UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de una APP para divulgar resultados del proyecto cyanofactory

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Adrián Cervera Gómez

Tutor: Dr. Lenin Guillermo Lemus Zúñiga

Curso 2016-2017

Resumen

Los métodos de formación y aprendizaje han sufrido un cambio radical desde la irrupción de la Web 2.0, en la que el usuario abandona el rol pasivo para obtener uno más enfocado a la interacción y la aportación de contenido en la web. La formación online es una de las opciones que mas crecimiento ha tenido a lo largo de los últimos años ya que ofrece numerosas ventajas. La principal es la de añadir, a la ya complicada conciliación del trabajo y la vida personal, la posibilidad de continuar con una formación constante que permita mejorar el crecimiento profesional. También sirve para aquellas personas a las que el desplazamiento contínuo al lugar de estudio les suponga un problema.

La formación online aporta otras ventajas como la mejora de la autodisciplina y organización del alumno, al tener que llevar el progreso de su formación de manera independiente, siguiendo su propio ritmo.

El objetivo de este proyecto es proporcionar una plataforma online que ofrezca y permita administrar cursos de formación. Que permita acceso a contenido y evaluaciones para los alumnos, gestión del temario y evaluación de los alumnos para el profesorado, y gestión del sistema para los administradores.

Actualmente ya existen muchas plataformas que ofrecen una buena solución, pero uno de los objetivos secundarios de este proyecto es aprender a implementar este tipo de tecnología, que en el futuro se puede adaptar fácilmente para la integración de herramientas para la mejora del aprendizaje del alumno mediante la medición y toma de métricas del uso de la plataforma (learning analytics).

Palabras clave: APP, Spring MVC, AngularJS, Hibernate, MySql, REST, Json, HTML, Bootstrap

Resum

Els mètodes de formació i aprenentatge han sofert un canvi radical des de la irrupció de la Web 2.0, en la qual l'usuari abandona el rol passiu per obtenir un més enfocat a la interacció i l'aportació de contingut a la web. La formació online és una de les opcions que mes creixement ha tingut al llarg dels últims anys, ja que ofereix nombrosos avantatges. La principal és la d'afegir a la ja complicada conciliació del treball i la vida personal la possibilitat de continuar amb una formació constant que permeti millorar el creixement professional. També serveix per a aquelles persones a les quals un desplaçament contínuat al centre d'estudi els suposi un problema.

La formació online aporta altres avantatges com la millora de l'autodisciplina i organització de l'alumne, por haver de portar el progrés de la seva formació de manera independent, marcant-se un ritme propi.

L'objectiu d'aquest projecte és proporcionar una plataforma online que ofereixi i permeti administrar cursos de formació. Que permeti l'accés a contingut i avaluacions per als alumnes, gestió del temari i avaluació dels alumnes per al professorat, i gestió del sistema per als administradors.

Actualment ja existeixen moltes plataformes que ofereixen una bona solució, però un dels objectius secundaris d'aquest projecte és aprendre a implementar aquest tipus de tecnologia, que en el futur es pot adaptar fàcilment per a la integració d'eines per a la

millora de l'aprenentatge de l'alumne mitjançant el mesurament i presa de mètriques de l'ús de la plataforma (learning analytics).

Paraules clau: APP, Spring MVC, AngularJS, Hibernate, MySql, REST, Json, HTML, Bootstrap

Abstract

The training and learning methods have suffered a radical change since the onset of Web 2.0, in which the user leaves a passive role to obtain a more focused interaction and contribution of content on the web. On-line training is one of the options that has most grown over the last few years as it offers numerous advantages. The main one is to add to the already complicated conciliation of work and personal life the possibility of continuing with a constant formation that allows to improve a professional growth. It also works for those people who have problems with continuous displacement to study centers.

The on-line training brings other advantages such as improving the student's self-discipline and auto-organization, having to take the progress of their training independently, with their own pace.

The main goal of this project is to provide an on-line platform that offers and allows to manage training courses. That allows access to content and assessments for students, management of involved temary and evaluation of students for the teachers, and easy management of the system for administrators.

Currently there are many platforms that offer a good solution, but one of the secondary objectives of this project is to learn how to implement this type of technology, which in the future can be easily adapted for the integration of tools to improve student learning through measurement of metrics of the usage of the platform (learning analytics).

Key words: APP, Spring MVC, AngularJS, Hibernate, MySql, REST, Json, HTML, Bootstrap

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Motivación

Una de las áreas educativas que tiene una gran demanda es la formación online, siendo fundamental que un ingeniero informático sepa como diseñar, implementar y mantener este tipo de aplicaciones.

Este tipo de plataformas deben ser diseñadas utilizando tecnologías actuales y de uso extendido. Actualmente, la tendencia es utilizar las conocidas como *Single Page App* (SPA). Una SPA permite la comunicación entre cliente y servidor y la actualización de los contenidos de la vista de manera transparente al usuario, sin tener que recargar la página, gracias al framework de JavaScript AngularJS. Por el lado del servidor, Spring MVC, uno de los frameworks más utilizados en proyectos Java EE, junto con Hibernate para la persistencia de datos y MySql como sistema de gestión de la base de datos.

Además, el diseño e implementación de una plataforma online basada en una SPA me permitirá aprender a utilizar este tipo de tecnologías, debido a que en los actuales planes de estudio de la carrera de GII este tipo de tecnologías se ve con poca profundidad.

1.2 Objetivos

El objetivo principal del proyecto es la implementación de una plataforma online para impartir cursos académicos de una manera sencilla e intuitiva, que facilite al alumnado el acceso al temario y al profesor la generación de contenido y la evaluación del alumno.

Como objetivo secundario se tiene el autoaprendizaje del desarrollo de aplicaciones WEB utilizando la tecnología SPA y el framework Spring.

El sistema deberá ser lo suficiente ligero como para funcionar en todo tipo de dispositivos, incluyendo los de menor rendimiento. Se debe garantizar el acceso a los datos en todo momento, evitando sobrecargar la BBDD con información no necesaria, y se deben obtener tiempos de respuesta aceptables mediante consultas optimizadas para el objetivo que se persiga gracias al uso de tecnología actual. Para ello usaremos las tecnologías que han tenido mayor éxito en los últimos años.

1.3 Estructura de la memoria

Este primer capítulo explica una introducción al objeto de este proyecto. La motivación que ha llevado a su desarrollo y realización, los objetivos marcados para éste.

En el segundo capítulo se detalla la Especificación de Requerimientos, según las directrices del estándar IEEE 830. Ésta se realiza sobre los objetivos marcados en la introducción, muestra la funcionalidad esperada y limita el alcance final del proyecto.

En el tercer capítulo se explica de que manera se va a implementar la aplicación, siguiendo una arquitectura determinada.

En el cuarto capítulo se explicará la parte de cliente, que utilizará la tecnología de Angular JS.

En el quinto capítulo entraremos en detalle para la parte de servidor, explicando el funcionamiento de Spring MVC.

En el sexto capítulo se detallará la implementación de la aplicación que se ha desarrollado durante este proyecto.

En el séptimo capítulo se realizará una serie de pruebas para cada uno de los actores del sistema, verificando que funciona de la manera esperada y que se cumplen los objetivos contemplados en el capítulo segundo.

En el octavo y último capítulo se recogen las conclusiones obtenidas durante la realización de este proyecto.

CAPÍTULO 2
Especificación de Requerimentos

2.1 introducción

En este capítulo trataremos la Especificación de Requisitos de Software (ERS) sobre el sistema que tratamos en este proyecto. Esta especificación se ha estructurado basándose en las directrices dadas por el estándar IEEE 830.

2.1.1. Propósito

El propósito de este capítulo será definir las especificaciones funcionales (casos de uso) de la aplicación desarrollada. El documento va dirigido tanto a todos los usuarios de la aplicación: estudiantes, profesores y administradores, como a futuros desarrolladores que pretendan ampliar el sistema, implementando mejoras o nueva funcionalidad al sistema.

2.1.2. Ámbito del sistema

El objetivo del proyecto es la implementación de una plataforma online para impartir cursos académicos de una manera sencilla e intuitiva, que facilite al alumnado el acceso al temario y al profesor la generación de contenido y la evaluación del alumno.

El sistema deberá ser lo suficiente ligero como para funcionar en todo tipo de dispositivos, incluyendo los de menor rendimiento. Se debe garantizar el acceso a los datos en todo momento, evitando sobrecargar la BBDD con información no necesaria, y se deben obtener tiempos de respuesta aceptables mediante consultas optimizadas para el objetivo que se persiga gracias al uso de tecnología actual. Para ello usaremos las tecnologías que han tenido mayor éxito en los últimos años.

2.1.3. Definiciones, acrónimos y abreviaturas

- Alumno. Usuario del sistema que accederá a los contenidos de los cursos matriculados y realizará las evaluaciones.
- Profesor. Usuario del sistema que gestionará los cursos y evaluaciones de los alumnos.
- Administrador. Usuario que gestionará el sistema: crear, eliminar y editar cursos o usuarios, asignar el profesor responsable de un curso, inscribir o desinscribir alumnos en un curso.

- Curso. Conjunto del material creado y recopilado para ofrecer formación sobre una materia determinada. Contiene un conjunto de temas, teoría, ejercicios para lograr este objetivo. Cada curso está gestionado por un profesor, y en él pueden inscribirse los alumnos.

- Tema. Conjunto de secciones que forman un curso. Cada tema tiene su propio contenido teórico y ejercicios relacionados. Son directamente gestionados por el profesor. El alumno puede acceder a los temas y sus contenidos de un curso en el que esté inscrito.

- Teoría. Contenido teórico de un tema. Gestionado por el profesor. El alumno puede consultarlo en modo lectura.

- Ejercicio. Contenido interactivo de un tema. Hay de dos tipos, que se muestran en apartados diferentes para cada tema: los ejercicios del tema y los que conforman un examen. El alumno puede ofrecer una respuesta, que será evaluada por el profesor responsable del curso. El profesor puede crear tantos ejercicios o ejercicios de examen como desee. El alumno puede consultarlos, responder y consultar la solución del ejercicio y la calificación obtenida una vez ha sido evaluado por el profesor.

- Evaluación. Respuesta a un ejercicio efectuada por el alumno. Una evaluación está relacionada con un ejercicio concreto y con un alumno concreto. El profesor se encargará de corregir y evaluar la solución del alumno.

2.1.4. Referencias

IEEE SA – 830-1998. Recommended Practice for Software Requirements Specifications (https://standards.ieee.org/findstds/standard/830-1998.html)

2.1.5. Visión general del documento

En el primer apartado se ha mostrado una visión general del proyecto y sus objetivos, sin especificar detalles técnicos. Durante los siguientes apartados se proporcionará información más detallada sobre sus características específicas.

2.2 Descripción general

2.2.1. Perspectiva del producto

El sistema se basará en una aplicación de servidor, independiente de otros sistemas, diseñada para funcionar en un entorno web utilizando las tecnologías propias.

2.2.2. Actores del sistema

En esta sección se presentan los actores del sistema, que serán los tres tipos de usuarios que darán uso de la aplicación:

ALUMNO:

- Estudiante. Su objetivo es recibir la formación desarrollada por el profesor.

- Acceso a los cursos en los que está inscrito. Acceso al temario, el contenido teórico y los ejercicios que componen el curso.
- Recibe una evaluación.

PROFESOR:

- Experto en la materia a impartir.
- Acceso a los cursos de los que es responsable. Creación y edición del contenido del curso.
- Evaluación de los alumnos inscritos en sus cursos.

ADMINISTRADOR:

- Técnico / administrador de sistemas.
- Gestión y control del sistema. Proporciona soporte hacia los usuarios.
- Crea, actualiza o elimina cursos y usuarios. Asigna al profesor responsable de un curso. Inscribe o desinscribe alumnos en los cursos.

2.2.3. Funciones del producto (casos de uso del sistema)

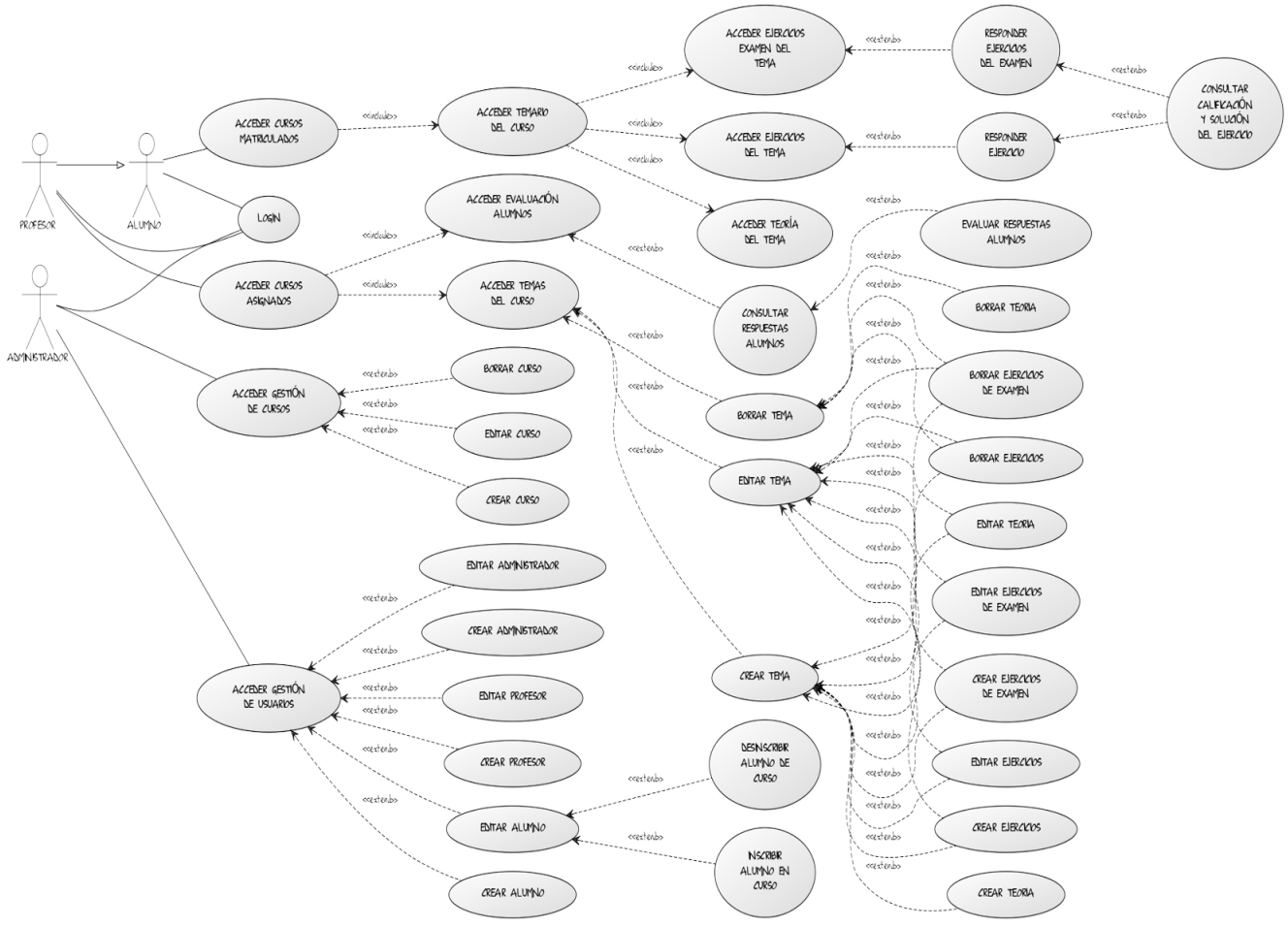


Figura 2.1: Diagrama de casos de uso

2.2.4. Diagrama de clases

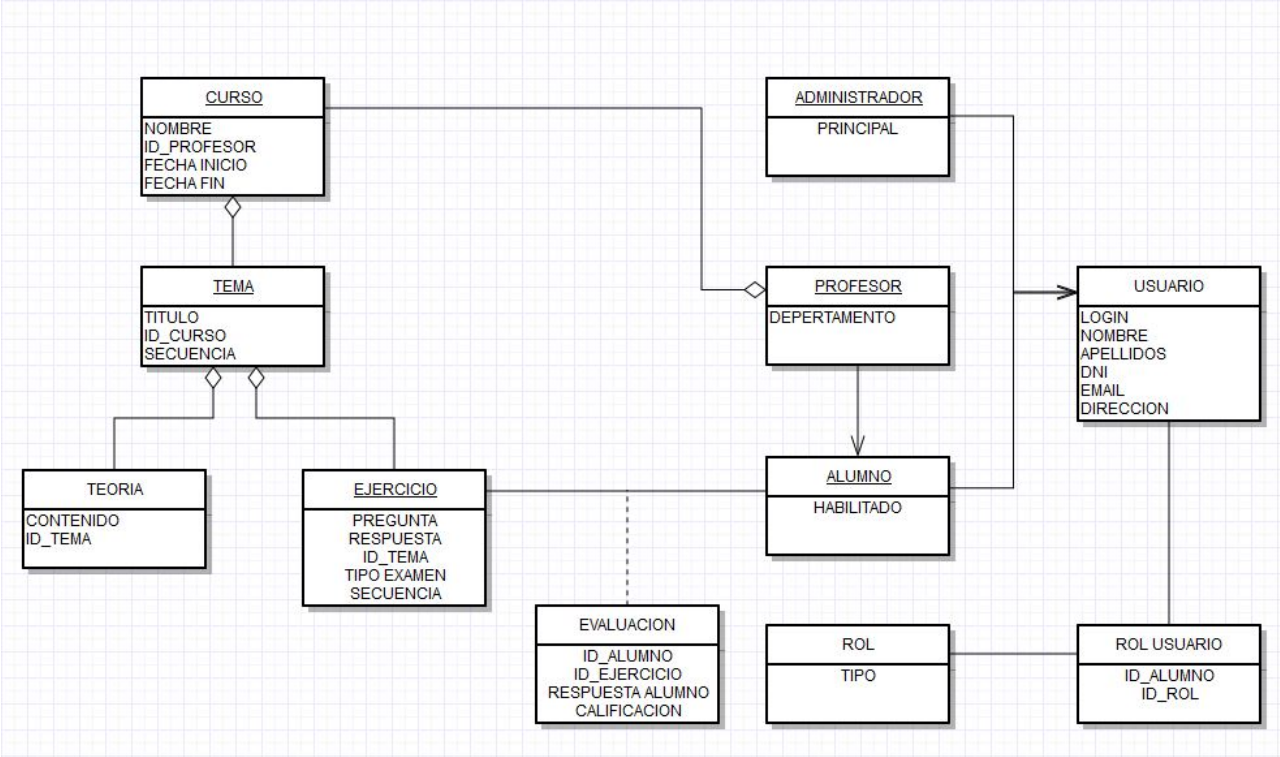


Figura 2.2: Diagrama de clases

2.2.5. Restricciones

La aplicación se limita a su uso a través de un navegador web. Se desarrollará utilizando HTML, CSS, Bootstrap, AngularJS 1.5.8, Java(servidor), Hibernate con MySQL para la persistencia. Utilizará una arquitectura cliente-servidor a 3 capas (presentación, lógica de negocio y de datos). Tendrá una interfaz sencilla, intuitiva y fácil de manejar.

2.2.6. Suposiciones y dependencias

La aplicación será compatible con cualquier sistema moderno actual siempre que el navegador web soporte las tecnologías relacionadas con su uso antes mencionadas, además de conexión a Internet.

2.3 Requisitos específicos

2.3.1. Interfaces externas

Se ofrecerá al usuario una interfaz de una sóla página. Tras acceder al sistema mediante el login, cada tipo de usuario accederá a su vista principal de usuario.
La vista del alumno ofrece un desplegable en el que seleccionar uno de sus cursos matriculados. Al seleccionarlo, se cargará dinámicamente el contenido en el menú lateral izquierdo de la vista. Podrá acceder a los temas que componen en curso con todos sus contenidos.
La vista de profesor muestra un listado de cursos a gestionar. Al seleccionar un curso,

podrá acceder a la edición de su contenido o a evaluar a los alumnos inscritos.
La vista de administrador ofrece un menú para seleccionar si desea gestionar los cursos del sistema o los usuarios del sistema, organizados por su rol.

Las vistas deben ser adaptativas para cualquier tipo de pantalla (responsive).

La aplicación no debe de depender de ninguna plataforma en concreto. Se debe de poder usar desde cualquier tipo de sistema.

2.3.2. Funciones

ALUMNO:

- Log-in/log-out del sistema.
- Acceder a cursos registrados. Desde cada uno dispondrá de acceso a los diversos temas que componen cada curso, que contienen el contenido teórico, ejercicios del tema y ejercicios de exámenes que podrá responder y donde podrá consultar su calificación y la solución correcta de los ejercicios.

PROFESOR:

- Log-in/log-out del sistema.
- Editar el contenido de sus cursos asignados. Añadir, modificar o borrar temas, que contienen el contenido teórico, actividades y ejercicios de examen.
- Evaluar a los alumnos conforme vayan respondiendo los ejercicios de cada tema.

ADMINISTRADOR:

- Log-in/log-out del sistema.
- Crear o eliminar cursos existentes, asignando un profesor responsable a cada uno de ellos.
- Dar de alta o modificar usuarios del sistema.
- Inscribir o desinscribir los alumnos del sistema.

2.3.3. Requisitos de Rendimiento

La carga de usuarios dependerá del total de usuarios por curso, multiplicado por el número de cursos existentes. Al ser de acceso no presencial se espera a que la carga del sistema debida al número de conexiones simultáneas se distribuya entre las horas de mayor actividad de la jornada, entre la mañana y la tarde.
El grado de utilización de la base de datos será muy alto en cuanto al número de peticiones de lectura. La información enviada entre cliente y servidor será de poco peso, ya que se utiliza REST y Json, pero la información que se almacena en el lado del cliente es mínima y para cada botón o enlace de la aplicación que se pulse se recargará la información desde el servidor de manera dinámica, por lo que obtendremos un alto número de peticiones de lectura a la base de datos.

2.3.4. Restricciones de Diseño

El sistema deberá ser lo suficiente ligero como para funcionar en todo tipo de dispositivos, incluyendo los de menor rendimiento. Se debe garantizar el acceso a los datos en todo momento, evitando sobrecargar la BBDD con peticiones no necesarias, y se deben obtener tiempos de respuesta aceptables mediante consultas optimizadas para el objetivo que se persigue. Para lograrlo usaremos las tecnologías que se describirán en esta memoria.

2.3.5. Atributos del Sistema

Garantizar la seguridad mediante el acceso identificado. Cada tipo de usuario podrá acceder únicamente a los elementos correspondientes a sus funciones, y se deberá impedir el acceso a recursos no autorizados desde el lado de servidor. Se deberá poder acceder desde cualquier sistema que disponga de un navegador moderno, como Firefox, Chrome o Safari, que tenga Javascript habilitado. Dado el carácter online y de libre acceso de la aplicación, la disponibilidad del sistema debe estar garantizada las 24 horas al día los 7 días de la semana, de manera contínua.

2.4 Paneles (para cada actor)

2.4.1. Login

Figura 2.3: Vista de bienvenida

La vista de inicio de la aplicación. Permite seleccionar el rol con el que acceder al sistema.

2.4.2. Alumno

Figura 2.4: Vista de alumno

Ésta es la vista principal para el alumno. Desde el botón superior puede acceder a los cursos en los que está inscrito. Al seleccionar un curso, en el menú lateral aparecerá el listado de temas que compone el curso. Desplegando cada tema, podrá acceder al contenido teórico, a los ejercicios y al examen del tema.

Figura 2.5: Vista de un ejercicio

Ejemplo de un ejercicio que compone el examen del tema. Se podrá enviar una respuesta. Solo cuando el ejercicio haya sido respondido por el alumno y calificado por el profesor, podrá consultar la calificación y la solución del ejercicio en esta misma vista.

2.4.3. Profesor

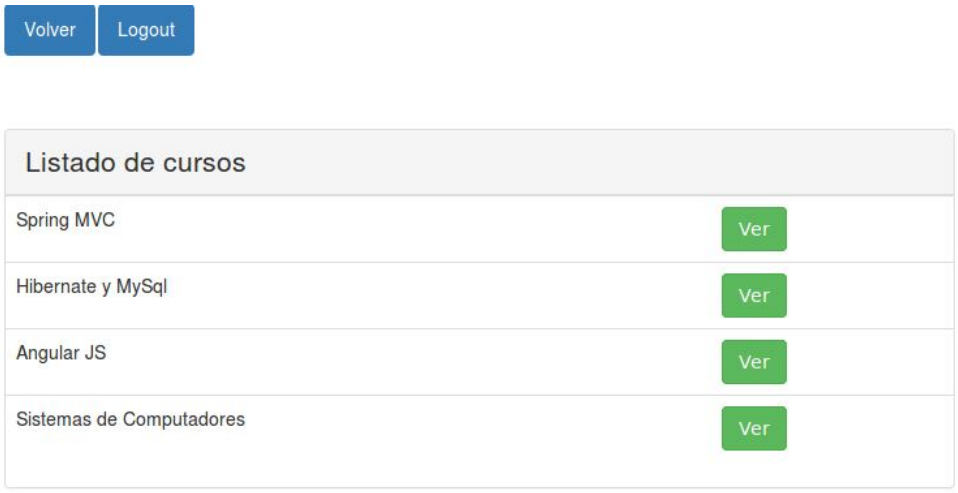


Figura 2.6: Listado de cursos asignados

Al acceder al sistema, el profesor se encontrará con un listado de cursos de los que es responsable. Podrá acceder a cada uno de ellos para modificar su contenido y evaluar a los alumnos.

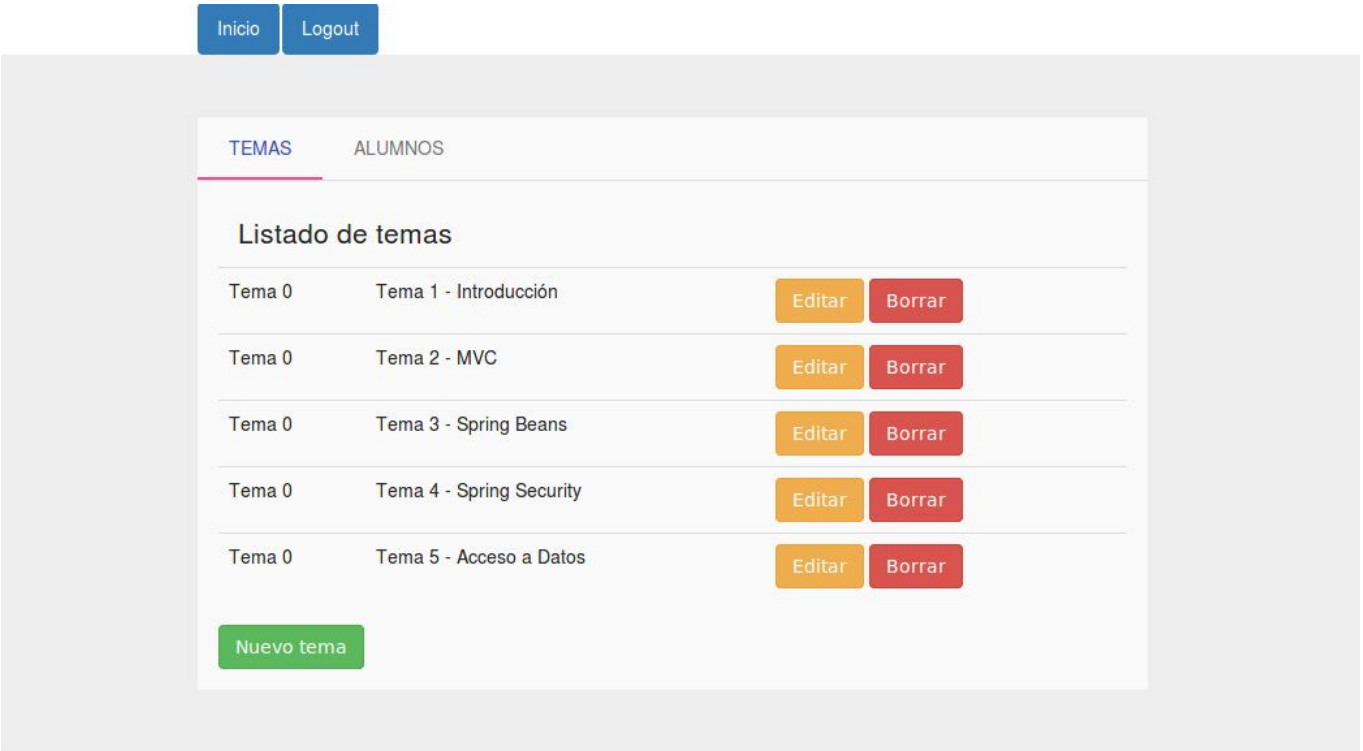


Figura 2.7: Listado de temas de un curso

Al acceder a un curso aparecerá una vista con dos pestañas. Una contiene el listado de temas que componen el curso, desde aquí podrá crear. editar o borrar el temario. La otra contiene el listado de alumnos inscritos en este curso. Desde aquí podrá ir evaluando las respuestas enviadas por el alumno, ordenadas por tema.

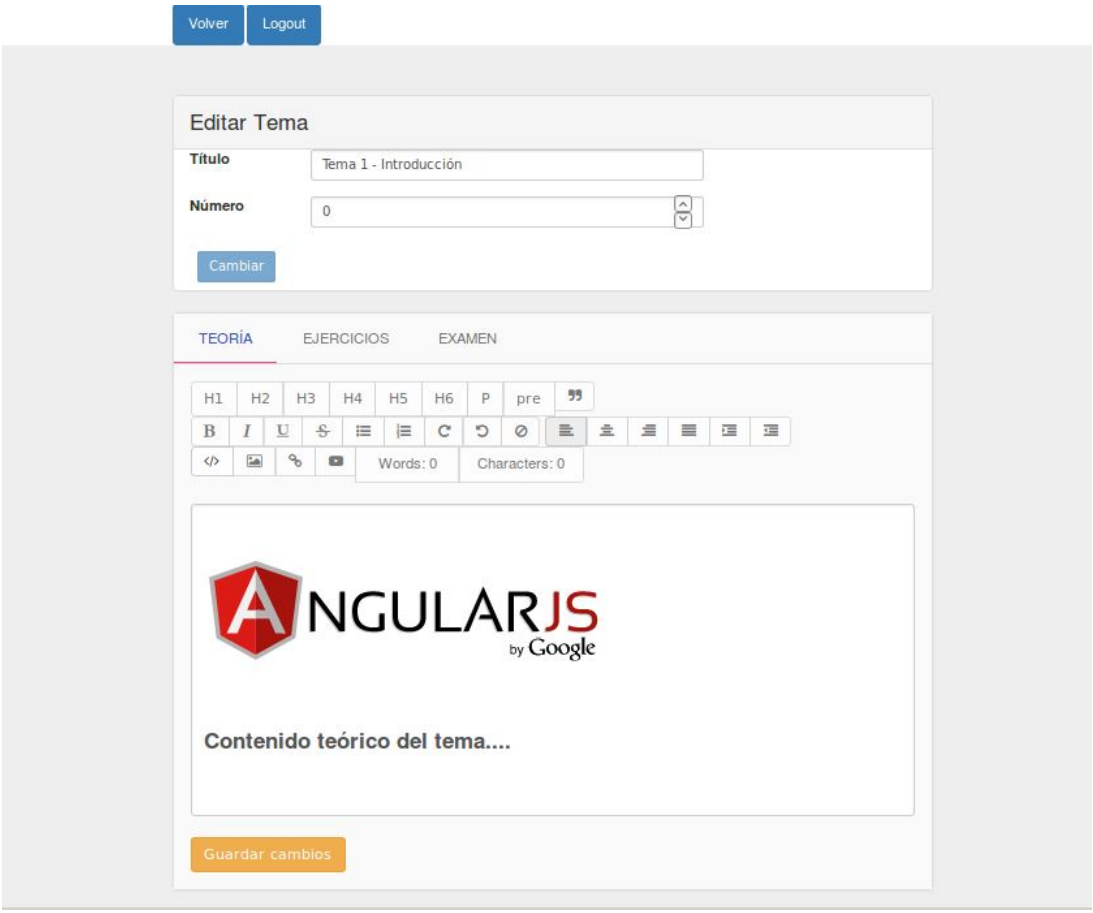


Figura 2.8: Crear/editar un tema. Teoría

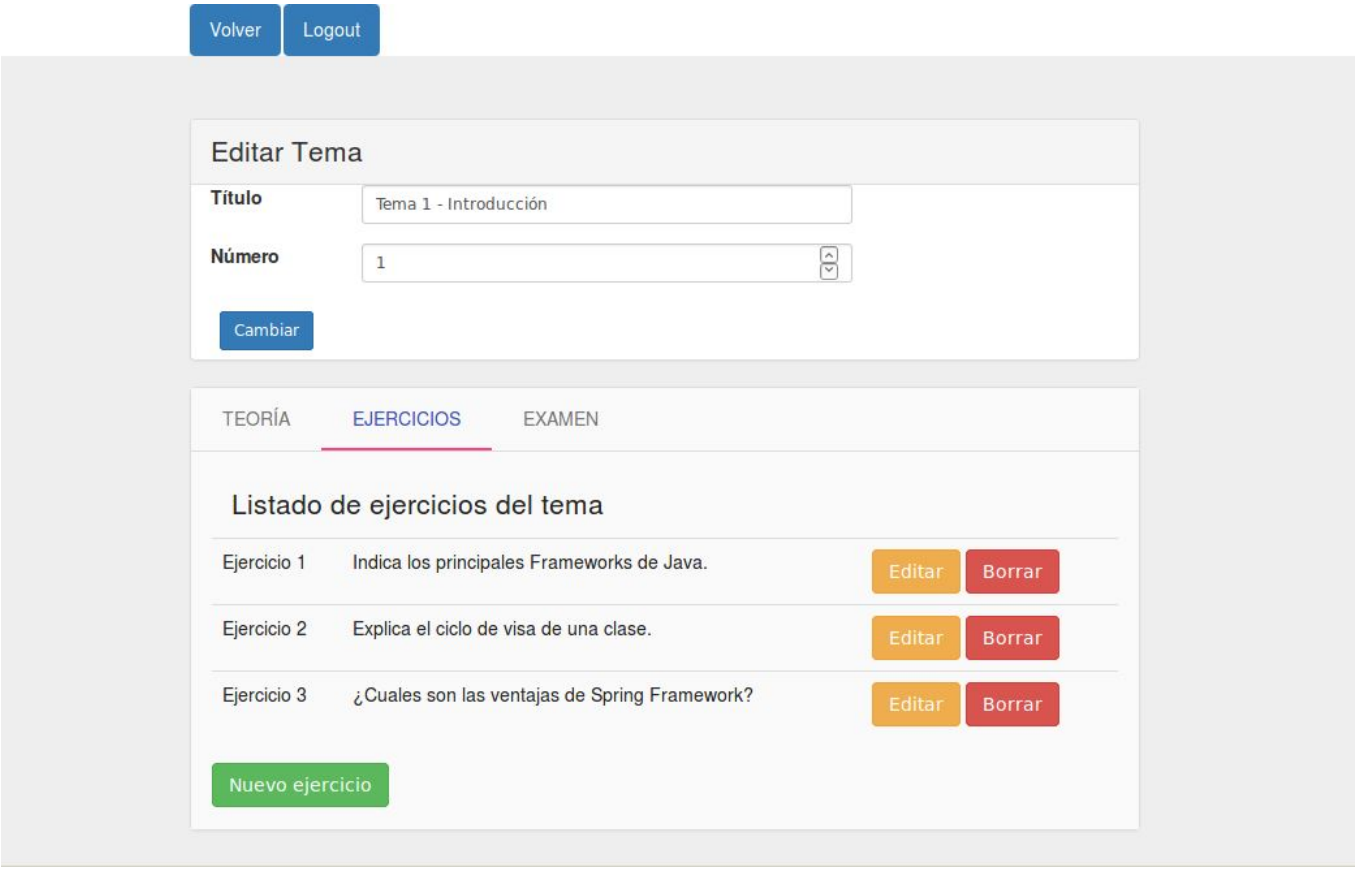


Figura 2.9: Crear/editar un tema. Listado de ejercicios

Al crear o editar un tema, se puede añadir el contenido teórico y crear los ejercicios del tema y los ejercicios del examen.

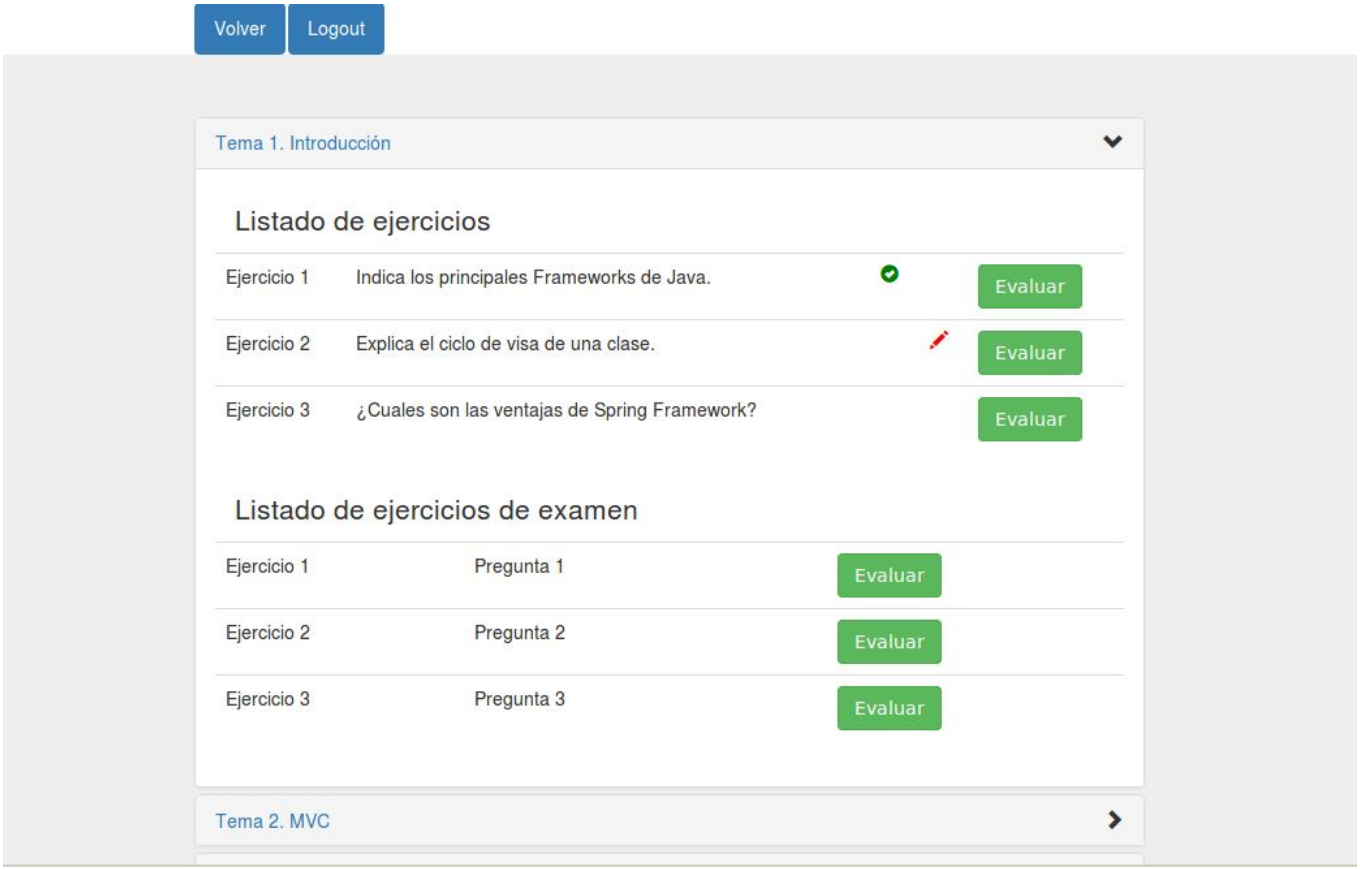


Figura 2.10: Listado de evaluaciones del alumno

Al seleccionar un alumno del curso, se podrá consultar el progreso del alumno respecto al temario. Ordenados por temas, podemos consultar si el alumno ha respondido un ejercicio (icono bolígrafo rojo) o si el ejercicio ya ha sido evaluado (icono check verde).

2.4.4. Administrador

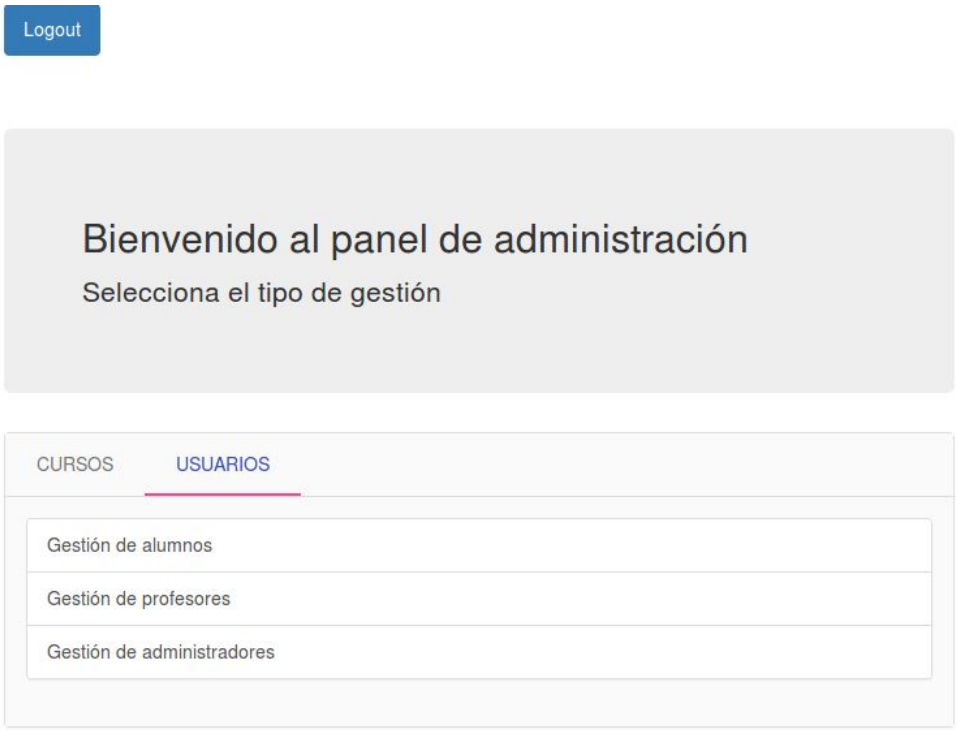


Figura 2.11: Panel de administración

Al acceder al sistema, el administrador tendrá acceso al panel principal de administración. Podrá acceder a la gestión de cursos y a la gestión de usuarios del sistema.

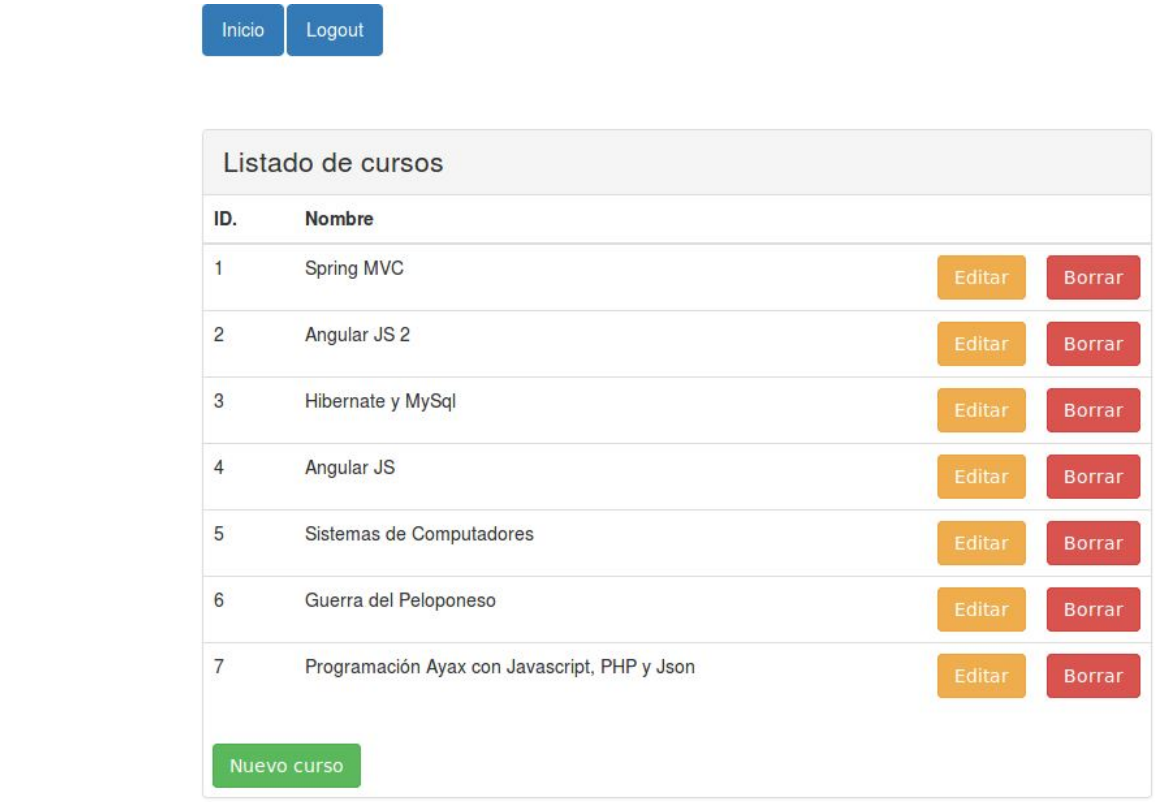


Figura 2.12: Listado de cursos creados

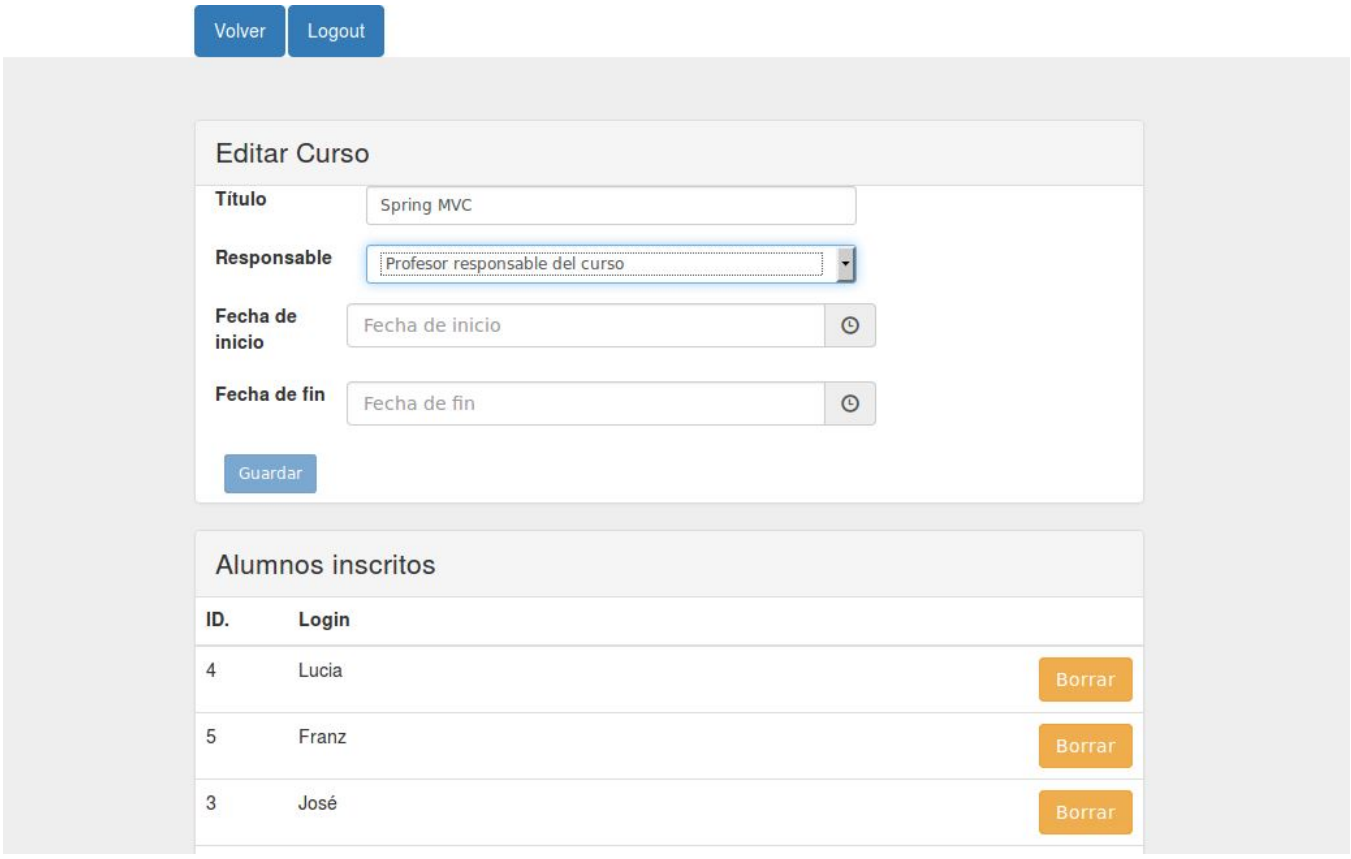


Figura 2.13: Crear/editar un curso

Desde esta vista el administrador podrá crear, editar o borrar cursos. Al crear o editar un curso, podrá asignar al profesor responsable del curso. También se muestra en esta vista los alumnos inscritos en el curso, y se permite desinscribirlos. Al borrar un curso, se borrará en cascada todo el temario con su contenido y evaluciones relacionadas.

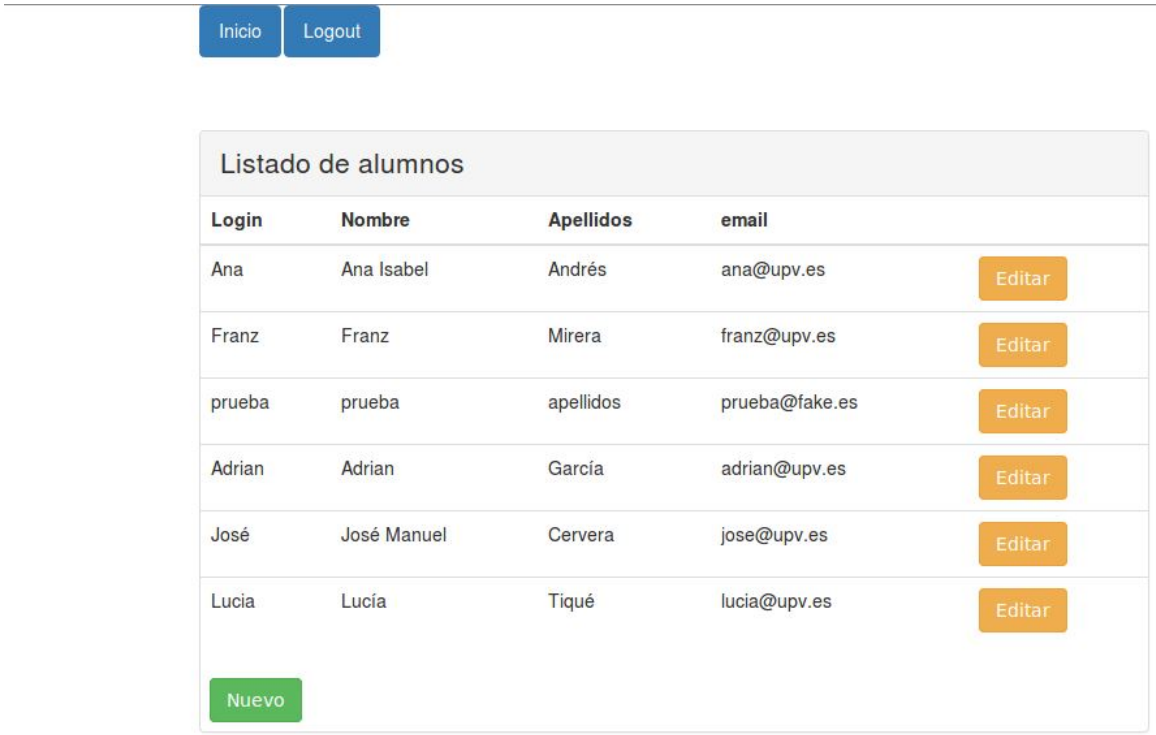


Figura 2.14: Gestión de usuarios

Desde la gestión de usuarios podrá acceder al listado de alumnos, profesores y administradores del sistema, desde donde podrá crear nuevos usuarios o editar los existentes. Los alumnos se pueden habilitar o deshabilitar para que no puedan acceder al sistema.

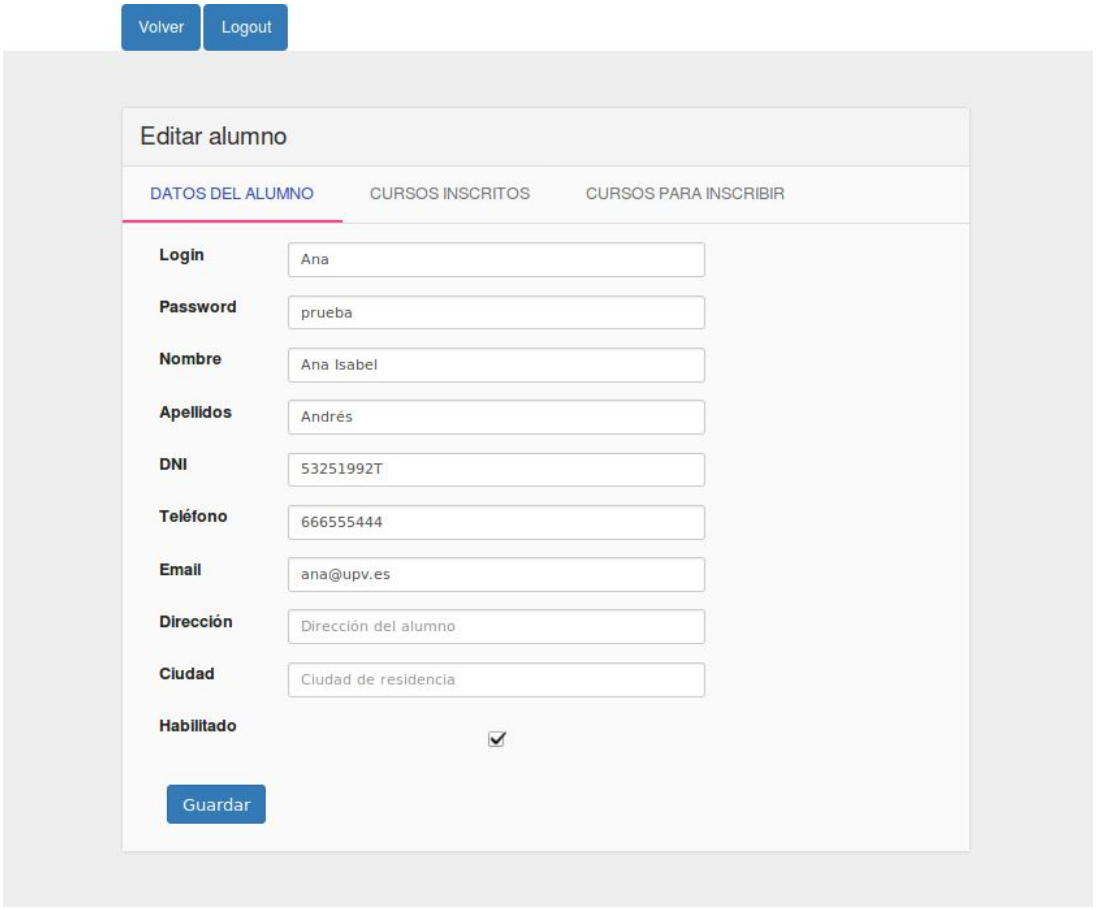


Figura 2.15: Editar usuario (alumno)

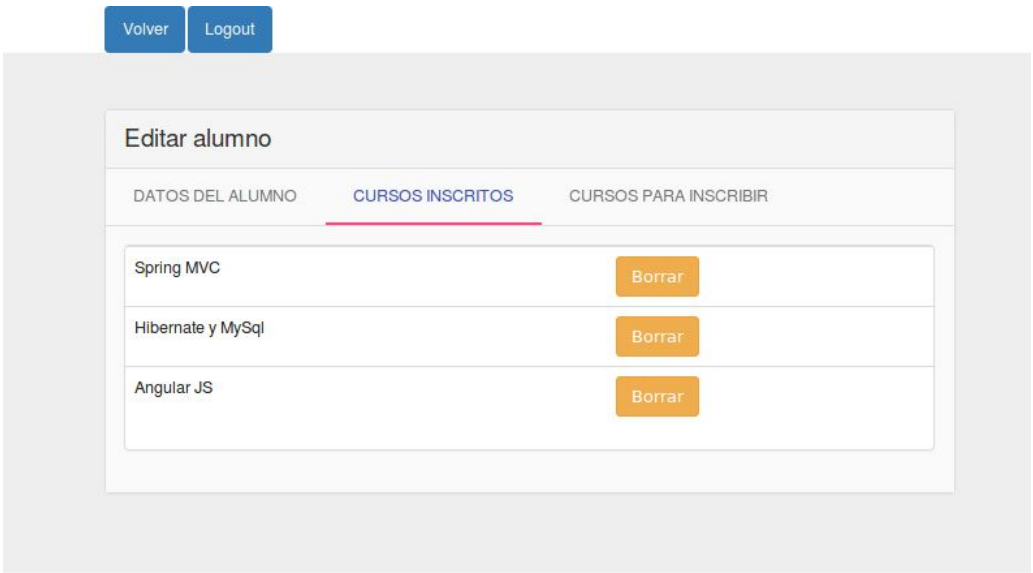


Figura 2.16: Cursos inscritos del alumno

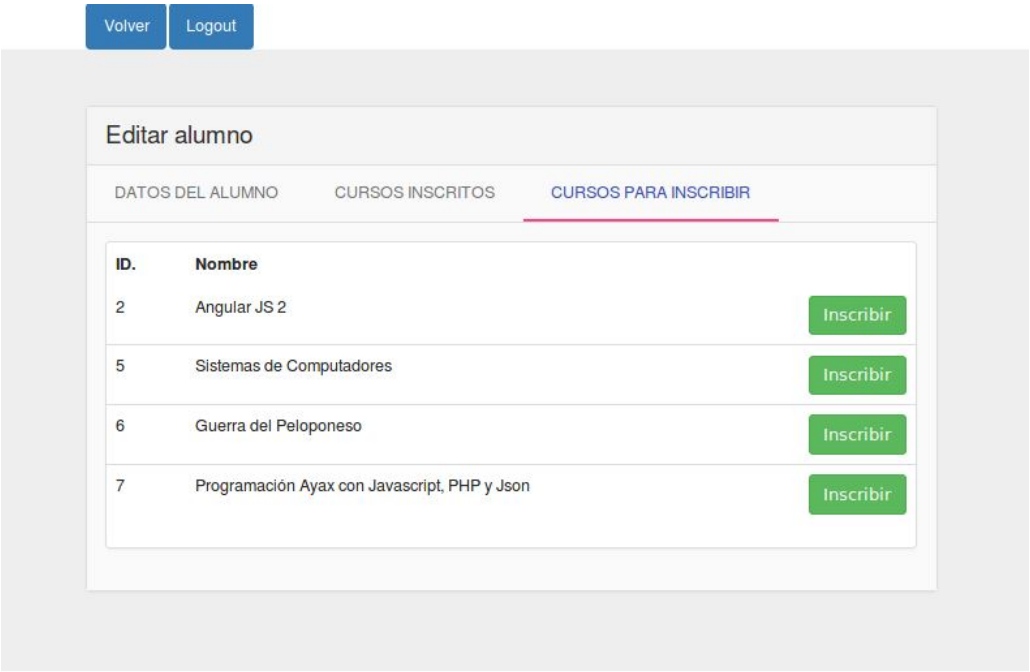


Figura 2.17: Cursos en los que no está inscrito el alumno

Desde el menú de gestión de alumnos, se puede inscribir o desinscribir a cada alumno de los distintos cursos existentes.

CAPÍTULO 3
Arquitectura de la aplicación

La arquitectura de software define cómo se va a conformar el sistema. Define sus componentes, su interfaz, y la comunicación que existirá entre ellos.

Para el desarrollo de este proyecto utilizaremos el patrón de arquitectura Modelo – Vista – Controlador (MVC), que divide la lógica de negocio de la interfaz de usuario. Se divide en tres componentes diferenciados: el modelo, que define la lógica de negocio; la vista, que muestra la información al usuario; y el controlador, que comunica entre ambas partes. El usuario utiliza la funcionalidad del controlador para manipular el modelo, y éste actualiza la interfaz mediante la información del modelo.

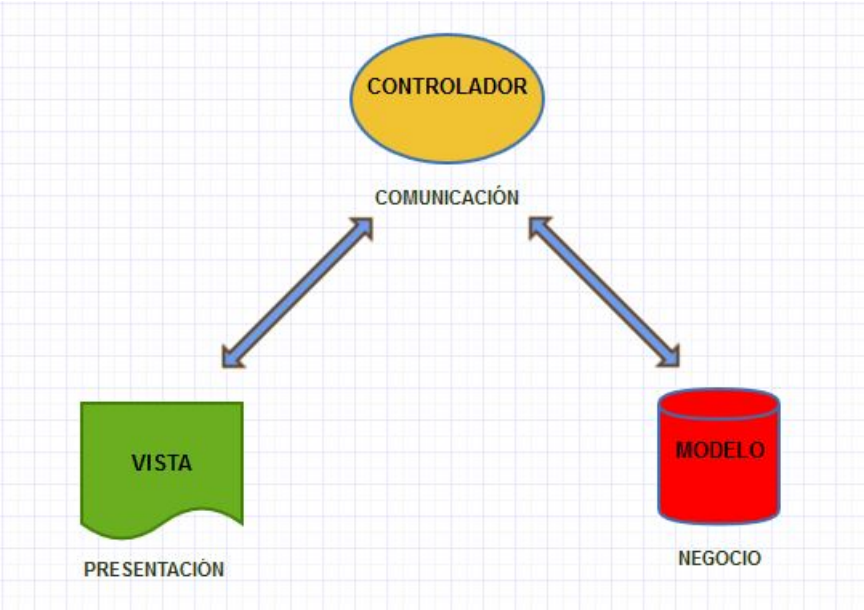


Figura 3.1: Patrón MVC

Este patrón se ha utilizado de forma muy común para el desarrollo de la parte servidor, especialmente en las aplicaciones Java EE clásicas: el cliente realiza peticiones al servidor y éste, utilizando un patrón MVC, generaba un HTML que era devuelto como respuesta al cliente. Siguiendo este sistema, la mayor carga de trabajo reside en el lado del servidor.

En la aplicación que ocupa este proyecto, utilizaremos una evolución de este patrón conocido como SPA (Single-Page-Application), cuyo principal objetivo es mover parte de la carga de trabajo desde el servidor al cliente. Para ello, el lado del cliente tiene su propio patrón MVC con el que tratar la información del cliente.

La principal diferencia es que el cliente recibe el contenido HTML sólo una vez desde el servidor. El resto de peticiones se conformarán de cadenas de datos en formato Json que el controlador del cliente usará para modificar la interfaz del usuario. Como esto se realiza en el lado del cliente, se libera de este trabajo al servidor.

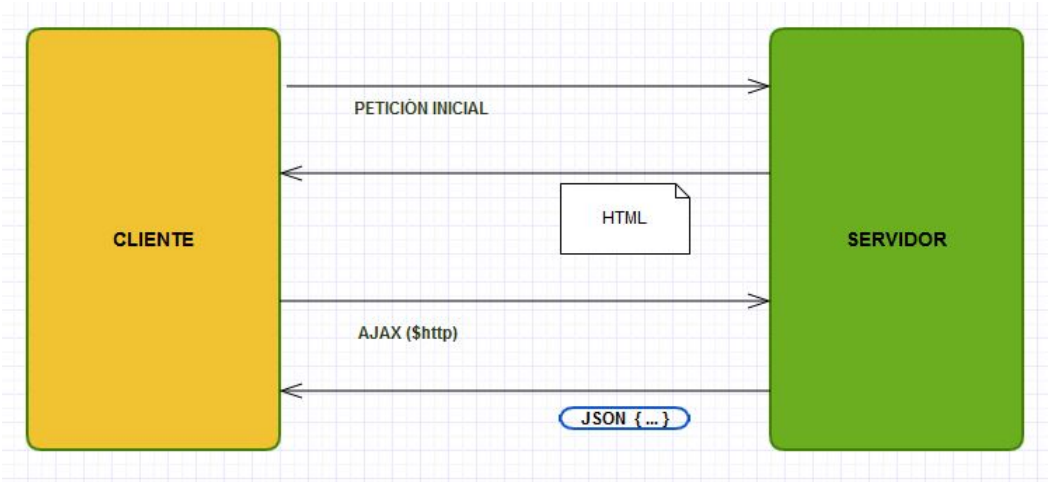


Figura 3.2: Patrón SPA

En el cliente únicamente se carga una página. Esta página se compone de varios fragmentos que modificarán la vista durante la interacción del cliente y se actualizarán dinámicamente con los datos recibidos del servidor sin la necesidad de recargar completamente la página, lo que otorga al usuario una mayor sensación de fluidez que con un sistema mas clásico.

Para implementar este patrón de arquitectura utilizaremos unas de las tecnologías que mas fuerza han conseguido durante los últimos años entre los desarrolladores: Angular JS para la parte del cliente, y Spring MVC para la parte del servidor. En los siguientes capítulos detallaremos el funcionamiento de estas tecnologías.

CAPÍTULO 4
Angular JS

Para el desarrollo de la parte del cliente se utiliza el Framework de Javascript Angular JS v.1.5.8. Se trata de un conjunto de librerías escritas en Javascript que permiten la modificación de las vistas de manera dinámica y sencilla de programar. Para utilizar Angular sólo hay que cargar el script en la página.

Angular añade directivas que se utilizan como atributos HTML, además de expresiones que se vinculan con los datos del modelo y que se actualizan en la vista de manera dinámica al modificarse éstos mediante código Javascript.

```
// MODELO - Objeto Javascript. Se corresponde con el modelo de la clase Java del servidor
self.alumno = {login, password, nombre, apellidos, telefono, email, direccion, habilitado}
```

Figura 4.1: Angular MVC. Ejemplo de modelo

```
<span ng-bind="alumno.login"></span>
<span ng-bind="alumno.nombre"></span
<span ng-bind="alumno.apellidos"></s
<span ng-bind="alumno.email"></span>
```

Figura 4.2: Angular MVC. Ejemplo de vista

```
// CONTROLADOR - Código Javascript
function cargar_alumno(){
    usuario_service.obtenerAlumno(Alumno.getId()).then(
            function(response) {
                $scope.alumno = response;
            },
            function(errResponse){
                console.log(errResponse);
            }
    );
}
```

Figura 4.3: Angular MVC. Ejemplo de controlador

4.1 Vistas

4.1.1. Directivas

A continuación se listan las directivas utilizadas en el proyecto, explicando su utilidad.

ng-app

Inicializa la aplicación Angular. Todo el contenido que se incluya dentro del elemento que contenga esta directiva formará parte de la aplicación.

ng-view

Dependiente del servicio de $route. En este servicio se establecen las rutas que redirigen a los diversos fragmentos de la página. Estos fragmentos de la vista se cargarán dentro del elemento HTML que contenga esta directiva.

ng-controller

Indica el nombre del controlador de la aplicación que se ejecutará al cargarse la vista. El controlador contiene la funcionalidad Javascript para el manejo de las variables de la aplicación.

Para el desarrollo de esta aplicación utilizamos una alternativa mas dinámica que lo que permite esta directiva. Se trata de cargar el controlador desde el servicio de $route, indicando según la ruta recibida que vista cargar con que controlador. Así se permite reutilizar una vista que, usando distintos controladores, puede tener diferente funcionalidad.

ng-model

Vincula el valor del elemento en que se añade con la variable del modelo del objeto Javascript. Es bidireccional, esto es, además de reflejar el valor del modelo en la vista, se puede acceder al valor introducido por el usuario en la vista mediante Javascript. Muy útil para formularios.

ng-bind

Muestra el valor del campo del modelo que se indica en el elemento en que se añade la directiva. Es equivalente a escribir directamente en la vista el campo del modelo entre llaves dobles: (alumno.nombre).

ng-repeat

Indicándole un listado de objetos Javascript, repite el elemento al que se añade la directiva por cada uno de los elementos del listado, enlazando los datos de cada objeto a uno de los elementos repetidos.

ng-click

Indica el código o la función que se debe ejecutar al hacer click en elemento que contiene la directiva.

ng-class

Enlaza el elemento con una clase CSS de manera dinámica. El valor de la directiva puede ser un objeto que cambie de valor, haciendo que inmediatamente cargue una clase distinta.

ng-submit

Captura el submit de un formulario HTML, ejecutando el código indicado y evitando que el formulario recargue la página. Se puede establecer para que no interrumpa la recarga de la página por el submit, pero esto rompe la promesa de la SPA.

ng-show

Sólo muestra el elemento si se cumple la condición que se especifique en su valor.

ng-hide

Oculta el elemento si se cumple la condición especificada.

ng-disabled

Deshabilita un elemento (un botón) si se cumple la condicion especificada.

ng-cloak

Oculta los elementos que contiene hasta que la aplicación Angular ha terminado de cargar. Con esto se evita quese puedan mostrar elementos en crudo antes de haber sido evaluados.

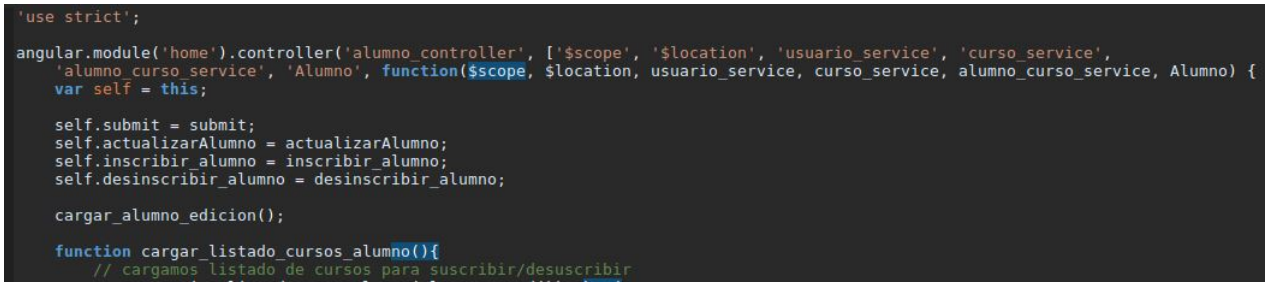
4.2 Controladores

Los controladores implementan la lógica de la aplicación, actualizan las variables de la vista con los datos del modelo e invocan los servicios de la aplicación.

Los métodos y variables invocados desde la vista deben de estar definidos en el controlador que se cargue con ella (indicado mediante la directiva ng-Controller o, como en nuestro caso, desde el servicio $route)

4.2.1. $scope

Se trata de un objeto especial de AngularJS. Es la forma de comunicar la vista con el controlador utilizando la funcionalidad de AngularJS. Asignándole todo tipo de variables, ya sean Strings, objetos Javascript o incluso funciones Javascript, se pueden acceder desde la vista y ésta se actualizará de inmediato de manera dinámica.



```
'use strict';

angular.module('home').controller('alumno_controller', ['$scope', '$location', 'usuario_service', 'curso_service',
    'alumno_curso_service', 'Alumno', function($scope, $location, usuario_service, curso_service, alumno_curso_service, Alumno) {
    var self = this;

    self.submit = submit;
    self.actualizarAlumno = actualizarAlumno;
    self.inscribir_alumno = inscribir_alumno;
    self.desinscribir_alumno = desinscribir_alumno;

    cargar_alumno_edicion();

    function cargar_listado_cursos_alumno(){
        // cargamos listado de cursos para suscribir/desuscribir
```

Figura 4.4: Ejemplo de declaración de un controlador en el proyecto

4.3 Servicios

Los servicios en Angular son objetos Javascript que contienen parte de la lógica de la aplicación. Son independientes de las vistas, se invocan desde los controladores y contienen funcionalidad reutilizable.

Los servicios son objetos Singleton, lo que significa que la aplicación los construye una vez y se accede al mismo objeto desde cualquier controlador que lo invoque.

Esta propiedad se utiliza en el proyecto para compartir información entre controladores. Por ejemplo para saber que usuario se ha seleccionado anteriormente, o que ejercicio debe cargar la vista de edición de ejercicio tras haber pulsado en el botón de 'editar ejercicio' de una vista diferente.

```
'use strict';

angular.module('home').factory('alumno_service', ['$http', '$q', 'REST_SERVICE_URI', function($http, $q, REST_SERVICE_URI){

    var REST_SERVICE_URI_ADMIN = REST_SERVICE_URI.admin;
    var REST_SERVICE_URI_AUTH = REST_SERVICE_URI.auth;

    var factory = {
        listadoAlumnos: listadoAlumnos,
        crearAlumno: crearAlumno,
    };

    return factory;

    function listadoAlumnos() {
```

Figura 4.5: Ejemplo de declaración de un servicio en el proyecto

Existen cinco tipos de servicios. Del mas sencillo a mas complejo, van ofreciendo una funcionalidad más extendida y compleja.

4.3.1. Constant

Es el servicio más sencillo de todos. Contiene valores constantes de cualquier tipo que no se pueden modificar.

4.3.2. Value

Además de lo anterior, permite ser utilizado en un módulo de configuración inicial para crear otras variables, evitando tener que crear para cada módulo variables globales con el mismo valor.

4.3.3. Service

Este tipo de servicio se compone de una clase Javascript. Al invocarse devuelve el constructor de la clase. Puede contener variables y funciones.

4.3.4. Factory

Con la misma funcionalidad que el anterior, la factoría devuelve un objeto con la funcionalidad definida. Esto permitiría añadir código propio en la factoría antes de crear el objeto a devolver, mientras que en el servicio se ejecuta únicamente el código de Angular para crear el servicio.

4.3.5. Provider

Con la funcionalidad del servicio Factory, pero permite modificar su configuración inicial antes de ejecutarse la aplicación.

4.4 $route

Este servicio permite capturas las rutas y utilizarlas para cargar el fragmento correspondiente en la vista con el controlador especificado.

La mayor fluidez y velocidad al cargar únicamente un fragmento del HTML de la página en lugar de la página entera, incluyendo CSS y scripts es la principal ventaja de las SPA.

```
angular.module('home').config(['$routeProvider', function($routeProvider) {
   $routeProvider.

   when('/login', {
      templateUrl: 'resources/views/login.jsp'
   }).

   when('/crear_tema', {
          templateUrl: 'resources/views/crear_tema.jsp',
          controller: 'tema_controller',
          controllerAs: 'tema_ctrl'
   }).

   when('/crear_curso', {
          templateUrl: 'resources/views/crear_curso.jsp',
          controller: 'curso_controller',
          controllerAs: 'curso_ctrl'
   }).
```

Figura 4.6: Rutas definidas que cargarán el fragmento de la vista correspondiente con su controlador.

4.5 $http

Es el servicio proporcionado por Angular para realizar peticiones AJAX a servidores HTTP remotos. Tras realizar la petición, espera una respuesta. Permite definir funciones callback en caso de que la respuesta sea correcta o haya habido algún error.

Permite las peticiones clásicas de operaciones CRUD GET, POST, PUT y DELETE entre otras.

```
function obtenerAlumno(id_alumno) {
    var deferred = $q.defer();
    $http.get(REST_SERVICE_URI_AUTH + 'alumno/' + id_alumno + '/')
        .then(
        function (response) {
            deferred.resolve(response.data);
        },
        function(errResponse){
            console.error('Error en el servidor. No se ha podido obtener el alumno.');
            deferred.reject(errResponse);
        }
    );
    return deferred.promise;
}
```

Figura 4.7: Petición POST al servidor mediante el servicio $http

4.6 $q

Es un tipo de objeto especial que actúa como una 'promesa'. Esto es, una variable de la que aún no se conoce el valor, pero no podemos dejar el servicio bloqueado a la espera de obtenerlo, por lo que se utiliza esta variable, que en el futuro contendrá un valor.

Un ejemplo sencillo se muestra en la figura anterior. Antes de realizar la petición POST, se crea la variable promesa, se lanza la petición y se devuelve la promesa. La promesa aun no contendrá una respuesta de la petición, pero se impide que la petición deje colgado el servicio a la espera de su resolución.

Una vez terminada la petición, se ejecuta una de las funciones callback. Si ha ido bien, se establece en esta variable el valor de la respuesta (resolve), si la respuesta ha sido de error, se devuelve una respuesta de error (reject), que indica que la promesa no ha podido obtener el resultado. En ambos casos esta respuesta deberá ser gestionada por el invocador de la función.

4.7 Otras herramientas

4.7.1. textAngular

Un completo editor de texto que se puede asociar a una variable de Angular de tipo texto. El texto enriquecido se almacenará en formato HTML y se mostrará renderizado en la vista. En el proyecto se utiliza para generar o mostrar el contenido teórico de los temas.

4.7.2. ui.bootstrap

Añade componentes de Bootstrap para Angular como menús en acordeón, alertas personalizadas, colapso de elementos o calendarios para inputs.

4.7.3. ngMaterial

Permite añadir pestañas configurables en las vistas para poder mostrar la información mas ordenada, entre otras funcionalidades.

CAPÍTULO 5
Spring MVC

Para el desarrollo de la parte del servidor se utiliza el Framework de Java Spring MVC 4.

Spring se basa en un servlet central (DispatcherServlet) que se encarga de despachar cada petición de cliente que recibe al controlador apropiado. El controlador ejecuta su funcionalidad y le devuelve una respuesta que el servlet se encargará de retornar al cliente.

En este proyecto la comunicación entre cliente y servidor se basa en cadenas de carácteres en formato Json, que compondrán los distintos valores y objetos que se transmiten entre cliente y servidor.

5.1 Conceptos

5.1.1. Spring Beans

Son los objetos Java que conforman la aplicación. Se definen en el fichero de configuración de Spring Servlet y se corresponden con una clase definida en el proyecto. Son instanciados y gestionados por Spring.

5.1.2. Contenedor de beans

En donde están contenidos los beans de la aplicación. Los beans son creados y gestionados por el contenedor. Éstos son accesibles desde cualquier parte de la aplicación.

5.1.3. Inyección de dependencias

Una característica de Spring es la gestión de los objetos por el contenedor de beans. En lugar de que cada clase tenga que instanciar los objetos de las clases de que dependa, es Spring quien crea estos objetos y se los inyecta a la clase que los necesite. Por lo tanto es el contenedor de beans el que instancia los objetos y establece las dependencias entre ellos. Esto nos permite conseguir un código menos acoplado y más fácil de modificar.

5.1.4. Anotaciones

Una forma cómoda de enlazar los beans. Para indicar a Spring que debe utilizar las anotaciones hay que añadir en el fichero de configuración de Spring la línea:

```
<context:annotation-config/>
```

El siguiente es un listado con las anotaciones de las que haremos uso en la aplicación.

@autowired

Indica a una variable o a un método que debe de estar conectado con la inyección de dependencias de Spring. En el proyecto se utiliza para poder utilizar objetos de una clase desde otra clase dependiente

@Component

Indica que la clase anotada es un 'componente'. Esto significa que será detectada automáticamente por Spring. Las anotaciones @Controller, @Service y @Repository son especializaciones de esta anotación, cada una asignada a una de las capas de la aplicación

@Controller

Para las clases de controlador. Pertenece a la capa de presentación.

@Repository

Indica que la clase es un repositorio, es decir, pertenece a la capa de persistencia (Data Access Object).

@Service

Indica que la clase con la anotación es un servicio. Con esta anotación Spring la detectará automáticamente.

@Transactional

Necesario en los métodos que hagan operaciones en base de datos. Si ocurre algún error durante la ejecución del método se hace un rollback de todos los cambios, si ha ido bien, se hace un commit con los cambios.

@Entity

Indica que la clase con la anotación es una entidad (una clase persistente, los objetos se almacenan en base de datos).

@Table

Indica el nombre de la tabla de la base de datos con la que se corresponde la entidad

@Column

Indica, para cada elemento de la entidad, a que columna se corresponde de la tabla en base de datos.

@Id

Sirve para indicar que campo de la entidad es el identificador (valor único en la tabla de la base de datos).

@GeneratedValue

Indica la estrategia a seguir a la hora de generar los valores del identificador. Con AUTO se indica a Spring que asigne uno por defecto. La estrategia variará en las clases con herencia dependiendo del tipo de estrategia de herencia establecida.

@Inheritance

Indica la estrategia a seguir para la herencia entre clases. Existen tres tipos de estrategias:

@PrimaryKeyJoinColumn

Para las clases que heredan de un padre, utilizamos esta etiqueta para indicar que atributo del padre se utilizará para identificar que registro del hijo corresponde con que registro del padre. Utilizamos el atributo identificador (id).

- SINGLE_TABLE. En Java tenemos tres objetos, pero en base de datos sólo existe una tabla, que contendrá columnas para todas las variables de la clase original y de todas las que heredan.

- JOINED. Una tabla por cada una de las clases, Una para la original con los campos que sean comunes en todas ellas, y otra por cada una de las clases que heredan, únicamente con los atributos propios y el identificador de la fila que le corresponde de la tabla principal.

- TABLE_PER_CLASS. Una tabla independiente para cada una de las clases. Esto hace que los atributos que comparta un hijo con el padre tienen que estar repetidos. Es la opción menos aconsejable por su peor rendimiento.

@RequestMapping

Contiene como valor el mapeo con la ruta especificada en la petición recibida por el cliente. La función anotada cuyo mapeo coincida se ejecutará.

@ResponseBody

Indica que la respuesta del método anotado será el body de la respuesta de la petición recibida del cliente. Si la respuesta es un objeto Java Spring lo transformará en una respuesta aceptable por una aplicación REST. En nuestro caso todas las comunicaciones serán arrays Json.

@PathVariable

Si la petición contiene una variable en la ruta (una petición GET o DELETE), con esta etiqueta podemos recuperarla e indicar que tipo de variable es. Un caso muy común es enviar el ID del objeto a recuperar en la petición GET y que la función del controlador retorne al cliente el objeto almacenado en base de datos en formato Json.

@RequestBody

Recupera el contenido de una petición (POST o PUT), lo serializa y lo asigna a una variable para la función del controlador.

@EnableWebSecurity

Se utiliza para extender o sobreescribir los métodos de Spring Security. Extendiendo la clase de WebSecurityConfigurerAdapter podremos añadir nuestra propia configuración de seguridad.

5.1.5. Controladores

En Spring, la capa de presentación se compone del controlador y la vista. El controlador maneja las peticiones del cliente e invoca los servicios definidos en la aplicación.

Al recibir el servidor una petición, el servlet DispatcherServlet intentará redirigir la petición a uno de los controladores dependiendo del tipo de petición y de la ruta de la petición. Se ejecutará la función correspondiente y retornará una respuesta. En nuestra aplicación los controladores únicamente retornan información en formato Json, no vistas generadas, dada la composición del cliente.

5.1.6. Servicios

Los servicios contienen la mayor parte de la lógica de la aplicación. Se invocan por los controladores y se encarga de las peticiones a los objetos DAO de la capa de persistencia.

5.1.7. Repositorios

En la capa de persistencia tenemos los objetos DAO. Éstos se encargan del acceso y manipulación de las bases de datos mediante las operaciones CRUD.

5.1.8. Modelos

Los modelos son las clases que definen los objetos Java. Contienen los atributos de éstos y se mapean con las tablas y colmunas de la base de datos.

5.1.9. Spring Security

Se trata de un framework propio de Spring para gestionar la seguridad, la autenticación y proporcionar autorización en el acceso a la funcionalidad del servidor. Es muy configurable y relativamente sencillo de instalar. La principal característica respecto a

otras herramientas es su facilidad de exportación (la configuración se incluye en el fichero .WAR).

CAPÍTULO 6
Hibernate

Usamos el framework Hibernate como ORM (herramienta de mapeo objeto-relacional) para la manipulación directa de la base de datos. Está completamente integrado con Spring, y se utiliza para los accesos a la base de datos desde la capa de persistencia de la aplicación.

El mapeo se los atributos de las clases Java y de las tablas de la base de datos se puede realizar de dos maneras: con un fichero XML donde se indican las correspondencias, o el sistema de anotaciones explicado anteriormente. Utilizamos este último por ser más novedoso y sencillo de utilizar.

El resto de configuración de Hibernate la añadiremos en el fichero de configuración de Spring Servlet.

6.1 SessionFactory

Se trata de una interfaz que conecta la aplicación Java e Hibernate. Cuando se crea el objeto SessionFactory, éste incluye toda la configuración establecida para Hibernate y permite utilizar sus métodos de ORM para realizar las operaciones sobre la base de datos.

6.1.1. Operaciones CRUD

Las operaciones CRUD se realizan del siguiente modo:

CREATE

Se realiza con el atributo *insert* de SessionFactory. Insertará un nuevo registro en la tabla de base de datos o retornará una excepción. Siempre es conveniente realizar las comprobaciones necesarias antes de hacer una consulta a la base de datos para tener controlados los posibles errores en el mayor grado posible.

READ

Podemos obtener un registro existente en la tabla de base de datos con el atributo *get* de SessionFactory. Retornará una excepción en caso de error.

UPDATE

Mediante el atributo *update* de SessionFactory, actualizará un registro de la tabla de base de datos o retornará una excepción.

DELETE

Con el atributo *delete* de SessionFactory. Eliminará un registro de la tabla de base de datos o retornará una excepción.

Además de estas operaciones básicas, tiene otras funcionalidades colo *load*, que sólo comprueba si un registro existe, en lugar de obtener todo el registro. También se permite crear queries con *createQuery*.

CAPÍTULO 7
Implementación

En este capítulo vamos a desarrollar cómo se ha llevado a cabo la implementación de la aplicación, que procesos se han seguido y cómo se ha llevado a cabo el proyecto.

El desarrollo del proyecto se ha realizado con una máquina Packard Bell MX52, sistema operativo Ubuntu 14.04 LTS. Entorno de desarrollo Eclipse Neon y Java 1.8.0_111. Para el control de versiones se ha utilizado GIT, haciendo cada nuevo desarrollo en una nueva rama para facilitar el cambio entre tareas.

Para poner en marcha el servidor se ha utilizado Apache Tomcat v8.5.

7.1 MySql

Para el almacenamiento de información se ha utilizado una base de datos MySql 5.7. Contendrá toda la información relativa a los usuarios, cursos y temario o evaluaciones.

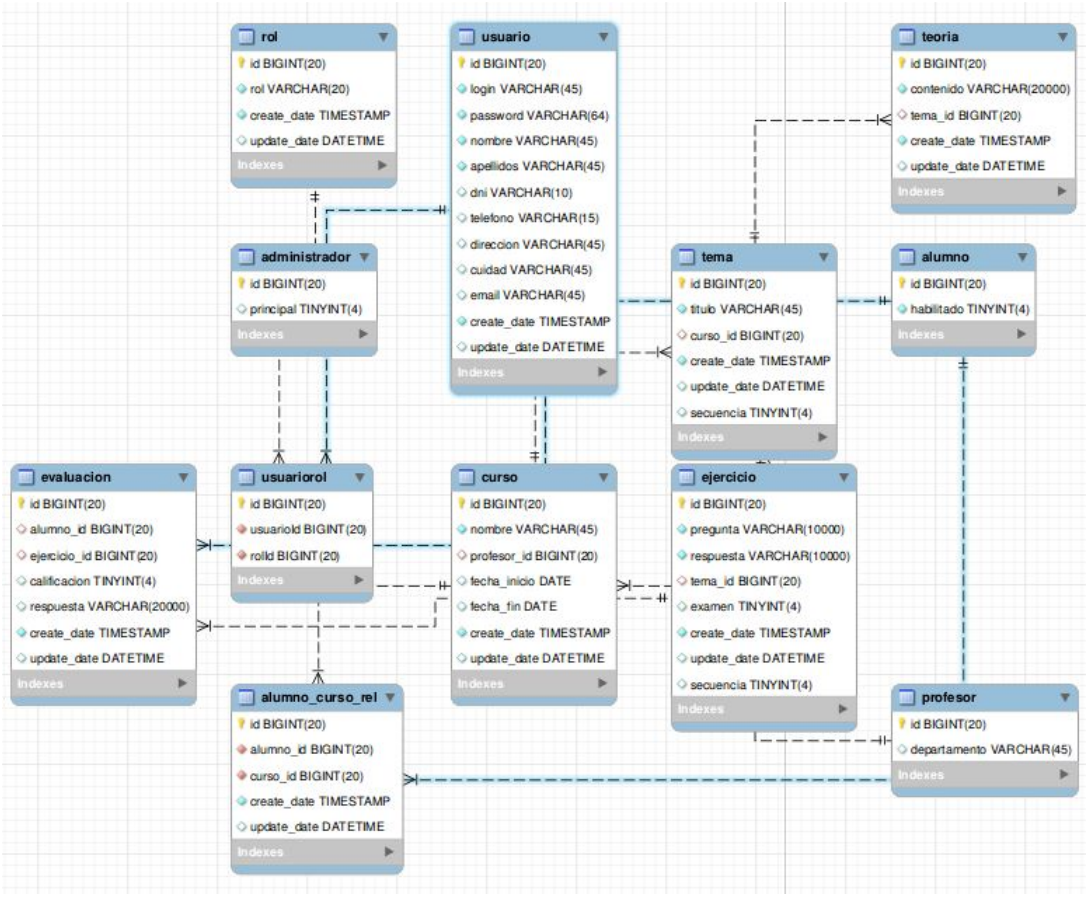


Figura 7.1: Diseño de la base de datos de la aplicación.

La base de datos se ha generado con el código SQL del anexo. Para el seguimiento y control de ésta se ha utilizado MySql Workbench 6.3.

7.2 Spring

El proyecto se ha desarrollado con Spring MVC 4. Ahora explicaremos cómo se ha generado el proyecto utilizando Maven y cómo se ha llevado a cabo la parte de servidor.

7.2.1. Maven

Maven es una herramienta para crear y gestionar todo tipo de proyectos Java. Lo utilizamos como un plug-in de Eclipse para generar el nuevo proyecto. Una vez generado hay que configurar el fichero POM (project object model), un XML donde se deben añadir todas las dependencias del proyecto utilizando las versiones adecuadas. Una vez validado, Maven descargará todas las librerías JAR necesarias. Hecho esto, ya tenemos una base con la que empezar a trabajar.

7.2.2. Modelos

Primero definimos los modelos en Java, que se corresponden con las tablas de la base de datos. Añadimos todos los atributos a cada clase, incluyendo Getters y Setters, las anotaciones para Hibernate indicando que tabla corresponde a cada clase con @Table y que columnas corresponden a cada atributo de la clase con @Column.

Para la herencia de clases utilizamos la estrategia JOINED, esto es, una tabla por cada una de las clases, pero la tabla padre contendrá los atributos que comparta con los hijos, mientras que los hijos únicamente tendrán sus atributos propios.

Esto lo utilizamos para los usuarios del sistema. La clase Usuario es la principal, que contiene la mayor parte de los campos. De ésta heredan las clases Alumno y Administrador. La clase Profesor hereda de Alumno, por lo que contendrá los atributos de la clase Usuario y Alumno.

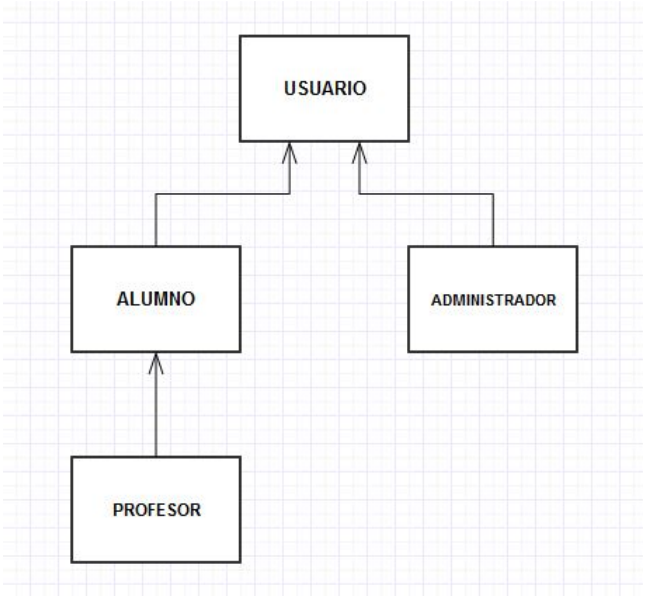


Figura 7.2: Herencia entre usuarios del sistema.

7.2.3. Persistencia

Creamos las clases de los DAO. Aquí se incluyen las operaciones sobre la base de datos usando el ORM de Hibernate. Definimos las funciones para las operaciones CRUD.

```
@Transactional
public List<Ejercicio> obtenerEjerciciosTema(long tema_id, boolean examen) {
    ejercicios = ejercicioDao.obtenerEjercicios(tema_id, examen);
    return ejercicios;
}

@Transactional
public void delete(long id) {
    ejercicioDao.delete(id);
}

@Transactional
public Ejercicio get(long id) {
    return ejercicioDao.get(id);
}
```

Figura 7.3: Ejemplos de operaciones con Hibernate para el DAO del modelo de Ejercicio.

```java
@Override
public void update(Ejercicio ejercicio) {

    java.text.SimpleDateFormat sdf = new java.text.SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String currentTime = sdf.format(new Date());
    ejercicio.setUpdate_date(currentTime);

    Session session = this.sessionFactory.getCurrentSession();
    session.update(ejercicio);

}
```

Figura 7.4: Ejemplo de una operación UPDATE utilizando Hibernate.

7.2.4. Servicios

En esta parte incluímos la lógica de la aplicación. Su funcionalidad será invocada desde los controladores y se encargará de llamar a la capa de persistencia cuando sea necesario.

Por ejemplo, si un profesor quiere borrar un tema de un curso, además de eliminarse el registro de la tabla *tema* de la base de datos, la lógica de la aplicación debe de encargarse de eliminar todo el contenido que dependa del tema, para evitar que quede contenido que no será accesible en la base de datos. En este caso, se eliminarán los registros de teoría y ejercicios correspondientes a este tema.

```java
@Transactional
public void delete(long id) {

    // eliminamos la teoria relacionada
    Teoria teoria = teoriaService.get_tema(id);
    if(teoria != null){
        teoriaService.delete(teoria.getId());
    }

    // eliminamos los ejercicios relacionados
    List <Ejercicio> ejercicios_tema = ejercicioService.obtenerEjerciciosTema(id, false);
    for(Ejercicio ejercicio : ejercicios_tema) {
        ejercicioService.delete(ejercicio.getId());
    }

    // eliminamos los ejercicios de examen relacionados (por separado por posible futura modificación)
    List <Ejercicio> ejercicios_examen_tema = ejercicioService.obtenerEjerciciosTema(id, true);
    for(Ejercicio ejercicio : ejercicios_examen_tema) {
        ejercicioService.delete(ejercicio.getId());
    }

    // eliminadas dependencias, podemos borrar el tema
    temaDao.delete(id);
}
```

Figura 7.5: Servicio para el borrado de un tema.

7.2.5. Controladores

Con los controladores gestionamos las peticiones redirigidas por el Servlet de Spring, y ofrecemos la respuesta al cliente.

En la siguiente figura tenemos dos ejemplos de cómo gestiona un controlador una petición. Una función para obtener el listado de temas pertenecientes a un curso y otra para crear un nuevo tema. Se invocan a los servicios correspondientes para llevar a cabo las operaciones.

```java
// OBTENER LISTADO DE TEMAS USUARIO
@RequestMapping(value = "/resource/temas/{curso_id}", method = RequestMethod.GET, produces = "application/json")
public @ResponseBody List<Tema> obtenerListadoTemas(@PathVariable long curso_id) throws JsonProcessingException {

    try{
        List<Tema> listadoTemas = temaService.obtenerTemasCurso(curso_id);
        return listadoTemas;

    } catch (Exception e) {
        System.out.println(e.getMessage());
        return null;
    }
}

// CREAR TEMA
@RequestMapping(value = "/auth/tema/", method = RequestMethod.POST)
public ResponseEntity<Tema> crearTema(@RequestBody Tema tema) {

    try{
        temaService.insert(tema);
    } catch (Exception e) {
        System.out.println(e.getMessage());
        return null;
    }

    // al crear un tema, se crea la teoría y se asocia
    Teoria teoria = new Teoria();
    teoria.setTema_id(tema.getId());
    teoriaService.insert(teoria);

    return new ResponseEntity<Tema>(tema, HttpStatus.CREATED);
  }
```

Figura 7.6: Funciones del controlador

La URL de la petición que llega del cliente debe seguir el siguiente modelo:

Para el listado de temas de un curso, la petición que llega del cliente, una vez pasado el sistema de autenticación, debe de tratarse de una petición GET Y seguir el siguiente modelo:

http://miservidor/proyecto_app/resource/temas/id_curso

Para la creación del tema, debe de tratarse de una petición POST y seguir el patrón adecuado:

http://miservidor/proyecto_app/auth/tema/

Si el Servlet no encuentra un controlador adecuado para gestionar la petición, retornará un error 404 de *Recurso no encontrado*.

7.2.6. Autenticación

En cualquier tipo de aplicación se hace necesaria una gestión de la seguridad y de acceso a los recursos disponibles. En esta aplicación utilizamos tres niveles, que se corresponden con los roles de los usuarios: el de alumno, que sólo podrá acceder a los recursos disponibles en cuya ruta se incluya el patrón */resource/*; el de profesor, que además del anterior podrá acceder a los recursos con el patrón */auth/*; y finalmente el de administración, que podrá acceder, además de los anteriores, a los recursos que incluyan el patrón */admin/*.

Para ello definimos los roles de usuario. Cada usuario podrá tener uno o varios roles. Para permitir una futura ampliación del sistema de roles, hemos realizado la siguiente implementación.

Tenemos una tabla ROL con los roles existentes. Estos son ALUMNO, PROFESOR y ADMINISTRADOR. Para indicar el rol de un usuario creamos una tabla intermedia relacional entre USUARIO y ROL. Esta tabla contendrá en cada registro el ID del usuario

en cuestión y el ID del rol que tiene. Si el usuario tiene varios roles, tendrá varios registros en la tabla, uno por cada rol asignado.

Para lograr la correcta autenticación usando Spring Security, debemos implementar una extensión de la clase *WebSecurityConfigurerAdapter* de Spring Security. Las funciones añadidas son las que siguen:

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http.csrf().disable()
    .authorizeRequests()
        .antMatchers("/admin/**").access("hasRole('ROLE_ADMINISTRADOR')")
        .antMatchers("/auth/**").access("hasRole('ROLE_ADMINISTRADOR') or hasRole('ROLE_PROFESOR')")
        .antMatchers("/resource/**").authenticated()
        .antMatchers("/resources/**").permitAll()  // librerias js y css
    .and().formLogin()
        .loginPage("/login")
        .permitAll();

}
```

Figura 7.7: Configuración de acceso a los recursos

Aquí se asignan los usuarios con que roles pueden acceder a que recursos. Como los recursos del alumno son accesibles por todos los tipos de usuario, simplemente con que el usuario esté autenticado ya le dejamos acceder.

```
@Bean
public CustomBasicAuthenticationEntryPoint getBasicAuthEntryPoint(){
    return new CustomBasicAuthenticationEntryPoint();
}


@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService);
}

@Bean(name = BeanIds.AUTHENTICATION_MANAGER)
@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}
```

Figura 7.8: Llamada a la función para obtener información sobre los roles del usuario.

Para obtener la información sobre los roles de un usuario necesitamos un desarrollo propio que localice los roles desde nuestra base de datos. Para ello debemos extender la clase *UserDetailsService* de Spring.

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    Usuario usuario = usuarioService.get(username);

    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    List<Rol> roles = usuarioRolService.getRoles(usuario.getId());

    for(Rol rol : roles){
        grantedAuthorities.add(new SimpleGrantedAuthority(rol.getRol()));  // TODO
    }

    return new org.springframework.security.core.userdetails.User(usuario.getLogin(), usuario.getPassword(), grantedAuthorities);
}
```

Figura 7.9: Obtención de los roles del usuario

El método extendido de Spring Security invocará a nuestro servicio creado para la obtención de roles del usuario.

```java
@Transactional
public List<Rol> getRoles(long id) {
    List<Rol> roles_obj = usuarioRolDao.getRoles(id);
    return roles_obj;
}
```

Figura 7.10: Service para la obtención de roles.

El service se encarga de invocar al DAO para obtener los roles de la base de datos.

```java
@Override
public List<Rol> getRoles(long id) {
    Session session = this.sessionFactory.getCurrentSession();

    String hql = "from UsuarioRol where usuarioId = :usuarioId";
    Query query = session.createQuery(hql);
    query.setParameter("usuarioId", id);
    List<UsuarioRol> usuarioRoles = query.list();


    List<Rol> roles = new ArrayList<Rol>();
    List<Rol> rolObtenido = null;

    for(UsuarioRol usuarioRol : usuarioRoles){
        hql = "from Rol where id = :id";
        query = session.createQuery(hql);
        query.setParameter("id", usuarioRol.getRolId());
        rolObtenido = (List<Rol>)query.list();
        if (!rolObtenido.isEmpty()){
            roles.add((Rol)rolObtenido.get(0)); // id unico para cada rol
        }

    }

    return roles;
}
```

Figura 7.11: Repositorio para obtener los roles de un usuario.

Dede el DAO realizamos la consulta siguiendo la lógica establecida para las tablas explicada anteriormente.

Ésta es la configuración principal para el manejo de roles y permisos de nuestra aplicación.

7.2.7. Hibernate

Utilizamos la versión de Hibernate 4.3.5.Final. Simplemente incluyéndola en el fichero POM se descargarán las librerías de Java necesarias.

Además de las anotaciones en los modelos de las clases, Hibernate requiere una configuración para poder acceder a la base de datos, localizar los beans definidos. Esta configuración la añadiremos en el XML de Spring Servlet:

```xml
<beans:bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <beans:property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <beans:property name="url"
        value="jdbc:mysql://localhost:3306/esquema_aplicacion" />
    <beans:property name="username" value="root" />
    <beans:property name="password" value="myPass" />
</beans:bean>
```

Figura 7.12: DataSource.

En esta definición se indica el driver de MySql que utilizará Hibernate para conectar con la base de datos, su localización y los permisos de acceso a la base de datos.

```
<!-- Hibernate 4 SessionFactory Bean definition -->
<beans:bean id="hibernate4AnnotatedSessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <beans:property name="dataSource" ref="dataSource" />
    <beans:property name="annotatedClasses">
        <beans:list>
            <!-- <beans:value>com.proyecto_app.model.Usuario</beans:value> -->
            <beans:value>com.proyecto_app.model.Rol</beans:value>
            <beans:value>com.proyecto_app.model.UsuarioRol</beans:value>
            <beans:value>com.proyecto_app.model.Curso</beans:value>
            <beans:value>com.proyecto_app.model.Tema</beans:value>
            <beans:value>com.proyecto_app.model.Teoria</beans:value>
            <beans:value>com.proyecto_app.model.Examen</beans:value>
            <beans:value>com.proyecto_app.model.Ejercicio</beans:value>
            <beans:value>com.proyecto_app.model.Alumno</beans:value>
            <beans:value>com.proyecto_app.model.Profesor</beans:value>
            <beans:value>com.proyecto_app.model.Admin</beans:value>
            <beans:value>com.proyecto_app.model.AlumnoCurso</beans:value>
            <beans:value>com.proyecto_app.model.Evaluacion</beans:value>
        </beans:list>
    </beans:property>
    <beans:property name="hibernateProperties">
        <beans:props>
            <beans:prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</beans:prop>
            <beans:prop key="hibernate.show_sql">true</beans:prop>
        </beans:props>
    </beans:property>
</beans:bean>
```

Figura 7.13: Beans.

En el fichero de configuración se deben añadir las clases anotadas para que pueda localizarlas.

```
<util:properties id="databaseProperties">
    <beans:prop key="hibernate.hbm2ddl.auto">update</beans:prop>
    <beans:prop key="hibernate.id.new_generator_mappings">true</beans:prop>
</util:properties>


<tx:annotation-driven transaction-manager="transactionManager" />

<beans:bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <beans:property name="sessionFactory"
        ref="hibernate4AnnotatedSessionFactory" />
</beans:bean>
```

Figura 7.14: Properties.

Configuración sobre el uso de la base de datos para SessionFactory.

La primera indica que el esquema de la base de datos se debe actualizar cuando se crea la SessionFactory. La segunda línea indica a Hibernate que debe usar la estrategia especificada en las anotaciones @GeneratedValue.

7.3 AngularJS

Con angular generaremos las vistas de la interfaz de la aplicación. Los controladores se encargarán del manejo de eventos y la manipulación del $scope, además de invocar a los servicios que gestionarán las variables compartidas entre los controladores y realizarán las peticiones al servidor.

7.3.1. Vistas

La vista principal se incluye en el fichero index.jsp. Incluye los scripts de Angular, Bootstrap, JQuery y las demás herramientas que utilizamos en la aplicación. También se incluyen los Javascript que contienen los controladores, servicios y herramientas propias de la aplicación. Además contiene el DIV principal donde se cargarán los fragmentos de las demás vistas, que son fragmentos en HTML, a través de $route.

```
<body ng-app="home">
    <div id="mainAppContainer">
        <div ng-view></div>
```

Figura 7.15: Fragmento que contiene el DIV principal de index.jsp

7.3.2. Controladores

Los controladores se dividen según su funcionalidad. Existe uno para cada caso de uso del sistema, además de otros específicos para cada tipo de usuario: login, gestion de la vista principal de cada actor, obtener, crear, editar o borrar un elemento, etc.

Existe un controlador principal, donde definimos el módulo de la aplicación Angular. Este módulo incluye las dependencias de otros módulos utilizados, y contiene las diferentes partes de la aplicación. Incluímos en este fichero las diferentes rutas de la aplicación y las variables compartidas (incluídas en factories) porque serán comunes para el uso de todos los controladores y servicios.

```
angular.module('home',['ui.bootstrap', 'ngRoute', 'ngSanitize', 'textAngular', 'ngMaterial']);

angular.module('home').config(['$routeProvider', function($routeProvider) {
   $routeProvider.

   when('/login', {
      templateUrl: 'resources/views/login.jsp'
   }).

   when('/crear_tema', {
          templateUrl: 'resources/views/crear_tema.jsp',
          controller: 'tema_controller',
          controllerAs: 'tema_ctrl'
   }).

   when('/crear_curso', {
          templateUrl: 'resources/views/crear_curso.jsp',
          controller: 'curso_controller',
          controllerAs: 'curso_ctrl'
   }).

   when('/crear_teoria', {
          templateUrl: 'resources/views/crear_teoria.jsp',
          controller: 'teoria_controller',
          controllerAs: 'teoria_ctrl'
   }).

   when('/crear_ejercicio', {
```

Figura 7.16: Declaración del controlador principal, con la configuración de algunas rutas.

La función del resto de controladores será, mediante el uso de servicios y la asignación de valores a $scope, de cargar los datos iniciales en las vistas y gestionar los eventos y la interacción del usuario con elementos de la aplicación como enlaces, botones y formularios.

En la siguiente figura se muestra la definición de un controlador, indicando a que módulo pertenece e incluyendo los servicios que utilizará. Las funciones que se añadan al propio objeto Javascript (self) o al $scope serán accesibles desde la vista.

```
angular.module('home').controller('evaluacion_controller', ['$scope', '$location', 'tema_service', 'ejercicio_service',
    'evaluacion_service', 'Curso', 'Alumno', 'Ejercicio', function($scope, $location, tema_service, ejercicio_service,
            evaluacion_service, Curso, Alumno, Ejercicio) {
    var self = this;

    self.evaluarAlumno = evaluarAlumno;
    self.evaluarEjercicio = evaluarEjercicio;
    self.cargarEjerciciosTema = cargarEjerciciosTema;

    cargar_datos_vista(); // se ejecuta nada mas cargar el controlador

    function cargar_datos_vista(){
```

Figura 7.17: Declaración de un controlador.

7.3.3. Servicios y factorías

La factoría es un tipo de servicio que ofrece Angular que utilizaremos principalmente en nuestra aplicación. En la siguiente figura se muestra un ejemplo de dos variables que serán compartidas por todos los controladores y servicios que las soliciten. Recordemos que las factorías son una especialización de los servicios de Angular, que son Singleton, por lo que sólo se instancian una vez y por lo tanto sus valores durante la ejecución en el cliente serán compartidos.

Para trabajar con los modelos dentro de la aplicación tenemos factorías de Usuario (se actualiza con el usuario que está autenticado en Spring) y para los demás modelos que se manipularán desde la aplicación (alumno, profesor, administrador, curso, tema, ejercicio y evaluación).

```
// Usuario - usuario activo
angular.module('home').factory('Usuario', function () {

    var usuario = {
        id: ''
    };

    return {
        getId: function () {
            return usuario.id;
        },
        setId: function (id) {
            usuario.id = id;
        }
    };
});

//Curso - curso activo
angular.module('home').factory('Curso', function () {

  var curso = {
      id: ''
  };

  return {
      getId: function () {
          return curso.id;
      },
      setId: function (id) {
        curso.id = id;
      }
  };
});
```

Figura 7.18: Variables compartidas entre controladores y servicios, accesibles a través de factorías.

Añadimos un servicio de tipo value para almacenar un diccionario con las rutas disponibles para realizar peticiones sobre el servidor. Sólo debemos asignar el nombre del servidor a la variable *REST_SERVICE_URI* cuando se cambie entre entorno de pruebas o en un servidor.

```
// URIs para el servidor

angular.module('home').value('REST_SERVICE_URI', {
    admin:    host + 'proyecto_app/admin/',     // peticion admin - CUD usuarios, cursos
    auth:     host + 'proyecto_app/auth/',      // peticion admin/profesor - CUD temas, evaluacion
    resource: host + 'proyecto_app/resource/',  // peticion admin/profesor/alumno - acceso a contenido
    login:    host + 'proyecto_app/login/',     // peticion abiertas a todos
    logout:   host + 'proyecto_app/logout/',    // peticion abiertas a todos
});
```

Figura 7.19: Factoría con las rutas disponibles.

Para realizar las peticiones al servidor hemos organizado los servidores en ficheros js distintos, organizados según el modelo que representan. La siguiente figura muestra la declaración de una nueva factoría para el servicio de usuarios. Se indican los servicios que utilizará (de los que tiene dependencia). Al ser invocado, retornará un objeto Javascript con los atributos indicados en la figura, en este caso, son todos funciones para lanzar peticiones al servidor.

```
angular.module('home').factory('usuario_service', ['$http', '$q', 'REST_SERVICE_URI',
                                                   function($http, $q, REST_SERVICE_URI){

    var REST_SERVICE_URI_ADMIN = REST_SERVICE_URI.admin + 'usuario/';
    var REST_SERVICE_URI_AUTH = REST_SERVICE_URI.auth + 'usuario/';

    var factory = {

        listadoAdmins: listadoAdmins,
        obtenerAdmin: obtenerAdmin,
        crearAdmin: crearAdmin,
        editarAdmin: editarAdmin,
        borrarAdmin: borrarAdmin,

        listadoProfesores: listadoProfesores,
        obtenerProfesor: obtenerProfesor,
        crearProfesor: crearProfesor,
        editarProfesor: editarProfesor,
        borrarProfesor: borrarProfesor,

        listadoAlumnos: listadoAlumnos,
        obtenerAlumno: obtenerAlumno,
        crearAlumno: crearAlumno,
        editarAlumno: editarAlumno,
        borrarAlumno: borrarAlumno,
    };

    return factory;
```

Figura 7.20: Declaración de una factoría.

Ahora se muestra una de las funciones anteriores en la figura siguiente. Obsérvese el uso de una promesa para evitar el bloqueo de la ejecución en la petición mientras espera la respuesta del servidor.

```
function listadoProfesores() {
    var deferred = $q.defer();
    $http.get(REST_SERVICE_URI_AUTH + 'profesor')
        .then(
        function (response) {
            deferred.resolve(response.data);
        },
        function(errResponse){
            console.error('Error en el servidor. No se ha podido obtener el listado de profesores.');
            deferred.reject(errResponse);
        }
    );
    return deferred.promise;
}
```

Figura 7.21: Ejemplo de función de una factoría.

CAPÍTULO 8
Pruebas Validación

A continuación realizaremos una serie de pruebas sobre la funcionalidad del proyecto y comprobaremos que todo funciona de manera correcta.

Tras ejecutar el script SQL (anexo), tenemos una base de datos vacía con un sólo usuario, el administrador principal. La primera prueba será utilizar esta cuenta para generar un curso, un usuario con rol de profesor y un segundo usuario con rol de alumno.

Para crear un usuario, tenemos unos validadores en los inputs HTML para impedir introducir datos erróneos. Estos datos se vuelven a comprobar en Spring mediante el uso de un service que hemos llamado ValidarFormulario. Se comprueba que el campos login tenga una longitud mínima y no contenga carácteres extraños, la contraseña debe tener una longitud mínima, el DNI se valida con la librería *valnif*, proporcionada por la AEAT, el email debe cumplir el siguiente patrón, que permite una cadena de carácteres (alfanuméricos) y puntos seguida de una arroba, continuada por una cadena de carácteres, un punto y otra cadena de carácteres obligatoria:

```
String PATTERN_EMAIL = "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@" +
    "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$";
```

Para crear el curso necesitamos asignarle un profesor. Solo aparecerá el usuario profesor que hemos creado anteriormente.

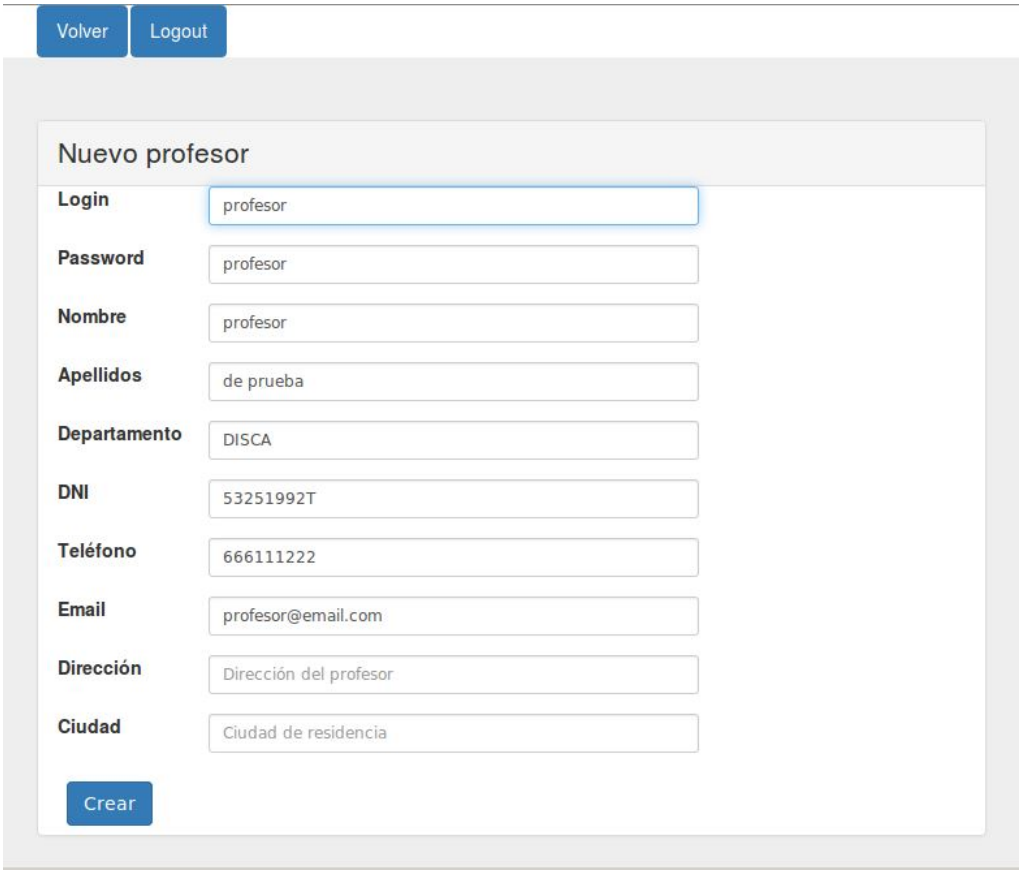


Figura 8.1: Creación de un profesor.

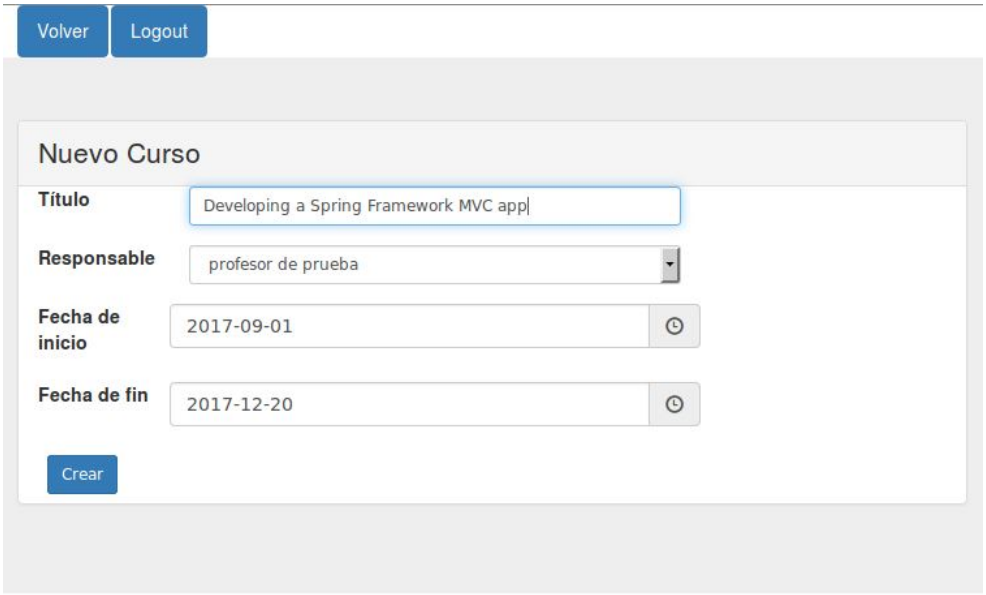


Figura 8.2: Creación de un curso.

Seguidamente creamos el alumno y lo inscribimos en el nuevo curso.

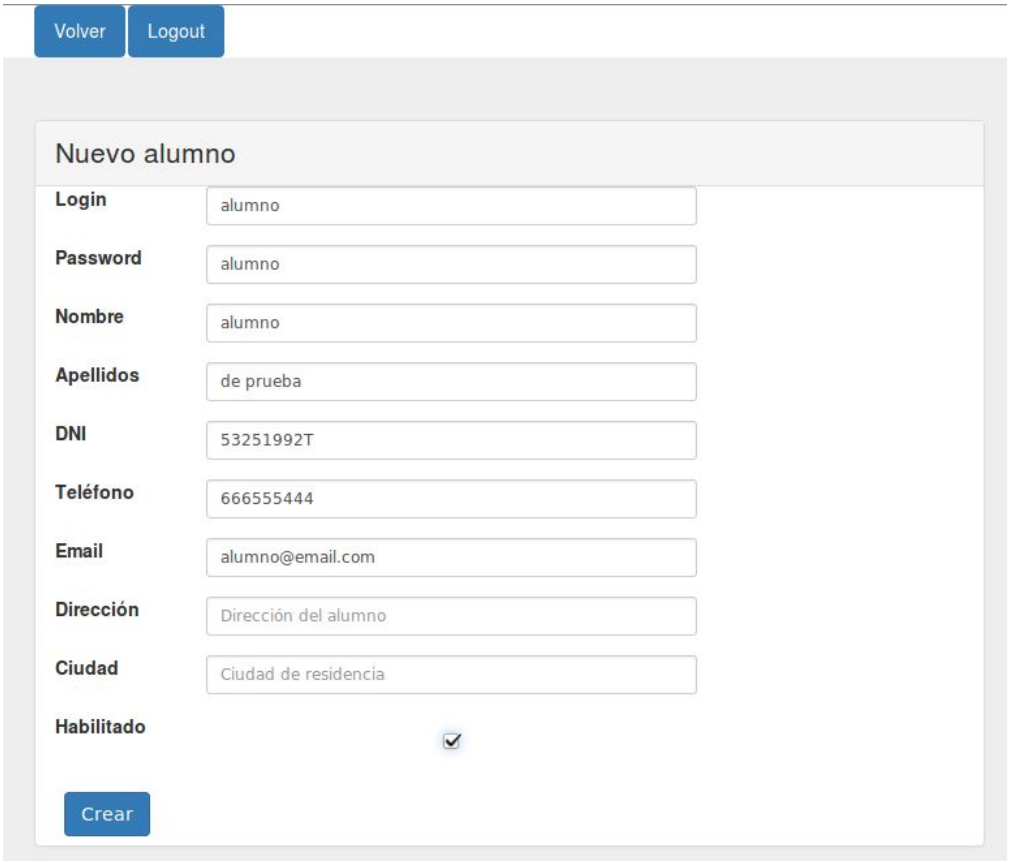


Figura 8.3: Creación de un alumno.

Comprobamos con MySql Workbench que la información almacenada es correcta:

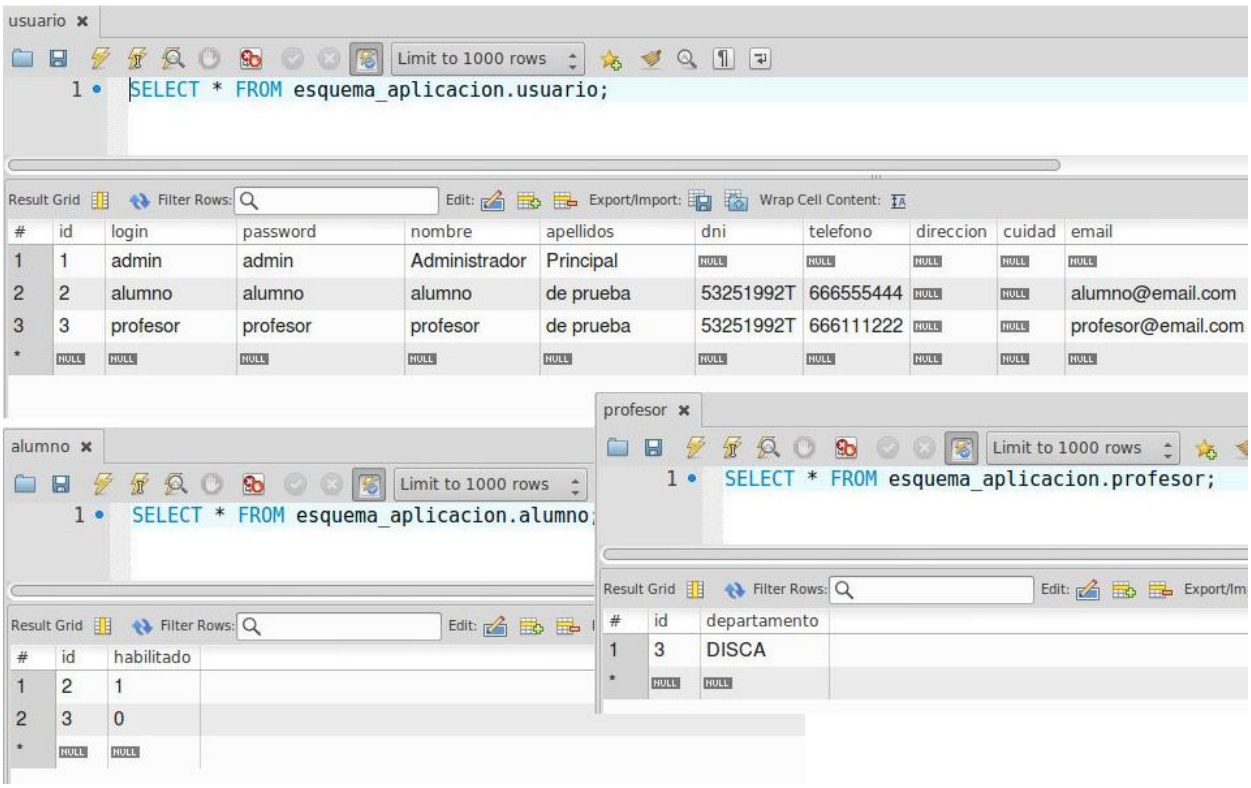


Figura 8.4: Tablas de usuario.

Ahora añadiremos contenido al nuevo curso para que el alumno inscrito pueda acceder a él. Con una pestaña de navegador privada iniciamos otra sesión en el sistema con el login y contraseña asignados al profesor. Accedemos al curso que tenemos asignado y comenzamos a crear el contenido. Creamos un nuevo tema, escribimos algo de teoría y creamos unos ejercicios.

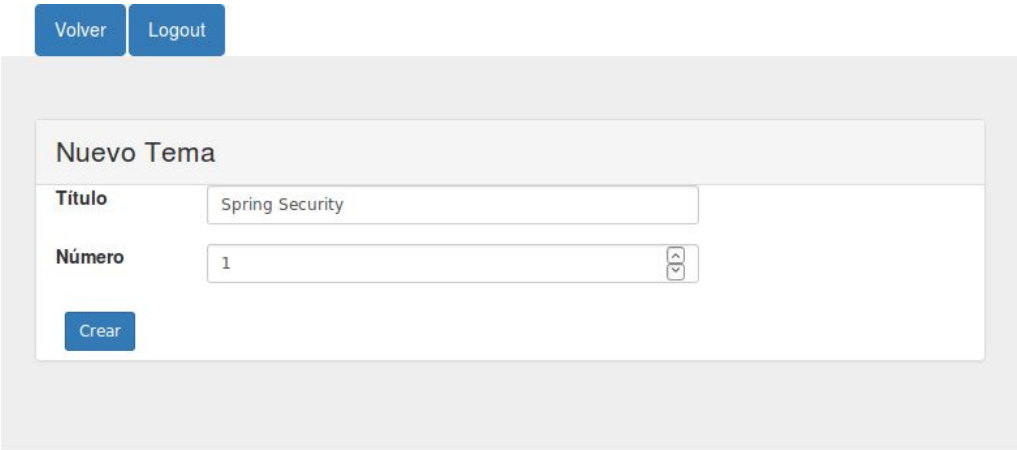


Figura 8.5: Creación de un nuevo tema.

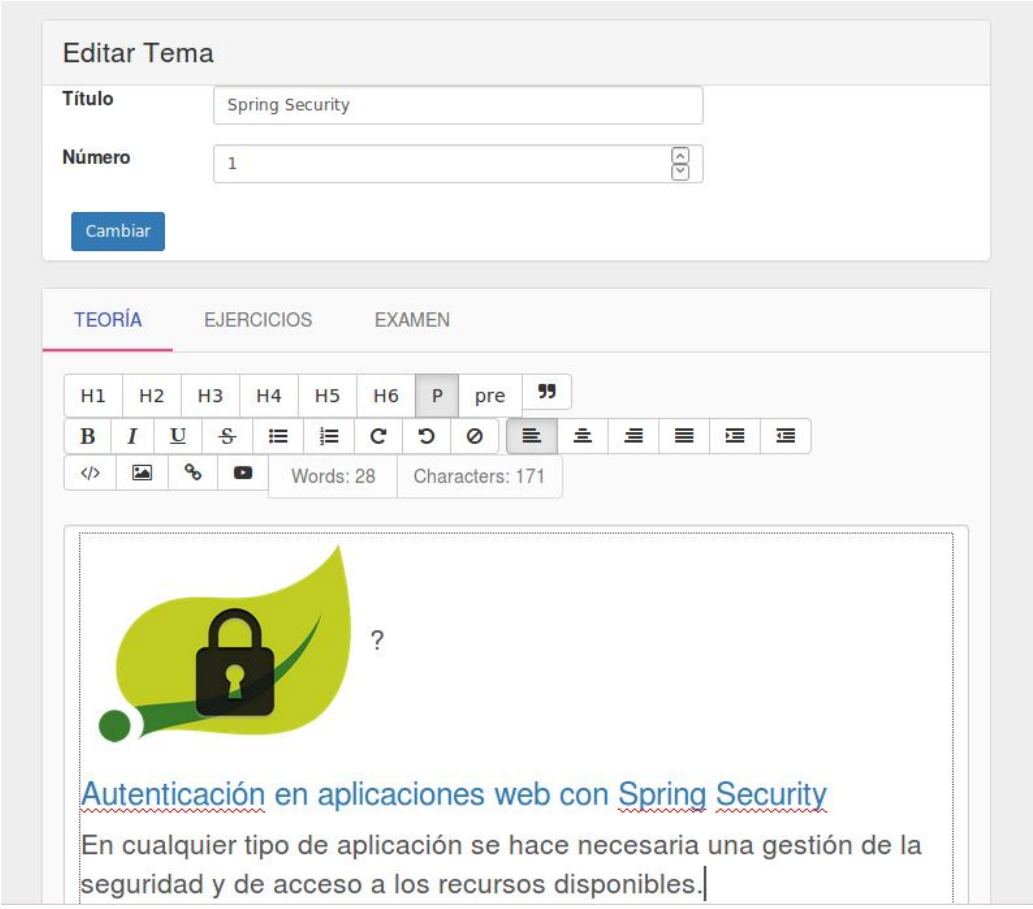


Figura 8.6: Edición del contenido teórico de un tema.

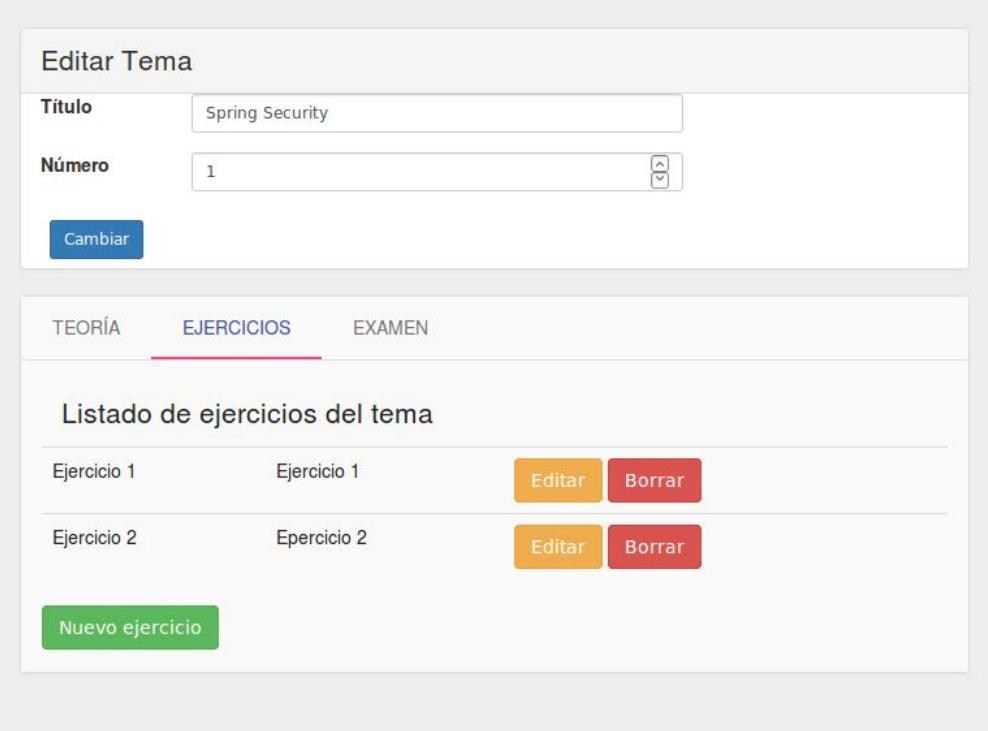


Figura 8.7: Creación de ejercicios para el tema.

Accedemos al sistema como el usuario alumno. Seleccionamos el curso en el que estamos inscritos:

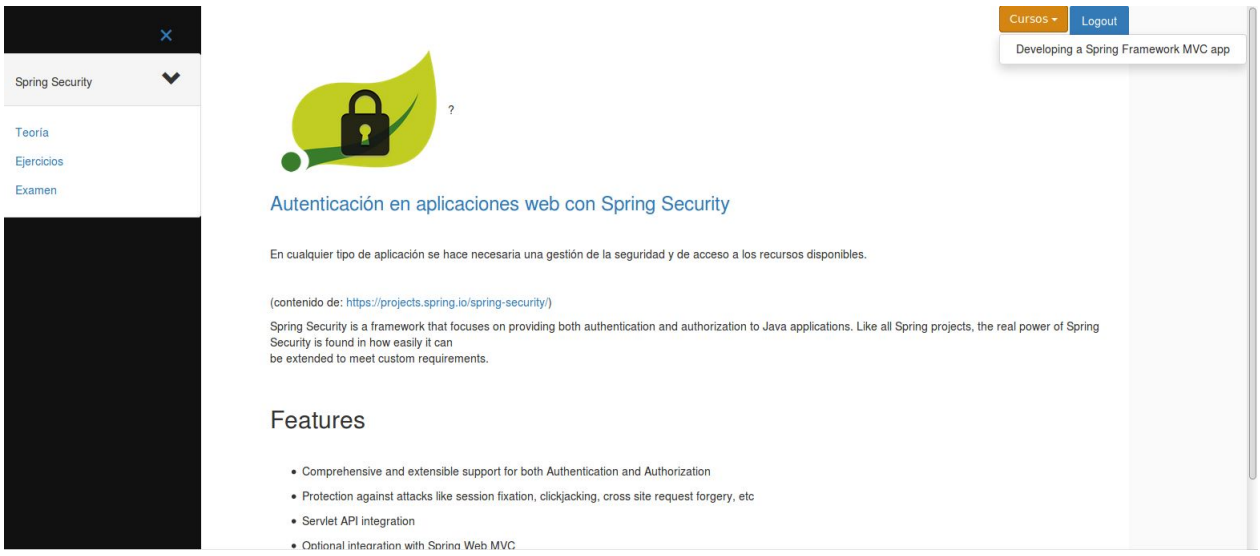


Figura 8.8: Vista principal del alumno.

Hacemos click en uno de los ejercicios y enviamos una respuesta. Se creará una nueva evaluación a la espera de que el profesor la califique.

Utilizamos la sesión del profesor para acceder al curso y entrar en el apartado de evaluación:

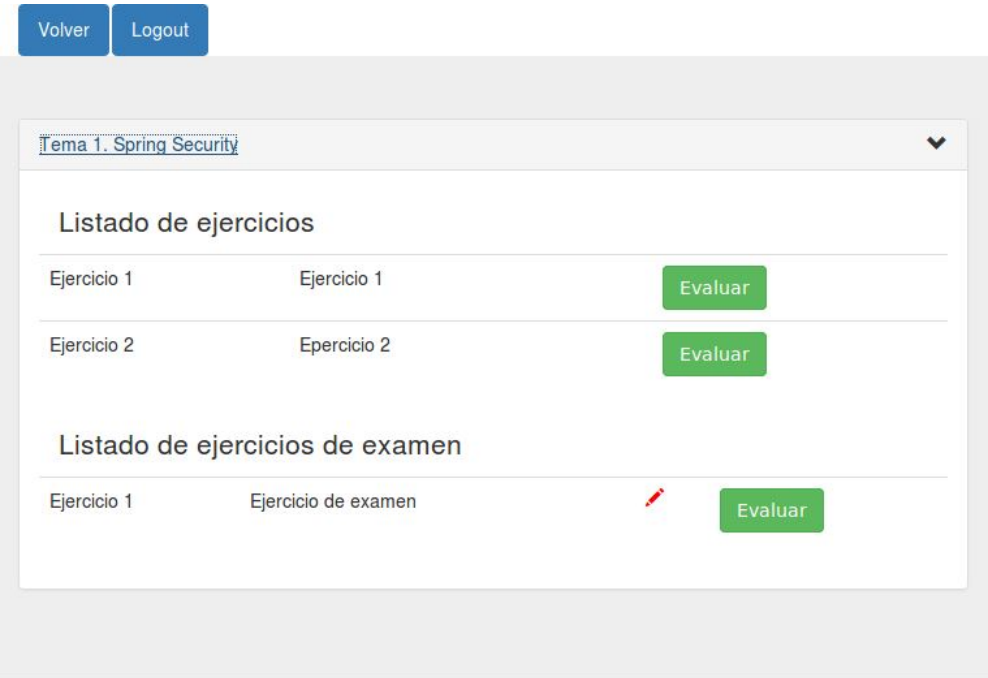


Figura 8.9: Vista de evaluación del profesor.

En esta vista aparecen los ejercicios del curso ordenados por tema y tipo. El icono de bolígrafo rojo indica que tenemos una evaluación pendiente.

Si hacemos click en el botón de evaluar, accedemos a la vista de evaluación. Aquí podremos ver la información del ejercicio y la respuesta otorgada por el alumno. Escribimos una calificación (valor entero limitado entre 0 y 10) y guardamos.

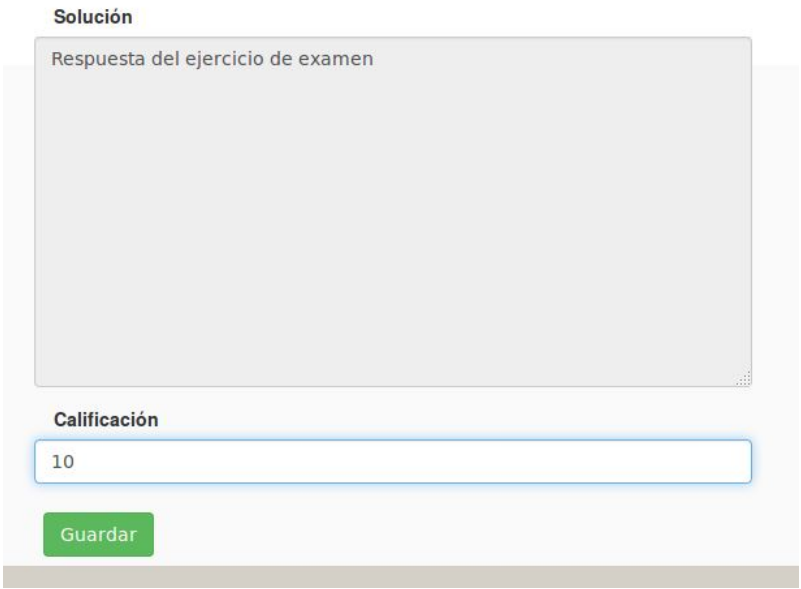


Figura 8.10: Vista de evaluación de un ejercicio.

Se puede comprobar que el ejercicio evaluado ya no tiene el bolígrafo rojo. El check verde indica que es un ejercicio ya evaluado por el profesor.

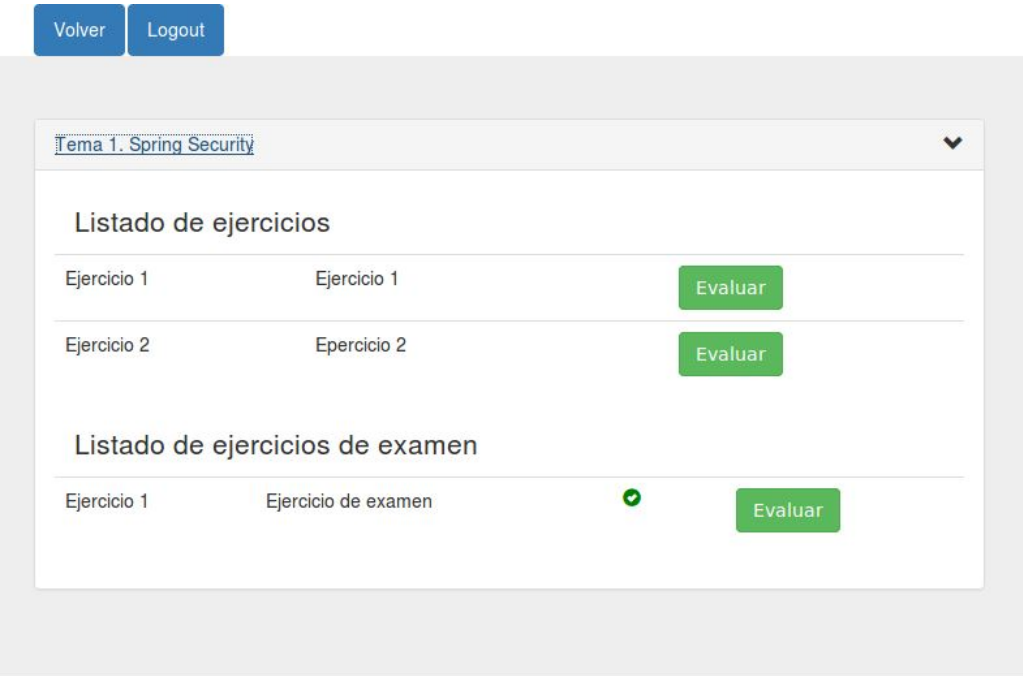


Figura 8.11: Vista de evaluación con ejercicio evaluado.

Accedemos como alumno y, haciendo click de nuevo en el ejercicio, podremos ver la calificación. El botón de mostrar solución sólo se muestra y funciona si la evaluación tiene una calificación. Al pulsarlo hace una consulta al servidor y se trae la respuesta del ejercicio, que muestra al alumno.

Figura 8.12: Vista del alumno de ejercicio evaluado.

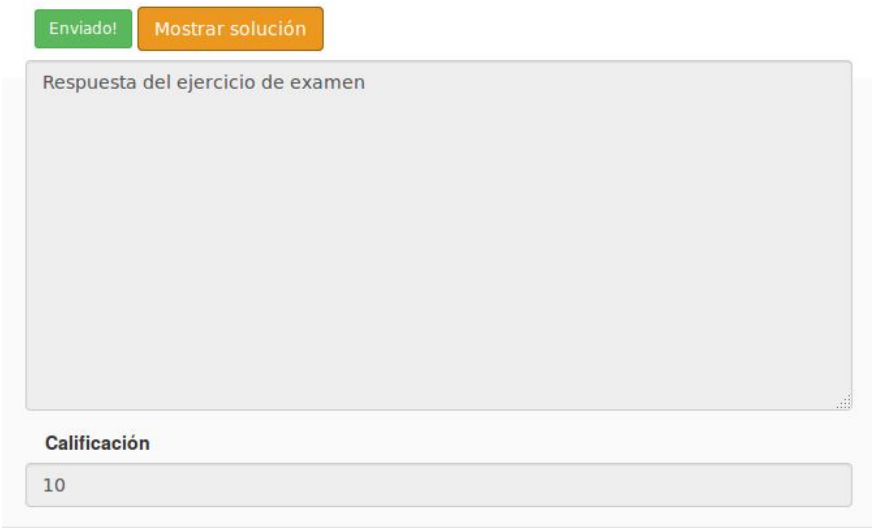


Figura 8.13: Al pulsar el botón de mostrar solución se muestra.

Comprobamos los datos de las tablas con MySql Workbench:

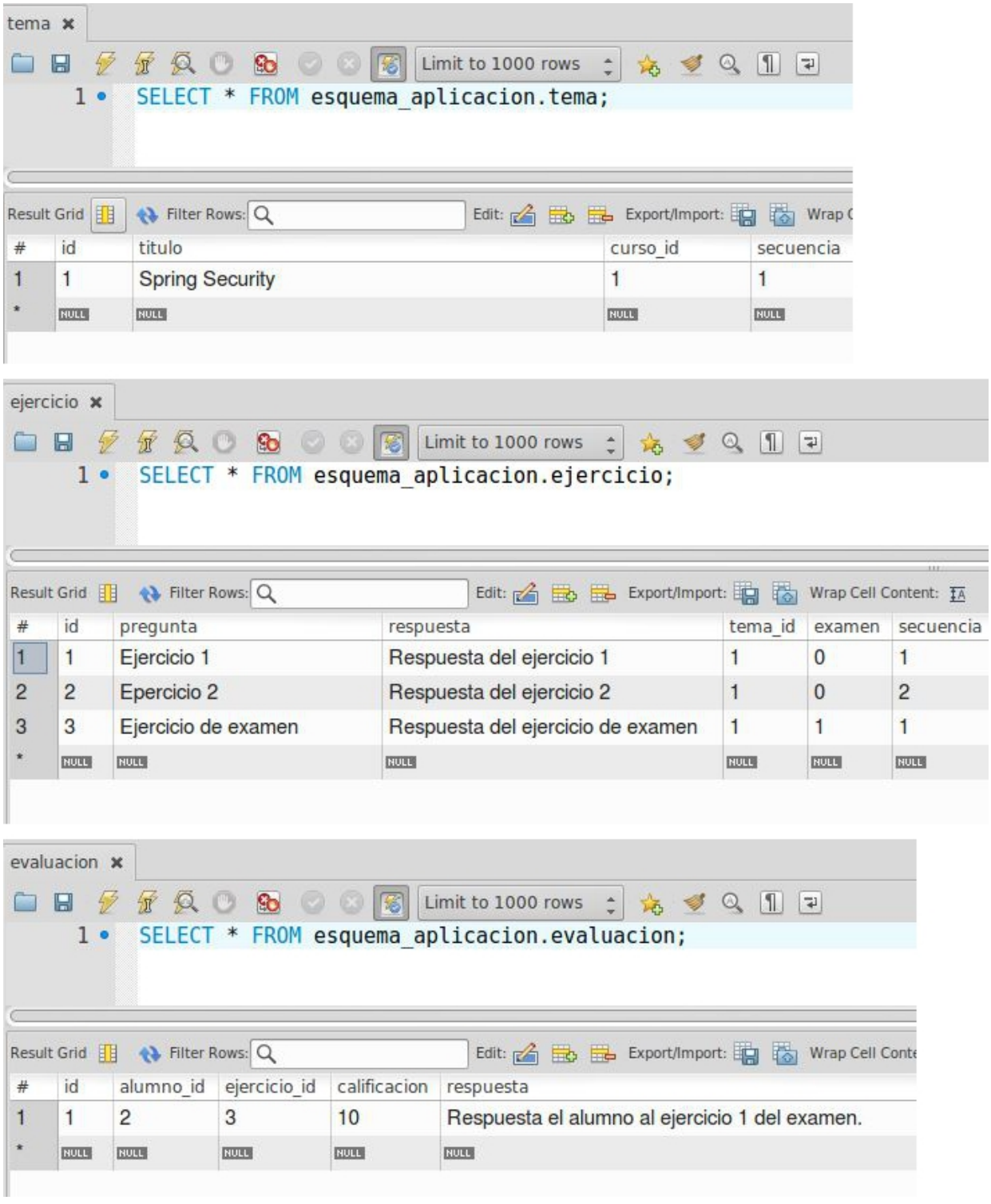


Figura 8.14: Tablas de tema, ejercicio y evaluación tras evaluar.

Figura 8.15: Listado de temas del curso.

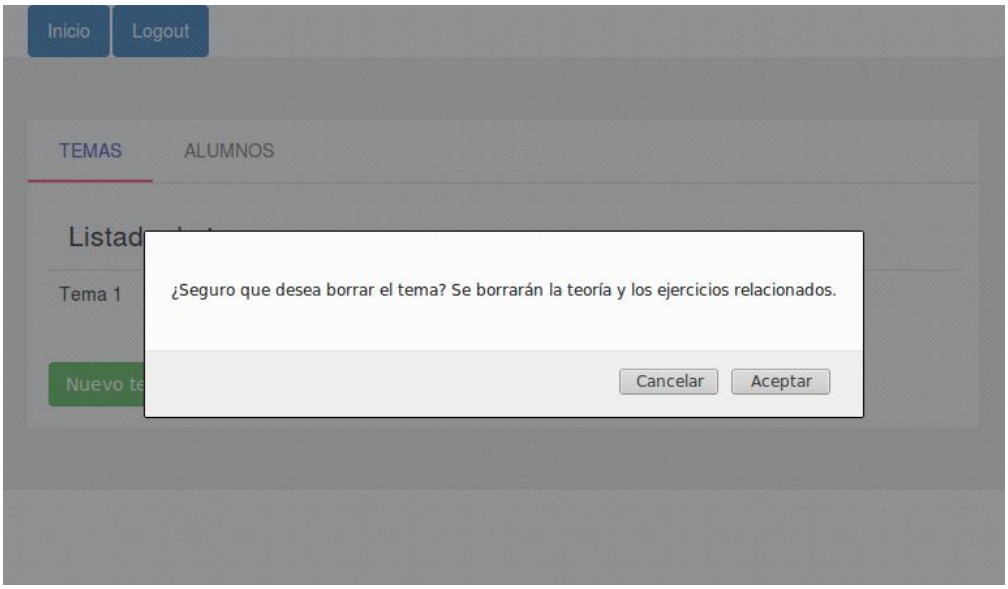


Figura 8.16: Aviso de borrado del temario y todos sus contenidos.

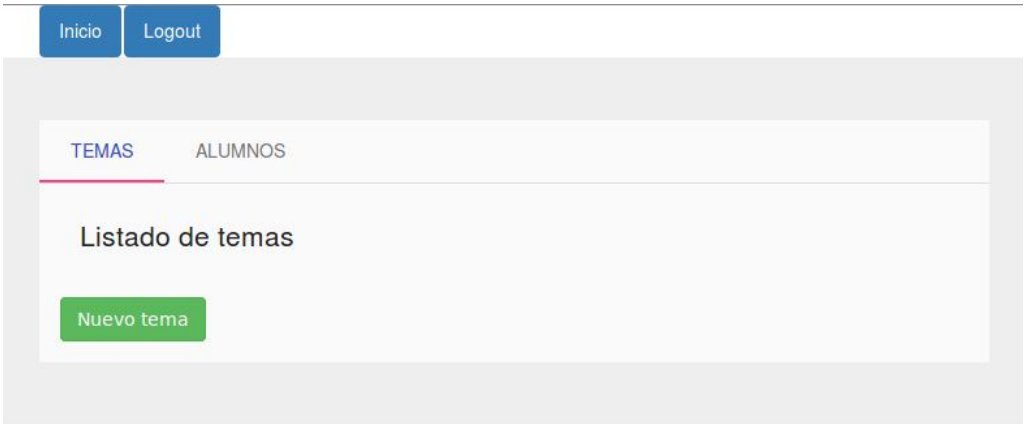


Figura 8.17: El tema ha sido borrado.

Comprobamos con el alumno que el tema ya no está disponible. El contenido relacionado se ha borrado de la base de datos.

Ahora accedemos a la cuenta de administrador y deinscribimos al alumno del curso

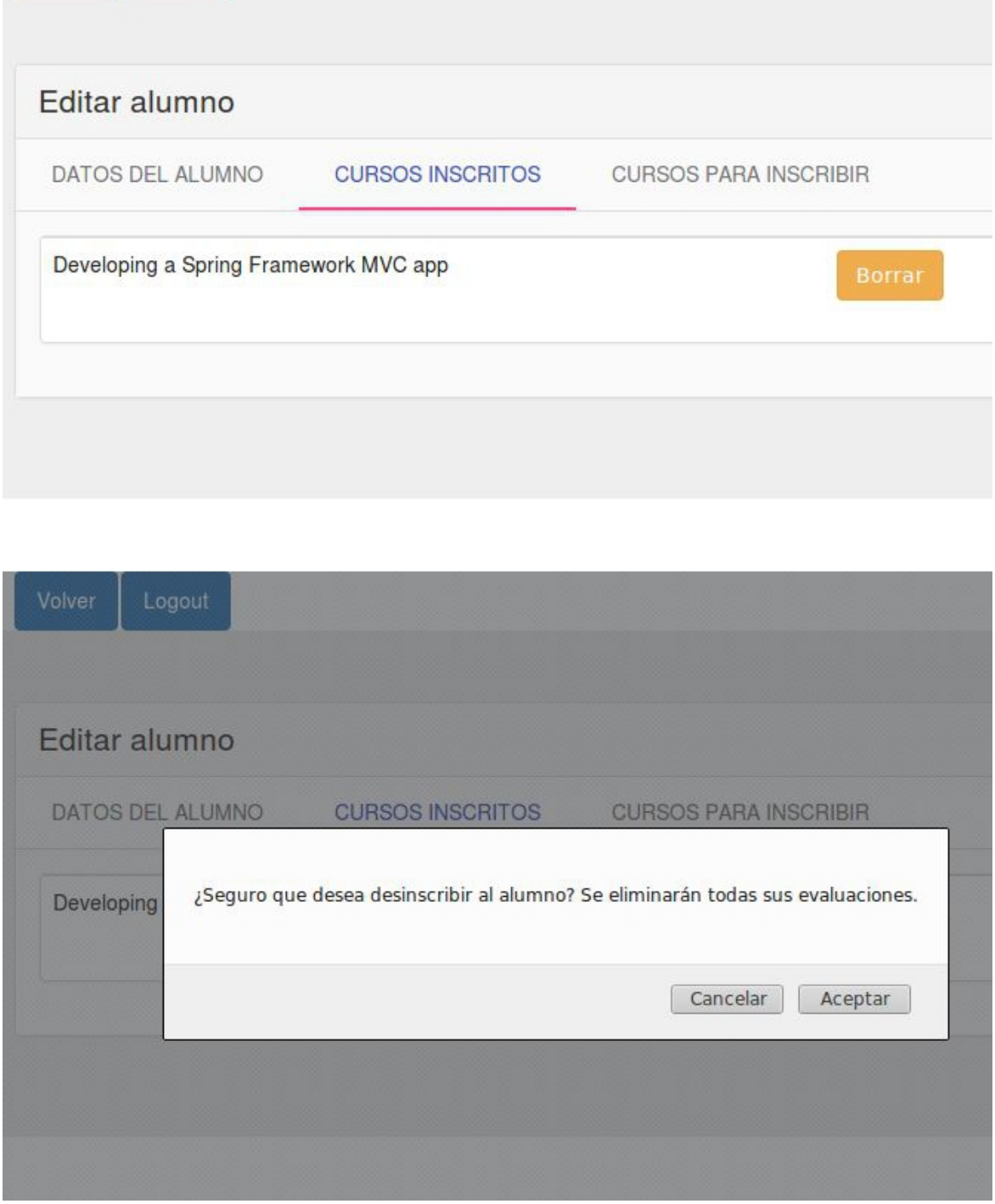


Figura 8.18: Desinscripción de un alumno.

Comprobamos que el alumno no tiene acceso al curso en su vista principal. En base de datos se ha eliminado el registro entre la tabla intermedia de curso y alumno.

CAPÍTULO 9
Conclusiones

Hemos diseñado una plataforma intuitiva y fácil de utilizar por los actores del sistema. Esta plataforma se puede utilizar como punto de partida para agregar en trabajos futuros un sistema de learning analytics, es decir, una serie de herramientas que recopilen datos sobre el uso de la aplicación por el alumno, y poder mejorar la experiencia ofrecida mediante el análisis de los resultados, por ejemplo, estudiando los puntos donde el alumno avanza con mayor lentitud o obtiene peores resultados. En este trabajo no se ha realizado esta implementación por falta de tiempo.

El desarrollo de este proyecto me ha permitido aprender a utilizar dos de las tecnologías más importantes del mercado actual, como son el framework de Javascript Angular JS y el framework de Java EE Spring MVC. Con esto podemos dar por satisfechos los objetivos propuestos durante el capítulo dos.

El código del proyecto es libre y se puede acceder desde el siguiente link:

`https://gitlab.com/AdrianCervera/proyecto_app.git`

Bibliografía

[1] Angular API reference. Consultado en https://docs.angularjs.org/api.

[2] Manual de AngularJS. Consultado en https://desarrolloweb.com/manuales/manual-angularjs.html.

[3] Spring Security. Consultado en https://spring.io/spring-security.

[4] Spring Security and Angular JS. Consultado en https://spring.io/guides/tutorials/spring-security-and-angular-js/#_the_login_page_angular_js_and_spring_security_part_ii.

[5] Roles and Permissions with Spring-Security 3. Consultado en http://slackspace.de/articles/roles-permissions-with-spring-security-3/.

[6] How to Correctly Use BootstrapJS and AngularJS Together. Consultado en https://scotch.io/tutorials/how-to-correctly-use-bootstrapjs-and-angularjs-together.

[7] Registration and Login Example with Spring Security, Spring Data JPA, Spring Boot. Consultado en https://hellokoding.com/registration-and-login-example-with-spring-security-spring-boot-spring-data-jpa-hsql-

[8] Spring Security + Hibernate Annotation Example. Consultado en http://www.mkyong.com/spring-security/spring-security-hibernate-annotation-example/.

[9] Servicios REST con Spring MVC y AngularJS. Consultado en https://www.adictosaltrabajo.com/tutoriales/springmvc-angular/.

[10] Server vs client side rendering (AngularJS vs server side MVC). Consultado en https://technologyconversations.com/2014/07/10/server-vs-client-side-rendering-angularjs-vs-server-side-mvc/.

[11] MVC Architecture. Consultado en http://www.tutorialsteacher.com/mvc/mvc-architecture.

[12] Status Code Definitions. Consultado en https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

APÉNDICE A
Script SQL

```
   -- MySQL Script generated by MySQL Workbench
-- 07/24/16 20:54:59
-- Model: New Model  Version: 1.0
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

-- -----------------------------------------------------
-- Schema esquema_aplicacion
-- -----------------------------------------------------
DROP SCHEMA IF EXISTS `esquema_aplicacion` ;

-- -----------------------------------------------------
-- Schema esquema_aplicacion
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `esquema_aplicacion` DEFAULT CHARACTER SET utf8
    COLLATE utf8_general_ci ;
USE `esquema_aplicacion` ;



-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`usuario`
-- -----------------------------------------------------

-- TODOS LOS USUARIOS, LOGIN Y PASSWORD DE ACCESO
DROP TABLE IF EXISTS `esquema_aplicacion`.`usuario` ;
CREATE TABLE `esquema_aplicacion`.`usuario` (
  `id` BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `login` VARCHAR(45) NOT NULL COMMENT '',
  `password` VARCHAR(64) NOT NULL COMMENT '',
  `nombre` VARCHAR(45) NOT NULL COMMENT '',
  `apellidos` VARCHAR(45) NOT NULL COMMENT '',
  `dni` VARCHAR(10) COMMENT '',
  `telefono` VARCHAR(15) COMMENT '',
  `direccion` VARCHAR(45) NULL COMMENT '',
  `cuidad` VARCHAR(45) NULL COMMENT '',
  `email` VARCHAR(45) NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME)
```

```
ENGINE = InnoDB;

INSERT INTO `esquema_aplicacion`.`usuario` (`id`, `login`, `password`, `nombre`,
    `apellidos`) VALUES ('1', 'admin', 'admin', 'Administrador', 'Principal');



-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`alumno`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`alumno` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`alumno` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `habilitado` TINYINT DEFAULT 0 NOT NULL COMMENT '',
  PRIMARY KEY (`id`) COMMENT '')
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`profesor`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`profesor` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`profesor` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `departamento` VARCHAR(45) NULL COMMENT '',
  PRIMARY KEY (`id`) COMMENT '')
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`administrador`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`administrador` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`administrador` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `principal` TINYINT DEFAULT 0 COMMENT '',
  PRIMARY KEY (`id`) COMMENT '')
ENGINE = InnoDB;


INSERT INTO `esquema_aplicacion`.`administrador` (`id`, `principal`)
VALUES ('1', '1');


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`curso`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`curso` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`curso` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `nombre` VARCHAR(45) NOT NULL COMMENT '',
  `profesor_id` BIGINT NULL COMMENT '',
  `fecha_inicio` DATE NULL COMMENT '',
  `fecha_fin` DATE NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
  `update_date` DATETIME,
  PRIMARY KEY (`id`) COMMENT '',
  CONSTRAINT `profesor_id_curso`
    FOREIGN KEY (`profesor_id`)
    REFERENCES `esquema_aplicacion`.`profesor` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

CREATE INDEX `profesor_id_idx` ON `esquema_aplicacion`.`curso` (`profesor_id`
    ASC) COMMENT '';
-- An index can be created in a table to find data more quickly and efficiently.
-- The users cannot see the indexes, they are just used to speed up
    searches/queries


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`tema`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`tema` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`tema` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `titulo` VARCHAR(45) NOT NULL COMMENT '',
  `curso_id` BIGINT NULL COMMENT '',
  `secuencia` TINYINT NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME,
  PRIMARY KEY (`id`) COMMENT '',
  CONSTRAINT `curso_id_tema`
    FOREIGN KEY (`curso_id`)
    REFERENCES `esquema_aplicacion`.`curso` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

CREATE INDEX `curso_id_idx` ON `esquema_aplicacion`.`tema` (`curso_id` ASC)
    COMMENT '';


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`teoria`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`teoria` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`teoria` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `contenido` VARCHAR(20000) NOT NULL COMMENT '',
  `tema_id` BIGINT NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME,
  PRIMARY KEY (`id`) ,
  CONSTRAINT `tema_id_teoria`
    FOREIGN KEY (`tema_id`)
    REFERENCES `esquema_aplicacion`.`tema` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
CREATE INDEX `tema_id_idx` ON `esquema_aplicacion`.`teoria` (`tema_id` ASC)
    COMMENT '';


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`ejercicio`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`ejercicio` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`ejercicio` (
  `id` BIGINT NOT NULL AUTO_INCREMENT COMMENT '',
  `pregunta` VARCHAR(10000) NOT NULL COMMENT '',
  `respuesta` VARCHAR(10000) NOT NULL COMMENT '',
  `tema_id` BIGINT NOT NULL COMMENT '',
  `examen` TINYINT DEFAULT 0 COMMENT '',
  `secuencia` TINYINT DEFAULT 0 COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME,
  PRIMARY KEY (`id`) COMMENT '',
  CONSTRAINT `tema_id_ejercicio`
    FOREIGN KEY (`tema_id`)
    REFERENCES `esquema_aplicacion`.`tema` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

CREATE INDEX `ejercicio_id_idx` ON `esquema_aplicacion`.`ejercicio` (`tema_id`
    ASC) COMMENT '';


-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`alumno_curso_rel`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`alumno_curso_rel` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`alumno_curso_rel` (
  `id` BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `alumno_id` BIGINT NOT NULL COMMENT '',
  `curso_id` BIGINT NOT NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME,
  CONSTRAINT `alumno_id_curso_rel`
    FOREIGN KEY (`alumno_id`)
    REFERENCES `esquema_aplicacion`.`alumno` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `curso_id_alumno_relj`
    FOREIGN KEY (`curso_id`)
    REFERENCES `esquema_aplicacion`.`curso` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

CREATE INDEX `alumno_id_idx` ON `esquema_aplicacion`.`alumno_curso_rel`
    (`alumno_id` ASC) COMMENT '';
CREATE INDEX `curso_id_idx` ON `esquema_aplicacion`.`alumno_curso_rel`
    (`curso_id` ASC) COMMENT '';
```

```
-- -----------------------------------------------------
-- Table `esquema_aplicacion`.`evaluacion`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `esquema_aplicacion`.`evaluacion` ;

CREATE TABLE IF NOT EXISTS `esquema_aplicacion`.`evaluacion` (
  `id` BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `alumno_id` BIGINT NULL COMMENT '',
  `ejercicio_id` BIGINT NULL COMMENT '',
  `calificacion` TINYINT COMMENT '',
  `respuesta` VARCHAR(20000) NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME,
  CONSTRAINT `alumno_id_evaluacion`
    FOREIGN KEY (`alumno_id`)
    REFERENCES `esquema_aplicacion`.`alumno` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `ejercicio_id_evaluacion`
    FOREIGN KEY (`ejercicio_id`)
    REFERENCES `esquema_aplicacion`.`ejercicio` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

CREATE INDEX `alumno_id_idx` ON `esquema_aplicacion`.`evaluacion` (`alumno_id`
    ASC) COMMENT '';
CREATE INDEX `ejercicio_id_idx` ON `esquema_aplicacion`.`evaluacion`
    (`ejercicio_id` ASC) COMMENT '';


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;



-- EL ROL DE CADA USUARIO (ADMIN,PROFESOR,ALUMNO)
DROP TABLE IF EXISTS `esquema_aplicacion`.`rol` ;
CREATE TABLE `esquema_aplicacion`.`rol` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `rol` VARCHAR(20) NOT NULL COMMENT '',
  `create_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_date` DATETIME,
PRIMARY KEY (`id`) COMMENT '')
ENGINE = InnoDB;

-- ROLES POR DEFECTO PARA LA APP:
INSERT INTO `esquema_aplicacion`.`rol` (`id`, `rol`) VALUES ('1',
    'ROLE_ADMINISTRADOR');
INSERT INTO `esquema_aplicacion`.`rol` (`id`, `rol`) VALUES ('2',
    'ROLE_PROFESOR');
INSERT INTO `esquema_aplicacion`.`rol` (`id`, `rol`) VALUES ('3',
    'ROLE_ADMINISTRADOR');


-- tabla intermedia usuario - rol (rel. many2many)
DROP TABLE IF EXISTS `esquema_aplicacion`.`usuariorol` ;
```

```
CREATE TABLE `esquema_aplicacion`.`usuariorol` (
    `id` BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    `usuarioId` BIGINT NOT NULL,
    `rolId` BIGINT NOT NULL,
    CONSTRAINT `usuario_rol_usuario`
      FOREIGN KEY (`usuarioId`)
      REFERENCES `esquema_aplicacion`.`usuario` (`id`)
      ON DELETE NO ACTION
      ON UPDATE NO ACTION,
    CONSTRAINT `usuario_rol_rol`
      FOREIGN KEY (`rolId`)
      REFERENCES `esquema_aplicacion`.`rol` (`id`)
      ON DELETE NO ACTION
      ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- ROL PARA EL USUARIO ADMIN
INSERT INTO `esquema_aplicacion`.`usuariorol` (`id`, `usuarioId`, `rolId`)
    VALUES ('1', '1', '1');
```