: Son pruebas dirigidas a probar clases java aisladamente y están relacionadas con el código y la responsabilidad de cada clase y sus fragmentos de código más críticos. Las pruebas unitarias se realizan sobre una clase, para probar su comportamiento de modo aislado, independientemente del resto de clases de la aplicación

2. **Pruebas de integración:** Son pruebas donde se integran los módulos o clases ya probados de forma independiente en las pruebas unitarias y se comprueba la validez de las relaciones entre cada una de las partes de la aplicación, así como la interfaz gráfica

3. **Pruebas de aceptación**: son pruebas que realiza el usuario para validar que los requerimientos funcionales se corresponden a las especificaciones

El tercer tipo de pruebas no se puede automatizar en código. Para los otros casos vamos a ver como crear casos de prueba.

Casos de prueba con JUnit

Las pruebas unitarias deben cumplir los siguientes requisitos:

- Rápidas : no debe llevar mucho tiempo realizarlas o enlenteceran el ciclo de desarrollo
- Independientes: Cada prueba debe poderse ejecutarse aisladamente del resto
- Repetible : Se pueden poder ejecutar en cualquier momento y un número de veces no determinado
- Autoconclusivas : la prueba debe conocer el resultado final a obtener
- Oportunas : se deben de realizar cuando se programa del código a testear y no en un momento posterior

Para realizar pruebas unitarias utilizaremos Junit, que es un *"framework"* para automatizar las pruebas unitarias de aplicaciones Java. Se utiliza en la fase de desarrollo, y son un conjunto de clases que permite realizar la ejecución de clases Java de manera controlada,

para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

`Junit` se integra en `Eclipse,` no es necesario descargarse ningún paquete ni instalar un nuevo plug-in para utilizarla. Eclipse facilita la creación de pruebas unitarias

Veamos un ejemplo de como realizar un test unitario para validar el funcionamiento de la función *"saveAlumno"* del servicio "*alumnoManager*". La prueba que vamos a realizar consiste en validar que la contraseña se guarda de forma cifrada al llamar a la función.

Para crear un nuevo caso de prueba en Eclipse seleccionaremos la opción *New → Other...* y elegiremos el tipo *"JUnit Test Case"*.

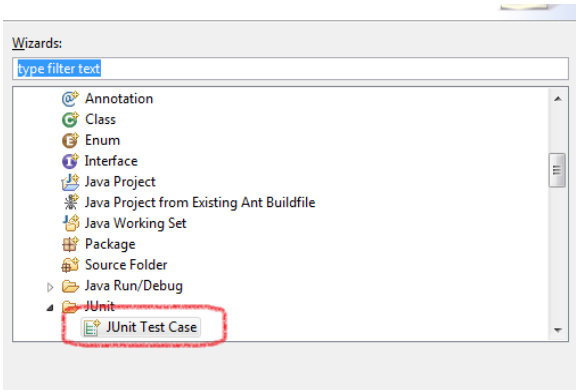


Ilustración 38: Creación de un nuevo caso de prueba

Como datos estableceremos:

Directorio fuente : `evaltics/src/test/java`
Paquete Java: `es.upv.evaltics.server.service.impl`
Nombre: `AlumnoManagerTest`
Clase padre : `junit.framework.TestCase`

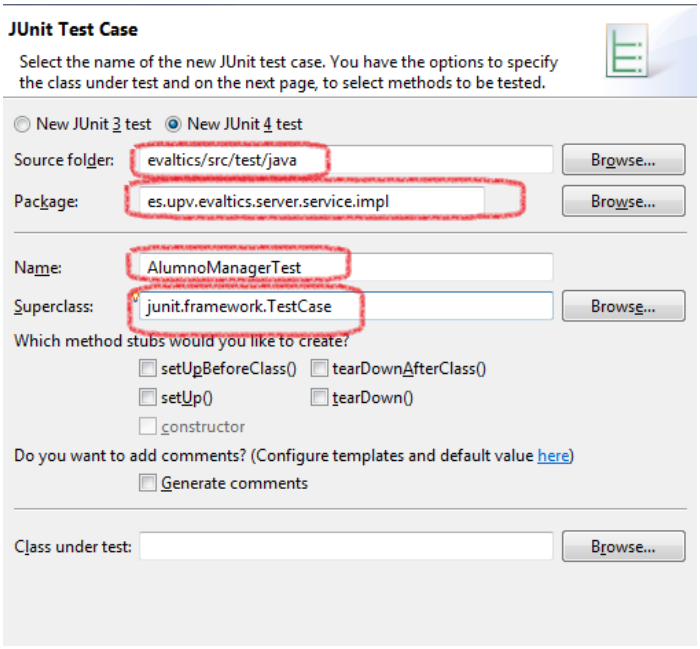


Ilustración 39: Propiedades del caso de prueba de alumnoManager

Un problema que aparece al ejecutar un caso de prueba es que normalmente las clases tienen dependencias que deben ser establecidas antes de poder realizar la prueba. El objetivo de una prueba unitaria es comprobar el funcionamiento aislado de la clase, pero muchas veces la clase no puede ser probada sin establecer esas dependencias. En el caso de la clase `AlumnoManagerImpl` a probar tenemos las siguientes dependencias:

- `AlumnoDao` : clase para acceder a la capa de persistencia de datos que obtiene los alumnos de la base de datos
- `PasswordEncoder` : Clase para cifrar una contraseña

La dependencia sobre `PasswordEncoder` puede ser solucionada fácilmente instanciando la clase. Sin embargo la clase *dao* supone el problema de que debe acceder a base de datos.

Para resolver estos problemas se han desarrollado librerías que ayudan a simular objetos que forman las dependencias de una clase que queremos probar. Esto creará objetos no reales que nos permitirán probar el funcionamiento de la clase sin problemas. Estas son librerías que se utilizan para crear objetos *"mock"*.

Una librería de este tipo es **"Mockito"** y la usaremos para simular el comportamiento del *dao* de acceso a datos.

Primeramente debemos incluir la dependencia en el proyecto sobre `mockito.` Para ello modificamos el *pom.xml* para agregar la siguiente dependencia:

```
<dependency>
	<groupId>org.mockito</groupId>
	<artifactId>mockito-core</artifactId>
	<version>1.9.0</version>
	<scope>test</scope>
</dependency>
```

Listado 36 : Dependencia para mockito para realizar tests unitarios

Ahora vamos a construir el caso de prueba. La idea general es instanciar un objeto de tipo `AlumnoManagerImpl` y ejecutar el método `saveUser`, para comprobar que el objeto devuelto por la función tiene la clave modificada.

```
public void testSaveAlumno () {
   AlumnoDao dao = mock(AlumnoDao.class);

   Alumno alumno = new Alumno ();
   alumno.setUsuario("prueba");
   alumno.setClave ("unittest");

   AlumnoManagerImpl manager = new AlumnoManagerImpl ();
   manager.passwordEncoder = new ShaPasswordEncoder();
   manager.setAlumnoDao(dao);

   when (dao.save(any(Alumno.class))).thenAnswer(new Answer<Alumno>() {
          @Override
          public Alumno answer(InvocationOnMock invocation) throws Throwable {
                        return (Alumno) invocation.getArguments()[0];
                 }
          });

   Alumno alumnoDB = manager.saveAlumno(alumno);
   assertFalse (alumnoDB .getClave(), "unittest".equals(alumnoDB .getClave()));

}
```

Listado 37 : Código del test de prueba del cifrado de la contraseña

Primeramente creamos un objeto *mock* de tipo `AlumnoDao`, este es un objeto simulado al que luego le definiremos reglas que debe satisfacer para que se pueda ejecutar el caso de prueba.

Seguidamente creamos un objeto Alumno al que le establecemos las propiedades de usuario y contraseña. La contraseña que vamos a probar es "*unittest*".

Creamos el objeto a probar que es una instancia de `AlumnoManagerImpl`, le establecemos el `passwordEncoder` y el *dao* simulado. Estos objetos, en la aplicación real, se inyectan automáticamente por `Spring` gracias a la tecnología de inversión de control, si necesidad de usar todo este código pegamento.

En el siguiente paso definimos la regla que debe de cumplir el *dao* simulado para realizar las mismas tareas que realizaría una instancia de `AlumnoDao` real. Este comportamiento es que devuelva el mismo que se pasa y que es guardado en base de datos. En un comportamiento real, para un objeto que se guarde en base de datos utilizando `Hibernate`, el objeto devuelto por la función `AlumnoDao.save` devolvería el mismo objeto, con un incremento en su valor de version si han habido modificaciones respecto al objeto en base de datos. En la prueba que estamos realizando este comportamiento no influye en nada y por eso no se ha simulado.

Hay que tener en cuenta, que aunque se esté utilizando un objeto simulado para probar el comportamiento de `AlumnoManager`, esto no invalida el test unitario, puesto que lo que realmente estamos probando es la funcionalidad de `AlumnoManager`. Las pruebas para la comprobación de la función `AlumnoDao.save` se deben de realizar en otros tests unitarios sobre ese objeto. Para el caso de las pruebas del manager nos vale con simular su comportamiento, teniendo en cuenta que el mock simule correctamente la funcionalidad.

La notación de `Mockito` permite expresar que cuando `(when)` se ejecute la función `dao.save`, entonces devuelva `(thenAnswer)` el objeto que se ha pasado como primer argumento.

Volviendo al código en el siguiente paso realizamos finalmente la llamada al método a testear:

```
    Alumno alumnoDB = manager.saveAlumno(alumno);
```

Esto, teniendo en cuenta el *mock,* nos devuelve el mismo objeto, pero por la ejecución de `saveAlumno` debemos comprobar que las contraseña que se guardado en `alumnoDB` no coincide con la inicialmente establecida *"unittest"*.

Esta comprobación la realizamos en la última línea, verificando que es falso que se cumpla que *"unittest"* y la clave guardada en `alumnoDB (alumnoDB.getClave()`) son iguales y por tanto queda comprobado.

También podemos realizar la prueba de que efectivamente, la clave almacenada en base de datos y accesible mediante `alumnoDB.getClave` es la codificación en *SHA1* de la palabra *"unittest":*

```
    assertEquals(new ShaPasswordEncoder().encodePassword("unittest", null),
alumnoDB.getClave());
```

12. Seguridad

En las aplicaciones cliente-servidor la seguridad debe centrarse sobre todo en la parte servidora puesto que es la zona que maneja los datos y debe ser capaz de limitar el acceso que se hace de los servicios que se publican por medio de usuarios y sólo permitir el acceso a usuarios autenticados.

Al estar basado en una única página web HTML, la seguridad en cliente sólo permitiría el acceso condicional a la página *Evaltics.html,* sin más capacidad de limitación.

Para el proyecto `Evaltics` se ha configurado la seguridad mediante el framework que proporciona `Spring` y se han securizado los servicios, en contraposición a la página de acceso. La seguridad en `Spring` se basa en la asignación de roles a los usuarios, mediante la validación de credenciales con una contraseña. Los roles asociados a un usuario le concederán la capacidad de utilizar los recursos a los que se les concede permiso. Por defecto un usuario anónimo tiene acceso a todos los recursos (páginas HTML, imágenes, *CSS.*.). Es posible configurar un recurso para que sólo los usuarios con un rol determinado puedan hacer uso de él, por ejemplo, el acceso a un página web sólo si el usuario autentificado mediante usuario y contraseña tiene concedido el rol correspondiente.

Las dependencias necesarias ya fueron instaladas antes. Son las siguientes:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>${org.springframework.version}</version>
 </dependency>
```

Listado 38 : Dependencias de Spring security

La configuración básica constituye un fichero *security.xml* donde se define las reglas básicas y la modificación del fichero *web.xml* añadiendo el siguiente filtro de *servlet:*

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
     <filter-name>springSecurityFilterChain</filter-name>
     <url-pattern>/*</url-pattern>
</filter-mapping>
```

Listado 39 : Definición del filtro para la seguridad en web.xml

Además declararemos el fichero *security.xml* en la relación de ficheros de configuración del contexto:

```
   <context-param>
       <param-name>contextConfigLocation</param-name>
```

```
        <param-value>
                    classpath:/applicationContext-*.xml
                    classpath*:/applicationContext.xml
                    /WEB-INF/applicationContext.xml
                    /WEB-INF/cxf-servlet.xml
                    /WEB-INF/security.xml
                </param-value>
      </context-param>
```

Listado 40 : Cambios en web.xml para la seguridad de Spring

El fichero *security.xml* se encuentra en el directorio *src/main/webapp/WEB-INF/security.xml* y en el se definen reglas para la limitación de uso de los recursos proporcionados por la aplicación.

La configuración contiene sentencias para regular el acceso a ciertas url, así como para configurar que usuarios y contraseñas se utilizarán para la validación de credenciales.

Para la aplicación `Evaltics` la configuración más importante se encuentra en la línea

```
    <global-method-security   secured-annotations="enabled"   access-decision-manager-
ref="evalticsAccessDecisionManager" />
```

que especifica que se utilizarán anotaciones a nivel de método para la regulación del acceso a los servicios. El uso de *evalticsAccessDecisionManager* se explica en la sección "*Lanzando excepciones hacia el cliente de GWT*".

La configuración en la sección `authentication-manager`

```
            <authentication-provider user-service-ref="alumnoDao">
                  <password-encoder ref="passwordEncoder" />
            </authentication-provider>
            <authentication-provider user-service-ref="autorDao">
                  <password-encoder ref="passwordEncoder" />
            </authentication-provider>
```

Listado 41 : Definición de los proveedores de autenticación

hace que `Spring Security` utilice los bean `alumnoDao` y `autorDao` para la búsqueda de los roles asociados a los usuarios. Estos *beans* deben implementar el interfaz `UserDetailsService`

```
    public    interface    AutorDao    extends    GenericDao<Autor,    Long>,
UserDetailsService
```

que es un contrato que obliga a la implementación del método

```
    UserDetails loadUserByUsername(String username)
```

que devuelve un objeto `UserDetails` a partir de un nombre de usuario. Este objeto `UserDetails` contiene la lista de roles asociados al usuario *(granted authorities)* a partir de la

contraseña que ha introducido. Si la contraseña no es válida, el usuario no tendrá ningún rol asociado.

Para configurar la seguridad de los recursos se utilizan anotaciones a nivel de método en lo servicios. Vemos por ejemplo la configuración del servicio de evaluaciones:

```
   @Secured(value={"ROLE_ALUMNO"})
      public   List<Evaluacion>   obtenerEvaluacionesPendientesAlumno   (Long
alumnoID) throws es.upv.client.security.web.EvalticsForbiddenException;
```

Listado 42 : Anotación para securizar a nivel de método

La anotación `Secured` restringe el acceso al método para obtener las evaluaciones de un alumno a que el usuario se haya autenticado y le haya sido otorgado el rol `ROLE_ALUMNO`. En la aplicación `Evaltics` este rol lo reciben los alumnos cuando se autentifican. Si se intenta acceder a este servicio desde el cliente (en el panel de alumnos, por ejemplo) sin estar autenticado, la seguridad de `Spring` impide que se ejecute el método y lanzará una excepción de tipo `AccessDeniedException`. Por defecto, esta excepción hace que la aplicación se redirija a una página de error o que muestre un diálogo de acceso solicitando un usuario y una contraseña. Para la aplicación `Evaltics` se modificado y cuando no se tiene acceso a un recurso se lanza la excepción `EvalticsForbiddenException`. Esto está establecido en la línea de *security.xml*

```
   <access-denied-handler ref="evalticsDeniedHandler" />
```

donde el *bean* está definido como

```
   <beans:bean id="evalticsDeniedHandler"
            class="es.upv.server.security.web.EvalticsAccessDeniedHandler"/>
```

El código de esta clase siempre lanza una excepción que será capturada por el cliente `GWT`. Por tanto, al intentar acceder a un servicio al que no se tiene permiso se obtendrá una excepción.

Lanzando excepciones hacia el cliente de GWT

Para poder capturar excepciones provocada en los servicios por parte del cliente de `GWT`, estas excepciones deben estar declaradas en el servicio. Hemos visto antes como el método securizado `obtenerEvaluacionesPendientesAlumno` definía la excepción `es.upv.client.security.web.EvalticsForbiddenException` que es la que se lanza por la librería `Spring Security` al acceder a un recurso limitado. Para que `GWT` sea capaz de capturar y pasar al cliente una excepción, esta debe ser declarada dentro del paquete `client` del módulo y debe implementar la interfaz `IsSerializable`. Vemos la definición de la excepción:

```
   public  class  EvalticsForbiddenException  extends  RuntimeException   implements
IsSerializable {
      /**
       *
       */
      private static final long serialVersionUID = 5216785884569752789L;

      public EvalticsForbiddenException() {
            super();
      }
```

```
    public EvalticsForbiddenException(String msg) {
            super(msg);
    }

}
```

Listado 43 : Código de la excepción propia de Evaltics para el acceso denegado

Puesto que la excepción `AccessDeniedException` que lanza `Spring Security` por defecto no implementa esa interfaz, `GWT` no es capaz de capturarla y notificarla al cliente que llama al servicio, sin embargo, la excepción `EvalticsForbiddenException` sí la captura pero debe estar definida en contrato del servicio.

Esta es la razón también por la que se declara el *bean* *evalticsAccessDecisionManager*. Este *bean* es el responsable de la acción a tomar cuando `Spring Security` se encuentra con un recurso protegido para un usuario que ya se ha autentificado. Esto es distinto de la acción a tomar cuando un usuario no está autentificado y solicita un recurso. En cualquier caso, en el proyecto `Evaltics,` tanto para un caso como para otro siempre se envía una excepción al cliente que es capaz de capturar. El *bean* está configurado tanto para los recursos que se protegen por url como para los métodos que se marcan como seguros con las anotaciones `Secured.` La definición es la misma:

```
    <beans:bean id="evalticsAccessDecisionManager"
            class="es.upv.server.security.web.EvalticsAccessDecisionManager">
        <beans:property name="decisionVoters">
            <beans:list>
            <beans:bean id="roleVoter"
class="org.springframework.security.access.vote.RoleVoter"/>
            <beans:bean id="authenticatedVoter"
class="org.springframework.security.access.vote.AuthenticatedVoter"/>
            </beans:list>
        </beans:property>
    </beans:bean>
```

Listado 44 : Definición del gestor de decisión de acceso propio

El *bean* consiste en una clase que se le establecen dos *"decisionVoters"*. Estos son estándar de `Spring Security` y se encargan de decidir si un usuario puede acceder a un recurso o no conforme a los roles que le hayan sido otorgados y los requerimientos del recurso solicitado.

En el proyecto `Evaltics,` cuando el cliente `GWT` recibe una excepción de tipo `EvalticsForbiddenException` al realizar una llamada a un servicio, muestra un cuadro de diálogo soliticando unas credenciales que serán utilizadas para realizar el acceso.

El proceso de autentificación conlleva la llamada al un `servlet` "*j_security_check*" que realiza el proceso de validación al pasarle los parámetros `j_username` y `j_password.` Esta dirección se configura en el fichero *security.xml* en la sección de control de accesos HTTP:

```
    <form-login    login-page='/login.jsp'    default-target-url="/"    login-
processing-url="/j_security_check" />
```

Roles en `Evaltics`

En la aplicación `Evaltics` sólo existen dos roles: el de alumno, `ROLE_ALUMNO` y el de autor, `ROLE_PROFESOR.` La asignación de roles se realiza en la función `loadUserByUsername` del *bean* `autorDao` que está relacionado con los usuarios autores y con el *bean* `alumnoDao` relacionado con

los usuarios alumnos. Esta función obtiene los datos de un usuario con los roles asignados en caso de ser validadas las credenciales.

Estas funciones son llamadas por `Spring Security` al estar configurados ambos *beans* como `authentication-provider` como hemos visto anteriormente. Spring `Security` primero intentará encontrar un usuario de tipo alumno con las credenciales solicitadas. En caso de resultar infructuoso utilizar el `authentication-provider` que es `autorDao` para hacer el mismo proceso.

Tanto la función en `alumnoDao` y `autorDao` devuelven un objeto de tipo `Usuario` donde se encuentra tanto el nombre de usuario como la contraseña para validar el acceso y los roles que se le asignará en caso de ser correctas. Mostramos el caso de `autorDao` :

```
    @Override
       public      UserDetails      loadUserByUsername(String      username)      throws
UsernameNotFoundException, DataAccessException {

              Map<String, Object> params = new java.util.HashMap<String, Object>();
              params.put("usuario", username);
              List<Autor>    result    =    findByNamedQuery("obtenerAutorPorUsuario",
params);

              if (result != null && !result.isEmpty()) {
                     Usuario us = new Usuario(username);
                     us.usuarioEsProfesor();
                     us.setId(result.get(0).getId());
                     us.setClave(result.get(0).getClave());
                     return us;
              } else {
                     // busacar como profesor
                     throw new UsernameNotFoundException(username);
              }
       }
```

Listado 45 : Función para obtener los roles asociados a un usuario

La función busca en la tabla autor, un registro con el usuario que se ha solicitado mediante la llamada a la `Named Query`. Si se encuentra devuelve un objeto de tipo Usuario donde se copian los datos de usuario, id, clave y los roles que se asignan mediante la llamada a `usuarioEsProfesor` que asigna los roles correspondientes:

```
    public void usuarioEsProfesor () {
              this.roles.add(new GrantedAuthorityImpl(ROLE_USER));
              this.roles.add(new GrantedAuthorityImpl(PERFIL_PROFESOR));
       }
```

Listado 46 : Asignación de roles a un autor

`ROLE_USER` corresponde al rol de un usuario autentificado en la web y `PERFIL_PROFESOR` es una constante que se resuelve en el rol `ROLE_PROFESOR`.

Para alumno es similar pero se le asigna el rol `ROLE_ALUMNO`.

Mediante la asignación de roles podemos asegurar que cada tipo de usuario de la aplicación sólo podrá acceder a los servicios que tiene concedidos.

Ofuscación de la clave en base de datos

La clave en base de datos se guarda de forma cifrada para evitar que se pueda conocer para cualquier usuario que no sea el propio. Esto se realiza al crear el usuario y darle una clave inicial y al editar el usuario.

`Spring Security` utiliza automática el algoritmo de cifrado definido en el *bean* referenciado en la definición de `authentication-provider` para la comprobación de la clave.

```
<password-encoder ref="passwordEncoder" />
```

Por tanto `Spring` Security, para comprobar si la contraseña introducida en el cuadro de diálogo es correcta, primeramente realizará el cifrado con el algoritmo definido en el *bean* `passworEncoder` y luego se comprobará con el dato recibido de la llamada a la función `loadUserByUsername`. Esta definición establece el algoritmo que implementa la clase `ShaPasswordEncoder` que es de tipo *SHA1*.

```
<beans:bean id="passwordEncoder" class=
  "org.springframework.security.authentication.encoding.ShaPasswordEncoder"
/>
```

Listado 47 : Definición del bean para cifrar la contraseña

Esta es la razón por la que la contraseña en base de datos debe utilizar el mismo algoritmo para almacenarse. Esto lo conseguimos inyectando el *bean* `passwordEncoder` en el servicio que guarda un alumno o un autor. Estos servicios con `alumnoManager` y `autorManager`. Vemos, por ejemplo, como en la implementación del servicio `alumnoManager` se inyecta el *bean* de cifrado:

```
@Service("alumnoManager")
public class AlumnoManagerImpl implements AlumnoManager {
   private static final Log log = LogFactory.getLog(AlumnoManagerImpl.class);

   AlumnoDao alumnoDao;
   GenericDao<Matricula, MatriculaPK> matriculaDao;

   @Autowired
   PasswordEncoder passwordEncoder;
```

Listado 48 : Inyección del cifrador de contraseña en los beans de alumno y autor

Y luego se utiliza en la función `saveAlumno` al guardar los datos en base de datos en la llamada al *DAO*.

```
if (cambiarClave) {
```

```
      alumno.setClave(passwordEncoder.encodePassword(alumno.getClave(),
null));

      alumnoDao.save(alumno);
```

Listado 49 : Cifrando la clave al guardar en base de datos

De esta manera también evitamos que la contraseña sea enviada al cliente en su forma original. El cliente `GWT` sólo tendrá conocimiento de la contraseña cifrada que no sirve para descifrar la original.

13. Instalación y despliegue de la aplicación

El desarrollo de un proyecto GWT con Eclipse permite ejecutar la aplicación en un modo especial de funcionamiento en el que no se ejecuta realmente código javascript en el navegador. Este modo de desarrollo incrementa la productividad en la creación de una aplicación puesto que ahorra el paso de compilación y traducción del código Java, pero a cambio obliga a la utilización de un *plugin* en el navegador. Este modo de funcionamiento es sólo para el desarrollo, no siendo aceptable para la instalación en un entorno de producción, siendo su instalación completamente independiente del entorno de desarrollo.

Para la instalación y despliegue de una aplicación GWT se puede utilizar cualquier contenedor web que soporte la especificación de *Servlets.* Para el proyecto Evaltics se ha utilizado el posiblemente más conocido en entorno no comerciales: Apache Tomcat. Este contenedor, a pesar de su licencia libre, es uno de los más utilizados y reconocidos. En cualquier caso existen numerosas alternativas, tanto libres como comerciales, como Jetty, Jboss o IBM WebSphere.

Para su instalación se debe crear un paquete *.war* que contiene toda la aplicación web. Este paquete se desplegará en el servidor Tomcat y permitirá su uso de igual forma que lo hacíamos en el entorno de desarrollo. En lo siguiente se muestra la instalación de Tomcat, de MySQL como servidor de base de datos y el proceso de creación y despliegue de la aplicación Evaltics. en un entorno de producción

Instalación de Apache Tomcat

Para la realización del proyecto se ha utilizado una versión 6.0.x de Tomcat. Esta versión es descargable desde la web http://tomcat.apache.org/download-60.cgi . Una vez descargado el paquete en su versión binaria lo descomprimiremos en un directorio, no es necesaria ninguna instalación.

Se pueden realizar varios ajusto como el puerto por donde será accesible el servidor. Esto lo realizamos modificando el fichero de configuración *server.xml* dentro del directorio conf donde se haya descomprimido el paquete. Para modificar el valor del puerto buscamos la cadena:

```
<Connector port="8087" protocol="HTTP/1.1"
```

Y la ajustamos cambiando 8087 por el puerto deseado.

La estructura de una instalación de Tomcat comprende los siguientes directorio:

- **bin**: contiene los ejecutables de inicio de la aplicación. Es posible que sea necesario ajustar las variables de entorno para especificar donde se encuentra la máquina virtual de Java. Tomcat buscará este lugar en la variable JRE_HOME o JAVA_HOME o en el PATH del sistema pero puede ser ajustado en el fichero setenv.bat o setclasspath.bat (en Windows). También es necesario ajustar correctamente la variable CATALINA_HOME al directorio donde se ha descomprimido el paquete.

- **conf**: diversos ficheros de configuración. Los más importantes son *server.xml* y *tomcat-user.xml* que permite definir los permisos para el acceso a la consola de administración de Tomcat.

- **`lib:`** librerías `Java` básicas para el funcionamiento de todas las aplicaciones como la librería *servlet-api.jar*

- **`logs:`** registra los eventos que emiten las aplicaciones que se ejecutan dentro del contenedor. Muy necesario para la resolución de problemas

- **`webapps:`** Es donde se instalan las aplicaciones web. La forma estándar es dejar un fichero *.war* en el directorio `webapps.` `Tomcat` detectará la nueva aplicación y la desplegará en un subdirectorio con el mismo nombre del paquete *.war* e iniciará su ejecución. Si el despliegue es correcto la aplicación ya estará lista para ser utilizada.

Para iniciar el servidor `Tomcat` podemos utilizar el ejecutable *startup.bat* situado en la carpeta bin. También es posible instalar un servicio con los *scripts* situados en la misma carpeta *(service.bat)*. Una vez arrancado accederemos a la dirección `http://localhost:[`*`puerto definido en server.xml`*`]` para comprobar su correcto funcionamiento.

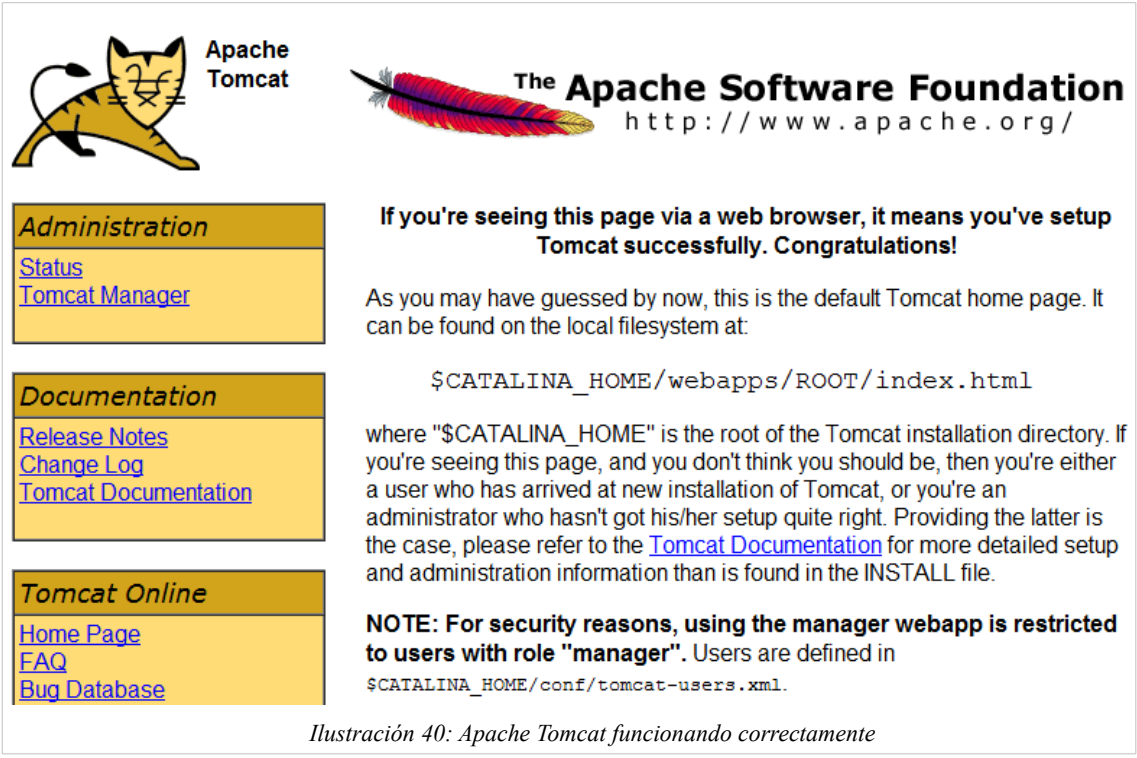


Ilustración 40: Apache Tomcat funcionando correctamente

Instalación de MySQL

`MySQL` es un servidor de base de datos con una versión libre que puede utilizarse sin ninguna restricción. La instalación no contiene ninguna dificultad y su desempeño es muy aceptable. Es muy utilizado en los entornos `J2EE.`

Para su instalación procederemos a la descarga del paquete de instalación desde la web : `http://dev.mysql.com/downloads/mysql/` en la versión de instalable para facilitar la instalación de los servicios.

Procederemos a la instalación ejecutando el paquete descargado y realizando una instalación estándar.

La instalación nos solicita la creación de los servicios para iniciar MySQL en caso de ser una instalación en Windows. También solicita una *password* de root y evitaremos utilizar la estándar en blanco por defecto.

Si la instalación ha sido correcta, podemos iniciar y parar el servidor de base de datos mediante el servicio instalado.

Accederemos a la utilidad de línea de comandos para crear un usuario básico para el acceso a la base de datos desde la aplicación Evaltics.

Desde la línea de comandos ejecutamos el cliente de Mysql que está situado en la carpeta bin del directorio de instalación de MySQL siguiendo estos pasos:

```
> mysql.exe -u root -p
```

Esto nos solicita la contraseña de root especificada en la instalación. Una vez en la línea de comandos introduciremos los siguientes comandos:

```
   mysql> create database evaltics default character set utf8;
   mysql> grant all privileges on evaltics.* to `evaltics`@`localhost` identified by
`xxxxx`;
   mysql> flush privileges;
   mysql> quit;
```

Listado 50 : Creación de esquema y usuario para Evaltics en producción

Esto crea un esquema evaltics en la base de datos y crea un usuario evaltics para acceder a todo su contenido. Este *usuario/password* y esquema de la base de datos tienen que ser los especificados en el fichero jdbc.properties del proyecto, puesto que serán las credenciales utilizadas por Hibernate para realizar la conexión.

Finalmente ejecutaremos cargaremos la base de datos inicial para el entorno de producción. Esto lo haremos con el siguiente comando del cliente de Mysql:

```
   $ mysql -u evaltics -p < script_inicial_evaltics.sql
```

Esto crea la estructura inicial de las tablas utilizadas por la base de datos y deja el esquema listo para ser utilizado por el usuario evaltics.

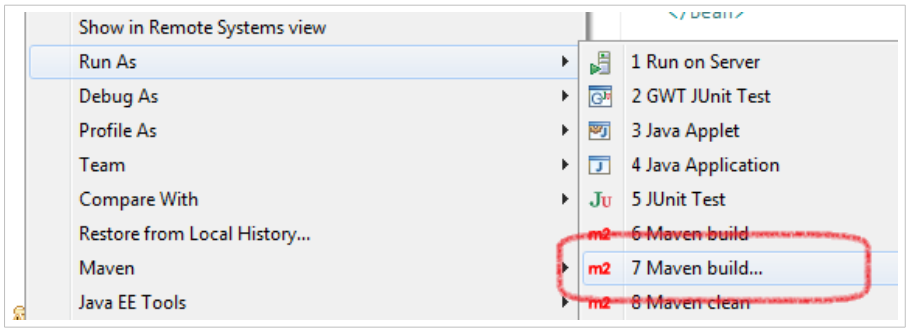
NOTA: existen muchas herramientas gráficas que facilitan estas tareas, así como oficiales de los mismos desarrolladores de MySQL (MySQL Workbench).

Creación e instalación de la aplicación

El proceso de creación e instalación es muy sencillo, consistiendo en la generación de un fichero *.war* que será copiado.

Primeramente nos aseguraremos de tener correctamente configurado el fichero *jdbc.properties* para los datos del entorno de producción.

Desde el entorno Eclipse ejecutaremos el siguiente objetivo de Maven para la creación del paquete *.war,* mediante el comando *Run As → Maven build...*



Especificamos el objetivo `package`:

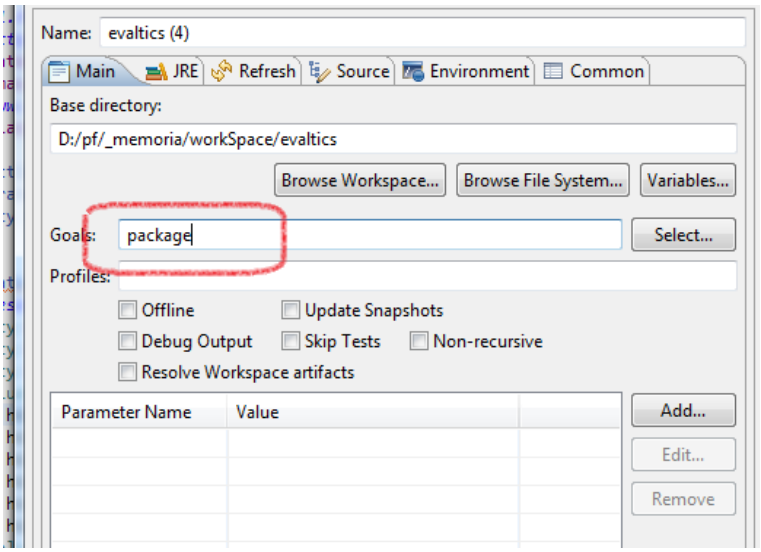


Ilustración 41: Objetivo Maven para la generación del empaquetado de la aplicación web

Una vez ejecutado se creará un fichero .war en el directorio target del proyecto. Esta es la aplicación web empaquetada y lista para desplegar en el servidor.

```
[INFO] Packaging webapp
[INFO] Assembling webapp [evaltics] in [D:\pf\_memoria\workSpace\evaltics\target\evaltics-0.0.1-SNAPS
[INFO] Processing war project
[INFO] Copying webapp resources [D:\pf\_memoria\workSpace\evaltics\src\main\webapp]
[INFO] Webapp assembled in [12578 msecs]
[INFO] Building war: D:\pf\_memoria\workSpace\evaltics\target\evaltics-0.0.1-SNAPSHOT.war
[WARNING] Warning: selected war files include a WEB-INF/web.xml which will be ignored
(webxml attribute is missing from war task, or ignoreWebxml attribute is specified as 'true')
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3:07.119s
[INFO] Finished at: Mon Sep 24 19:55:57 CEST 2012
[INFO] Final Memory: 21M/202M
[INFO] ------------------------------------------------------------------------
```

Ilustración 42: Registro de la ejecución del objetivo Maven package

Copiaremos este fichero en el directorio `webapps` de `Tomcat`. Si el servidor está funcionando automáticamente detectará la nueva aplicación y la descomprimirá en un directorio con el mismo nombre que el fichero *.war*. Este será el *path* relativo al que se accederá a la web, por lo que puede ser interesante modificar el nombre del fichero *.war* antes de copiar.

Una vez desplegada la aplicación, vemos como el directorio `webapps` ahora muestra el fichero *.war* y un directorio nuevo:

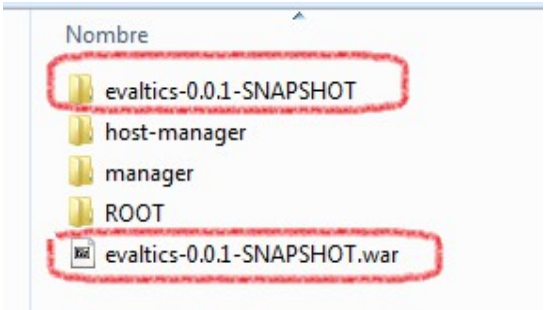


Ilustración 43: Contenido del directorio webapps de Tomcat

Si el servidor no está iniciado, el despliegue lo realizará automáticamente al arrancarse. Sólo queda comprobar que la aplicación funciona perfectamente en el entorno de producción accediendo a la dirección `http://localhost:[puerto]/evaltics-0.0.1-SNAPSHOT` (o la ruta creada para la aplicación)

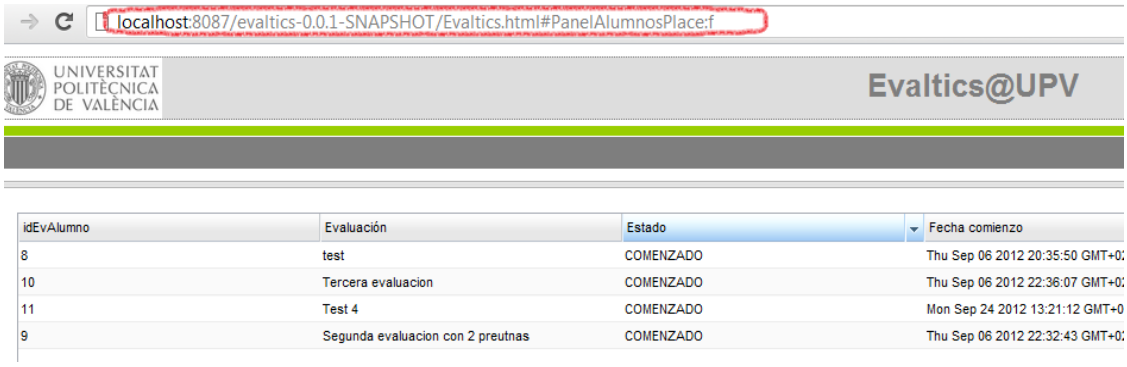


Ilustración 44: Aplicación Evaltics desplegada en producción

14. Conclusiones y mejoras

Dentro de las plataformas para el desarrollo de aplicación ricas basadas en web GWT supone cuanto menos una idea original con varios puntos a favor, al saber aprovechar las oportunidades que brinda usar código Java para el desarrollo de una aplicación completa, en contraposición a usar código Javascript, tradicionalmente muy poco estandarizado entre navegadores. Entre estos puntos a favor, podemos destacar:

- Integración en el entorno de desarrollo. Tradicionalmente los trabajos en Javascript no han estado soportados por buenos entornos de desarrollo.
- Relacionado con el anterior, el desarrollo de la aplicación con Java permite incrementar las posibilidades de refactorización y mantenimiento de código al estar basado en tipado fuerte.
- Posibilidades de tests con Junit.

El hecho de que la programación se realice en lenguaje Java debe suponer una pequeña curva de aprendizaje para muchos desarrolladores a la hora de enfrentarse a un proyecto web.

Sin embargo persisten algunas deficiencias. El entorno de trabajo basado en Eclipse es flojo, con continuos fallos y pequeños detalles que enlentecen el desarrollo. La integración con Maven es mucho menos que estable.

También el ciclo de desarrollo es lento, consumiendo muchos recursos al ejecutarse en modo desarrollo en el navegador.

La creación de interfaces gráficas es un terreno con muchas mejoras posibles, el diseñador es muy lento y consume muchos recursos. Es poco intuitivo y apenas incrementa la productividad en el desarrollo de una aplicación. Relacionado con esto, sería deseable una mejor integración de los componentes gráficos con los modelos de datos puesto que a veces es necesario mantener dos objetos para representar un mismo objeto de dominio.

Pero GWT está continuamente en desarrollo y estando en un entorno puntero y con plataformas muy competitivas presentando mejoras es de esperar que sigan implementando prestaciones, más teniendo en cuenta que está soportado por una empresa grande.

Como posibles mejoras o continuación de los trabajos podemos citar la posibilidad de desplegar una aplicación en la nube de forma sencilla con la integración de GWT y Google AppEngine. El primer paso sería la implementación de la capa de persistencia utilizando JPA. Google AppEngine soporta aplicaciones JPA con la librería DataNucleus.

Sin embargo existen inconvenientes en este métodos pues supone gastos extras si la aplicación es de consideración. También existen limitaciones en cuanto a las librerías a utilizarse por estar en un entorno cerrado. Tampoco hay que menospreciar las diferencias que existirán entre una aplicación desarrollada en una máquina y distribuida en un entorno completamente diferente y sin control.

15. Referencias y recursos

Documentación oficial de GWT
https://developers.google.com/web-toolkit

"GWT in action : easy Ajax with the Google Web toolkit"
Robert Hanson Adam Tacy Greenwich, CT : Manning cop. 2007

Ejemplos de utilización de la librería SmartGWT
http://www.smartclient.com/smartgwt/showcase/

Integración de Spring con GWT
http://raibledesigns.com/rd/entry/integrating_gwt_with_spring_security

Modelo *MVP* en GWT
http://www.slideshare.net/martyhall/gwt-tutorial-advanced-mvp-the-gwt-mvp-framework

Pruebas para el patrón *MVP*
http://blog.hivedevelopment.co.uk/2009/10/introduction-to-mvp-unit-testing-part.html

El patrón *MVP*
http://chuwiki.chuidiang.org/index.php?title=Patr%C3%B3n_Modelo-Vista-Presenter_(MVP)

Información sobre Apache Tomcat
tomcat.apache.org

Información sobre MySQL
dev.mysql.com

Plugin maven para Eclipse
http://mojo.codehaus.org/gwt-maven-plugin/

Información sobre el IDE Eclipse
www.eclipse.org

Seguridad en Spring
http://static.springsource.org/spring-security/site/reference.html

Appfuse, proyecto que integra distintas tecnologías para construir aplicaciones Web
http://appfuse.org/display/APF/Home

Índice de ilustraciones

Ilustración 1: Dirección para la instalación de plugin GWT....12
Ilustración 2: Opciones mínimas a instalar para el plugin de GWT en Eclipse....13
Ilustración 3: Opciones de instalación de plugin m2e....13
Ilustración 4: Selección de un proyecto Maven para un nuevo proyecto en Eclipse....14
Ilustración 5: Selección de un arquetipo adecuado para un proyecto GWT....15
Ilustración 6: Opciones del arquetipo gwt-maven-plugin....15
Ilustración 7: Estructura inicial del proyecto GWT Evaltics....16
Ilustración 8: Posibles errores en la vista Markers de Eclipse al generar el proyecto inicial....17
Ilustración 9: Configuración de un proyecto GWT en Eclipse....17
Ilustración 10: Resolución de problemas en Eclipse con los interfaces asíncronos....18
Ilustración 11: Ejecución de un objetivo Maven....18
Ilustración 12: Ejecución de objetivo de internacionalización del plugin Maven de GWT....19
Ilustración 13: Probando la primera versión del proyecto....19
Ilustración 14: Primera prueba de funcionamiento....20
Ilustración 15: Solicitud de plugin GWT para Internet Explorer....21
Ilustración 16: Modelo MVP....24
Ilustración 17: Creación de un paquete....27
Ilustración 18: Estructura de paquetes de la aplicación....27
Ilustración 19: Crear un nueva vista MVP....28
Ilustración 20: Datos para una vista MVP....28
Ilustración 21: Estructura del proyecto tras implementar MVP....30
Ilustración 22: Crear interfaces asíncronos con ayuda de eclipse....36
Ilustración 23: Actualizar interfaces asíncronos con ayuda de eclipse....36
Ilustración 24: Estado del proyecto tras implementar los servicios....42
Ilustración 25: Configurar un proyecto para utilizar SmartGWT....43
Ilustración 26: Creación de interfaces gráficas con GWT Designer....44
Ilustración 27: Pantalla inicial de Evaltics....46
Ilustración 28: Mantenimiento de asignaturas....47
Ilustración 29: Gestión de alumnos y matrículas....47
Ilustración 30: Gestión de respuesta de una pregunta de tipo V/F....48
Ilustración 31: Creación de una nueva evaluación....49
Ilustración 32: Pantalla para la validación de credenciales de un alumno....50
Ilustración 33: Evaluaciones disponibles para un alumno....50
Ilustración 34: Realizando una evaluación....51
Ilustración 35: Modelo de base de datos para la aplicación Evaltics....53
Ilustración 36: Estructura del proyecto Evaltics tras configurar Hibernate....59
Ilustración 37: Estructura del proyecto tras introducir los DAO....61
Ilustración 38: Creación de un nuevo caso de prueba....66
Ilustración 39: Propiedades del caso de prueba de alumnoManager....67
Ilustración 40: Apache Tomcat funcionando correctamente....78
Ilustración 41: Objetivo Maven para la generación del empaquetado de la aplicación web....80
Ilustración 42: Registro de la ejecución del objetivo Maven package....80
Ilustración 43: Contenido del directorio webapps de Tomcat....81
Ilustración 44: Aplicación Evaltics desplegada en producción....81

Índice de listados

Listado 1 : Definición del contrato de un servicio en GWT....21
Listado 2 : Código cliente para obtener una referencia al un servicio remoto....21
Listado 3 : Código del cliente para llamar a un servicio remoto....22
Listado 4 : Especificación de un módulo GWT....24
Listado 5 : Código del entrypoint de la aplicación con el modelo MVP....26
Listado 6 : Interfaz de un presentador en MVP....29
Listado 7 : Mapeo de actividades con lugares en MVP....29
Listado 8 : Dependencias Maven para el proyecto....33
Listado 9 : Versión inicial del fichero de configuración de la aplicación web....34
Listado 10 : Definición de un servicio RPC....35
Listado 11 : Servlet para servir servicios RPC con Spring....35
Listado 12 : Definición de un servicio RPC....35
Listado 13 : Servicio asíncrono....36
Listado 14 : Dependencias Maven para la librería de uso JSON....37
Listado 15 : Dependencias para publicar servicios REST....37
Listado 16 : Configuración del ficher cxf-servlet.xml....38
Listado 17 : Definición de servicios en CXF....38
Listado 18 : Creación de un servicio REST....39
Listado 19 : Implementación de servicio REST....40
Listado 20 : applicationContext.xml....40
Listado 21 : Configuración del servlet CXF en web.xml....41
Listado 22 : Referencia a la configuración de CXF en web.xml....41
Listado 23 : Anotaciones en Hibernate....54
Listado 24 : Anotaciones para asociaciones en Hibernate....55
Listado 25 : Dependencias para Hibernate....55
Listado 26 : applicationContext-dao.xml....56
Listado 27 : applicationResources.xml....57
Listado 28 : Propiedades de la conexión a base de datos....57
Listado 29 : hibernate.cfg.xml....58
Listado 30 : Interfaz de un DAO....60
Listado 31 : Implementacion de un DAO en Hibernate....61
Listado 32 : Llamada RPC desde el cliente....62
Listado 33 : Modificación del módulo para añadir soporte JSON al cliente....63
Listado 34 : Llamada a un servicio REST desde el cliente....64
Listado 35 : Convertir un respuesta de un servicio a objetos en Javascipt....64
Listado 36 : Dependencia para mockito para realizar tests unitarios....67
Listado 37 : Código del test de prueba del cifrado de la contraseña....68
Listado 38 : Dependencias de Spring security....70
Listado 39 : Definición del filtro para la seguridad en web.xml....70
Listado 40 : Cambios en web.xml para la seguridad de Spring....71
Listado 41 : Definición de los proveedores de autenticación....71
Listado 42 : Anotación para securizar a nivel de método....72
Listado 43 : Código de la excepción propia de Evaltics para el acceso denegado....73
Listado 44 : Definición del gestor de decisión de acceso propio....73
Listado 45 : Función para obtener los roles asociados a un usuario....74
Listado 46 : Asignación de roles a un autor....74
Listado 47 : Definición del bean para cifrar la contraseña....75
Listado 48 : Inyección del cifrador de contraseña en los beans de alumno y autor....75
Listado 49 : Cifrando la clave al guardar en base de datos....76
Listado 50 : Creación de esquema y usuario para Evaltics en producción....79