Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Aplicación web de gestión de una academia con Spring Framework

Autor: Eloísa Alés Esquivel

Tutor: Juan Antonio Ternero Muñiz

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Aplicación web de gestión de una academia con Spring Framework

Autor:
Eloísa Alés Esquivel

Tutor:
Juan Antonio Ternero Muñiz
Profesor colaborador

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Proyecto Fin de Carrera: Aplicación web de gestión de una academia con Spring Framework

Autor: Eloísa Alés Esquivel

Tutor: Juan Antonio Ternero Muñiz

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia

A mis profesores

Agradecimientos

En primer lugar, quisiera agradecer a Juan Antonio Ternero la oportunidad que me ha brindado para realizar este proyecto y aprender de él, y al Departamento de Ingeniería Telemática en general.

A mi familia, en especial a mis padres y a mi hermana, por ser las personas con las que he contado, cuento y contaré incondicionalmente durante toda mi vida, porque ha sido un trabajo en equipo, protagonizado por mí, pero con una gran recompensa que sé que ellos sentirán.

A todos los profesores que he tenido durante mi vida académica, no sólo en esta escuela sino también desde pequeña, porque entre todos han formado la base para que hoy pueda ser lo que soy.

A mis compañeros de la universidad, sobre todo a Jesús y a Mª España, porque sin todos vosotros, vuestros resúmenes, vuestros consejos, y vuestra ayuda seguro que no estaba escribiendo estas líneas.

A mis amigos, por estar ahí dándome fuerzas y ánimos, sobre todo a Jose María, con el que tantos momentos de biblioteca he compartido, ya verás como pronto eres todo un Doctor.

A Ruth, porque en estos últimos meses me has hecho ver la vida desde otra perspectiva que jamás imaginé y has conseguido que me valore más en todos los aspectos como persona.

Por último, a Luisa y a Ana, por confiar en mí para la herramienta de gestión de su academia, os deseo todo lo mejor en esta bonita andadura.

Gracias.

Resumen

Actualmente en Sophistas Academia, las responsables llevan la gestión de ésta a base de "papeleo": las altas de los alumnos, de los profesores, sus horarios, las tarifas, etc. Esto provoca que las situaciones de caos se repitan cada vez de manera más continuada a medida que la academia crece.

El objetivo de este proyecto es doble. Por un lado, desarrollar una aplicación web que satisfaga las necesidades expuestas por las responsables de la academia, siguiendo el enfoque propio de un entorno de trabajo real; y por otro, realizar un estudio del *framework* Spring, el cual se ha convertido en un completo ecosistema en el que caben tecnologías para todas las capas de la aplicación, desde el acceso a datos hasta la presentación, pasando por el negocio e incluyendo aspectos transversales como la seguridad.

Abstract

Currently Sophistas Academy has paper-based management systems: personal data of students, teachers, the schedules, the fares, ... This causes chaos situations which occur more continuously as the academy grows.

The target of the project is twofold. On the one hand, developing a web application that meets the needs expressed by those responsible for the academy, following the proper approach of a real work environment; and secondly, a study of the Spring framework, which has become a complete ecosystem in which fits technologies for all layers of the application, from data access to the presentation, through the business and including cross-cutting issues as security.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Muchas palabras no dan prueba del hombre sabio, porque el sabio no ha de hablar sino cuando la necesidad demanda, y las palabras han de ser medidas y correspondientes a la necesidad.

Tales de Mileto

El término sophista, del griego sophía (σοφία), «sabiduría» y sophós (σοφός), «sabio», es el nombre dado en la Grecia clásica al que hacía profesión de enseñar la sabiduría. El pasado diciembre de 2014 mis amigas Luisa y Ana abrieron Sophistas Academia con ese fin, enseñar.

La academia, desde sus inicios, ha ido creciendo y creciendo en alumnos, así como en clases impartidas y en profesores. Esto ha provocado que la gestión de la misma sea cada vez más difícil de llevar a base de "papeleo".

Figura 1-1. Logo de Sophistas Academia

Este hecho, unido a la necesidad de realizar mi Proyecto Fin de Carrera para finalizar los estudios de Ingeniería de Telecomunicación y a los más de tres años de experiencia profesional en el sector del desarrollo web, me ha llevado a realizar la aplicación web "Gestión de Sophistas Academia" que en el presente texto expongo.

1.1 Objetivos del Proyecto

Los objetivos que se han pretendido conseguir con la realización del Proyecto son:

- Diseño de un proyecto web
- Uso del patrón MVC (modelo–vista–controlador)
- Uso del *framework* Spring, una infraestructura de código abierto para la plataforma Java, del cual se ha hecho un estudio teórico usando ejemplos mayoritariamente de la aplicación web desarrollada
- Búsqueda de información de referencia y su aplicación práctica
- Generación de documentación funcional y técnica

1.2 Enfoque metodológico

Para la realización del proyecto se ha intentado seguir el enfoque propio de un entorno de trabajo real.

El primer paso fue una reunión con Luisa y Ana para estudiar detalladamente cuales eran sus necesidades, es

decir, una descripción de todas las tareas que querían llevar a cabo a través de la aplicación.

Posteriormente se hizo un Análisis de requisitos, cuyo objetivo es determinar las condiciones o capacidades que debe cumplir la aplicación que se quiere diseñar para satisfacer las necesidades de sus usuarios. Tras dicho análisis, hubo una nueva reunión para la resolución de ciertas dudas que surgieron en esta etapa y para cerrar puntos y dejar el alcance bien definido.

Tras el análisis vino la fase de diseño. Esta fase fue muy importante ya que en base a lo que se decidió aquí se construyó el software de la aplicación. Además del Diseño técnico, se generó un Plan de pruebas integradas con el fin de certificar que la aplicación abarcara el conjunto de funcionalidades que el Análisis de requisistos había identificado.

Una vez el diseño estaba definido se pasó a la implementación de la aplicación. En esta fase cabe indicar que, a la hora de desarrollar, siempre estuvo presente la idea de explorar la gran cantidad de funcionalidades que ofrece Spring.

Ya implementada la aplicación vino la fase de pruebas. Se fueron pasando las pruebas definidas en el Plan de pruebas integradas y se tuvieron que hacer algunas correcciones hasta conseguir que la totalidad de las pruebas dieran un resultado satisfactorio.

Hubo entonces una nueva reunión con Luisa y Ana, antes de la cual se creó un Manual de usuarios. En esta reunión se les presentó la aplicación apoyándose en dicho manual. La reunión fue satisfactoria, por lo que se pasó a terminar la documentación del proyecto, la cual ya se había avanzado bastante en las diferentes etapas.

Además de las reuniones con las que van a ser las usuarias finales de la aplicación, a lo largo de este tiempo hubo diversas reuniones con el tutor del proyecto en las que se iban viendo los avances de éste.

1.3 Productos obtenidos

Tras la realización del proyecto se han obtenido los siguientes productos:

- Código fuente de la aplicación. Se aporta un fichero .war que puede ser directamente añadido y desplegado por el servidor de aplicaciones.
- Script SQL. Genera el esquema de datos de la aplicación, las tablas y relaciones, así como datos maestros y de prueba.
- Memoria del proyecto. Es el presente documento.
- Presentación. Extracto del contenido de la memoria.

1.4 Estructura de la memoria

La memoria está estructurada en dos partes bien diferenciadas.

La primera parte consta de un análisis sobre las distintas tecnologías que se han empleado en el desarrollo de la aplicación, centrándose en el *framework* Spring. Dichas tecnlogías se pueden observar en la siguiente figura:

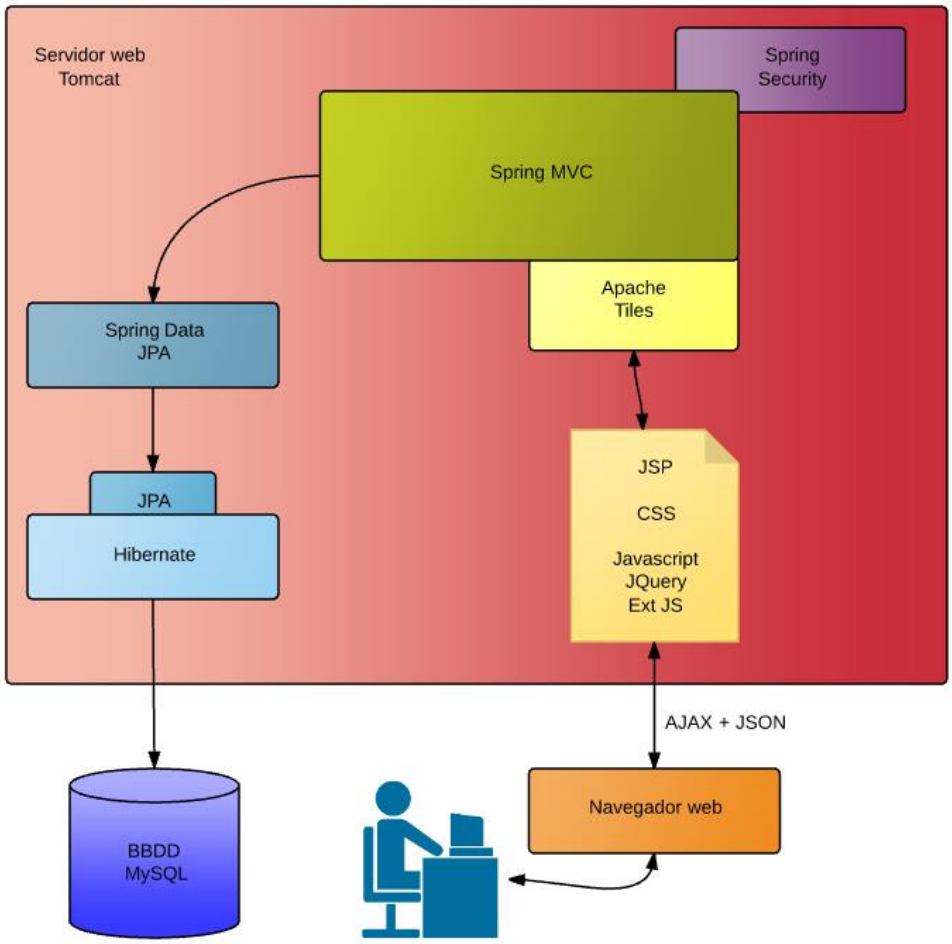


Figura 1-2. Tecnologías usadas en el proyecto

Los capítulos de esta primera parte son:

- Capítulo 2. SPRING.

  En este capítulo se introduce Spring con la presentación de la Inyección de dependencias (DI) y la Programación orientada a objetos (AOP).

- Capítulo 3. ACCESO A DATOS: MYSQL, HIBERNATE, JPA, SPRING DATA.

  En este capítulo se trata el sistema de gestión de base de datos utilizado, MySQL; el mapeo objeto-relacional, y la persistencia en aplicaciones basadas en Spring.

- Capítulo 4. SPRING MVC, LOS CONTROLADORES.

  En este capítulo se presentan los aspectos básicos del uso de Spring MVC, el marco de trabajo web de Spring.

- Capítulo 5. VISTAS.

  En este capítulo se tratan las vistas, las opciones de vistas y resolución de vistas de Spring. Además, se describen las tecnologías usadas en el cliente.

- Capítulo 6. SEGURIDAD: SPRING SECURITY.

  En este capítulo se presenta Spring Security, el marco de trabajo que proporciona seguridad para las aplicaciones basadas en Spring.

La segunda parte consta de los diferentes documentos que se han ido generando en las diferentes fases del

desarrollo de la aplicación web.

Los capítulos de esta segunda parte son:

- Capítulo 8. ANÁLISIS DE REQUISITOS.

  En este capítulo se hace una descripción de la aplicación y se detallan los requerimientos de la misma.

- Capítulo 9. DISEÑO TÉCNICO.

  En este capítulo se especifican la arquitectura y el diseño técnico de la aplicación junto con las tecnologías usadas.

- Capítulo 10. MANUAL DE USUARIO.

  En este capítulo se describe cómo el usuario debe utilizar la aplicación web desarrollada.

- Capítulo 11. PLAN DE PRUEBAS INTEGRADAS

  En este capítulo se lista una serie de pruebas que la aplicación debe pasar para certificar su correcto funcionamiento.

- Capítulo 12. PUESTA EN PRODUCCIÓN

  En este capítulo se describe cómo poner en marcha la aplicación web.

También se ha incluido un anexo, Anexo A. TECNOLOGÍAS USADAS EN EL DESARROLLO, donde se describen las tecnologías usadas para el desarrollo de la aplicación: el entorno de desarrollo Eclipse, la herramienta Maven y las pruebas unitarias con JUnit.

2 SPRING

Podrán cortar todas las flores, pero no podrán detener la primavera.

Pablo Neruda

Spring es un marco de trabajo de código abierto creado por Rod Johnson, quien describió su funcionamiento en el libro "*Expert One-on-One: J2EE Design and Development*". Tiene como objetivo simplificar el desarrollo empresarial con el lenguaje Java.

Actualmente Spring se ha convertido en el estándar de facto para innumerables proyectos de Java y ha influenciado en la evolución de algunas de las especificaciones y estructuras que originalmente pretendía reemplazar. [1] [2]

2.1 Introducción

Para este objetivo de simplificar el desarrollo de aplicaciones, es esencial la Inyección de dependencias (DI) y la Programación orientada a aspectos (AOP), ambas características principales del marco de trabajo Spring.

Inyección de dependencias

La inyección de dependencias (DI) es una forma de asociar objetos de aplicación, de forma que no tengan por qué saber de dónde proceden sus dependencias o la forma en que se implementan. En lugar de adquirir las dependencias por ellos mismos, se proporciona a los objetos dependientes aquellos de los que dependen.

Cualquier aplicación está formada por varios objetos que deben trabajar de forma conjunta para conseguir un objetivo de negocio. Estos objetos deben ser conscientes de la existencia de los otros y comunicarse entre sí para llevar a cabo el trabajo.

El enfoque tradicional al crear asociaciones entre objetos de aplicaciones genera código complicado difícil de reutilizar y de probar como unidades. En Spring, los objetos no son responsables de encontrar o crear el resto de objetos que necesitan para llevar a cabo su trabajo. En su lugar, les asigna referencias a los objetos con los que tienen que colaborar.

La acción de crear estas asociaciones entre objetos de aplicación es la esencia de la inyección de dependencias.

Programación orientada a aspectos

En el desarrollo de software, las funciones que abarcan varios puntos dentro de una aplicación se denominan preocupaciones transversales. Por lo general, estas preocupaciones se encuentran conceptualmente separadas de la lógica de negocio de la aplicación (aunque suelen encontrarse incluidas de forma directa en ésta). Una de estas preocupaciones, presentes en la gran mayoría de las aplicaciones, sería la seguridad.

Separar estas preocupaciones transversales de la lógica de negocio es donde la programación orientada a aspectos (AOP) entra en acción. Se puede pensar en los aspectos como mantas que cubren varios componentes de una aplicación. En su núcleo, una aplicación está formada por módulos que implementan funcionalidades de negocio. Con la AOP se puede cubrir la aplicación principal con capas de funcionalidad. Estas capas pueden aplicarse de forma declarativa a lo largo de una aplicación de forma flexible, sin que el núcleo de la aplicación

ni siquiera sepa que existen.

Mientras que la DI ayuda a desacoplar los objetos de una aplicación entre sí, la AOP ayuda a desacoplar las preocupaciones transversales de los objetos a los que afectan.

2.2 Componentes de Spring

Spring es un marco de trabajo modular que cuenta con una arquitectura organizada en 20 módulos diferentes, que se pueden separar en seis categorías de funcionalidad (véase la figura 2-1).

En conjunto, estos módulos proporcionan todo lo necesario para desarrollar una aplicación empresarial. Además, no es necesario basar la aplicación al completo en Spring, basta con hacer uso de aquellos módulos que se ajusten a ésta. (Spring incluso ofrece puntos de integración con otros marcos de trabajo y bibliotecas).

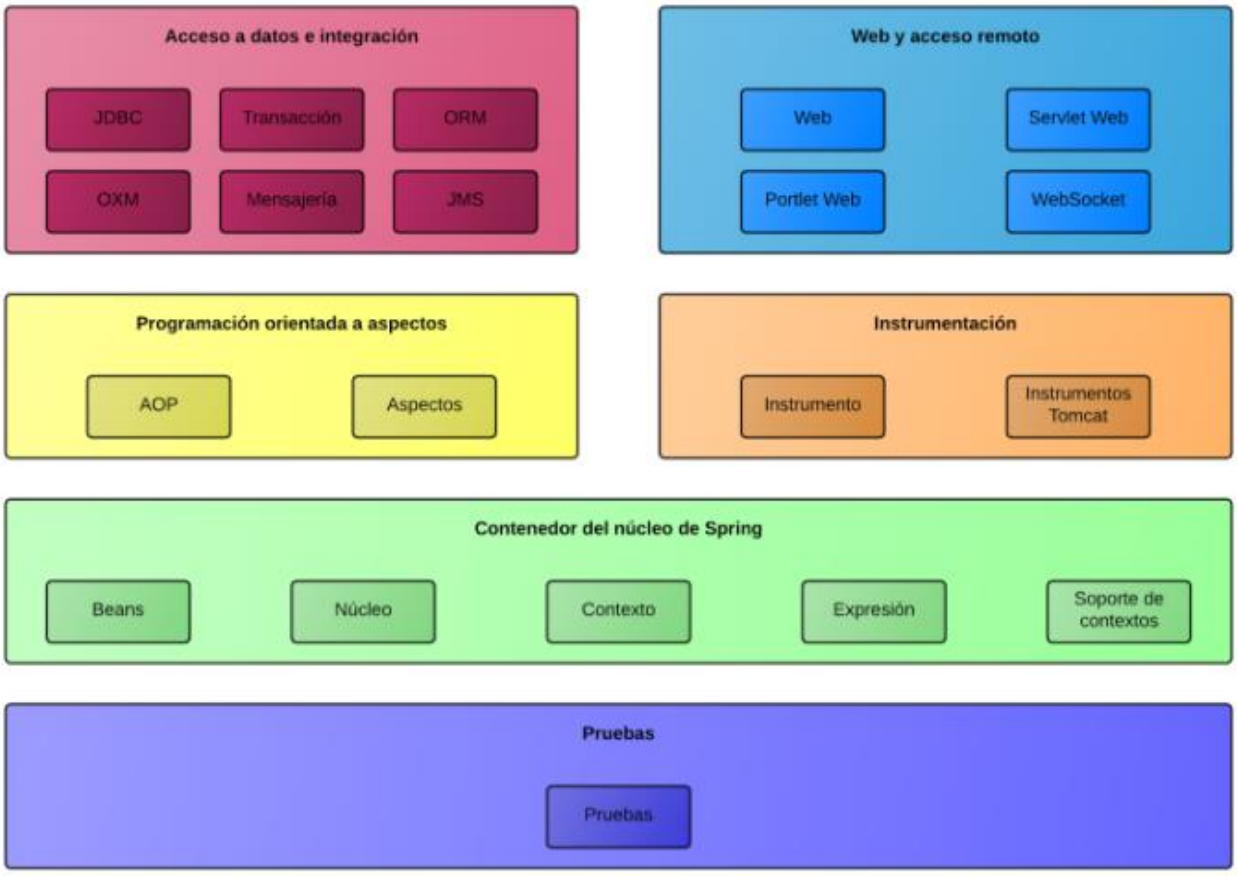


Figura 2-1. Componentes de Spring

A continuación, se describen cada una de las seis categorías:

- Contenedor del núcleo de Spring: El elemento central del marco de trabajo Spring es un contenedor que gestiona la forma en que los *bean*[1] de una aplicación de Spring se crean, configuran y administran. Esta fábrica de *bean* de Spring, es la que proporciona la inyección de dependencias.

[1] Los *bean* son la manera que tiene de denominar Spring a los objetos Java de los que se encarga, es decir aquellos que se encuentren en el contenedor de Spring. Las clases de estos objetos deben cumplir varios criterios:
◦ Implementación en serie.
◦ Tener todos sus atributos privados.
◦ Tener métodos *set()* y *get()* públicos de los atributos privados que nos interesen.
◦ Tener un constructor público por defecto.

- Programación orientada a aspectos: Spring proporciona una amplia compatibilidad para la programación orientada a aspectos mediante los módulos de esta categoría. Sirve como base para desarrollar los aspectos propios de una aplicación Spring.
- Acceso a datos e integración: La persistencia de datos es elemento fundamental de la mayoría de las aplicaciones. Spring proporciona una serie de facilidades para implementar los objetos de la capa de acceso a datos. Permite simplificar el código, reduciendo el código repetitivo, y uniformizar el tratamiento independientemente de la implementación subyacente.
- Web y acceso remoto: El paradigma Modelo-Vista-Controlador (MVC) es un enfoque aceptado, de forma general, para la creación de aplicaciones Web, en el que la interfaz de usuario se separa de la lógica de la aplicación. Spring incluye un marco de trabajo MVC que promueve las técnicas de acoplamiento débil de Spring en el nivel web de una aplicación. Además de para las aplicaciones web de cara al usuario, Spring también incluye varias opciones de acceso remoto para la creación de aplicaciones que interactúan con otras.
- Instrumentación: Spring incluye compatibilidad para añadir agentes a la JVM. (Apenas tiene aplicación práctica.)
- Pruebas: Spring tiene en cuenta la importancia de las pruebas escritas por los desarrolladores y, por ello, incluye un módulo para probar las aplicaciones.

2.3 Inyección de dependencias

Cualquier aplicación está formada por una serie de objetos que deben trabajar de forma conjunta para conseguir un objetivo de negocio.

Estos objetos deben ser conscientes de la existencia de los otros y comunicarse entre sí para llevar a cabo el trabajo. Sin embargo, el enfoque tradicional al crear asociaciones entre objetos de aplicaciones (mediante construcción o búsqueda) genera código complicado, difícil de reutilizar y de probar como unidades.

En Spring, los objetos (*bean*) no son responsables de encontrar o crear el resto de objetos que necesitan para llevar a cabo su trabajo. En su lugar, el contenedor les asigna referencias a los objetos con los que tienen que colaborar.

En una aplicación basada en Spring, sus objetos de aplicación van a residir dentro del contenedor de Spring. El contenedor va a crear los objetos, los va a conectar, a configurar y a administrar su ciclo de vida completo.

Como se indicó en el apartado anterior, el contenedor de Spring se encuentra en el núcleo del marco de trabajo. Utiliza inyección de dependencias (DI) para administrar los componentes que forman una aplicación. Esto incluye la creación de asociaciones entre componentes que colaboran entre sí. De esta forma, los objetos están más limpios y son más fáciles de comprender, permiten su reutilización y son más fáciles de probar.

2.3.1 Ciclo de vida de un *bean*

En una aplicación Java tradicional, el ciclo de vida de un *bean* es sencillo. La palabra clave *new* de Java se utiliza para instanciar el *bean*. A continuación, el *bean* está listo para utilizarlo. Una vez deje de utilizarse, puede eliminarse.

Frente a esto, el ciclo de vida de un *bean* en un contenedor Spring es más complejo, pasa por varios pasos entre su creación y su eliminación. Cada paso es una oportunidad para personalizar la forma en que el *bean* se administra en Spring.

Los pasos son:

1. Spring instancia el *bean*.
2. Spring inyecta valores y referencias de *bean* en las propiedades de éste.
3. Si el *bean* implementa *BeanNameAware*, Spring proporciona el ID del *bean* al método *setBeanName()*.
4. Si el *bean* implementa *BeanFactoryAware*, Spring invoca el método *setBeanFactory()*, proporcionando

él mismo la fábrica de *bean*.

5. Si el *bean* implementa *ApplicationContextAware*, Spring invoca el método *setApplicationContext()*, proporcionándolo en una referencia al contexto de aplicación contenedor.
6. Si el *bean* implementa la interfaz *BeanPostProcessor*, Spring invoca su método *postProcessBeforeInitialization()*.
7. Si el *bean* implementa la interfaz *InitializingBean*, Spring invoca su método *afterPropertiesSet()*. De forma similar, si el *bean* se ha declarado con un método *init*, se invoca el método de inicialización especificado.
8. Si el *bean* implementa la interfaz *BeanPostProcessor*, Spring invoca su método *postProcessAfterInitialization()*.
9. Llegados a este punto, el *bean* estará listo para que la aplicación lo utilice, y va a permanecer en el contexto de la aplicación hasta que se elimine.
10. Si el *bean* implementa la interfaz *DisposableBean*, Spring invoca sus métodos *destroy()*. Del mismo modo, si se ha declarado con un método *destroy*, se invoca el método especificado.

2.3.2 Contenedores para *bean*

Como ya se ha indicado anteriormente, en una aplicación basada en Spring, sus objetos de aplicación van a residir dentro del contenedor de Spring.

Hay varios tipos de contenedores. Spring incluye varias implementaciones de contenedor que pueden clasificarse en dos tipos.

- Las fábricas de *bean* (definidas por la interfaz *org.springframework.beans.factory.BeanFactory*), son los contenedores más sencillos y proporcionan un soporte básico para la DI. Ofrecen una funcionalidad demasiado limitada para la mayoría de aplicaciones.
- Los contextos de aplicación (definidos por la interfaz *org.springframework.context.ApplicationContext*), se crean sobre la base de una fábrica de *bean* que proporciona servicios de marco de trabajo de aplicación.

  Spring cuenta con diferentes tipos de contexto de aplicación. Los más habituales son:

  - *AnnotationConfigApplicationContext*: Carga un contexto de aplicación de Spring desde una o varias clases de configuración basadas en Java.
  - *AnnotationConfigWebApplicationContext*: Carga un contexto de aplicación Web de Spring desde una o varias clases de configuración basadas en Java. Este es el contenedor que se ha usado.
  - *ClassPathXmlApplicationContext*: Carga una definición de contexto a partir de un archivo XML situado en la ruta de clases, y trata los archivos de definición de contexto como recursos de ruta de clases.
  - *FileSystemXmlApplicationContext*: Carga una definición de contexto desde un archivo XML en el sistema de archivos.
  - *XmlWebApplicationContext*: Carga definiciones de contexto a partir de un archivo XML almacenado en una aplicación web.

2.3.3 Configuración de *beans*

Spring ofrece tres mecanismos para configurar los *beans* y sus dependencias:

- Configuración explícita en XML
- Configuración explícita en Java
- Detección implícita y conexión automática de *bean*

Se van a usar las siguientes clases para ver los diferentes mecanismos:

- Clase *Curso* (similar a la clase *Curso* de la aplicación):

```
package com.sophistas.memoria;

public class Curso {

	private Integer nivel;
	private String etapa;

	public Curso() {
	}

	public Curso(Integer nivel, String etapa) {
		this.nivel = nivel;
		this.etapa = etapa;
	}

	public Integer getNivel() {
		return nivel;
	}

	public void setNivel(Integer nivel) {
		this.nivel = nivel;
	}

	public String getEtapa() {
		return etapa;
	}

	public void setEtapa(String etapa) {
		this.etapa = etapa;
	}

}
```

- Clase *Asignatura* (similar a la clase *Asignatura* de la aplicación):

```
package com.sophistas.memoria;

import com.sophistas.memoria.Curso;

public class Asignatura {

	private Curso curso;
	private String nombre;

	public Asignatura() {
	}

	public Asignatura(Curso curso, String nombre) {
		this.curso = curso;
		this.nombre = nombre;
	}

	public Curso getCurso() {
		return curso;
	}

	public void setCurso(Curso curso) {
		this.curso = curso;
	}

	public String getNombre() {
```

```
		return nombre;
	}

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

}
```

2.3.3.1 Configuración explícita en XML

Desde los inicios de Spring, XML ha sido la principal forma de expresar configuraciones.

La configuración XML para Spring consiste en un fichero XML cuya raíz sea un elemento `<beans>`.

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

	<!-- Aquí va la configuración -->

</beans>
```

Para declarar un *bean* se usa el elemento del esquema spring-beans: `<bean>`.

Por ejemplo, para definir un *bean* simple:

```
<bean id="matematicasPrimeroSecundaria" class="com.sophistas.memoria.Asignatura" />
```

Se indica su nombre con el atributo *id* y la clase empleada para crear el *bean* con el atributo *class*.

A la hora de declarar la DI en XML, existen varias opciones:

- *Bean* con inyección por constructor, para lo que se usa el elemento `<constructor-arg>`. Por ejemplo:

  ```
  <bean id="primeroSecundaria" class="com.sophistas.memoria.Curso">
  	<constructor-arg name="nivel" value="1" />
  	<constructor-arg name="etapa" value="SECUNDARIA" />
  </bean>

  <bean id="matematicasPrimeroSecundaria" class="com.sophistas.memoria.Asignatura">
  	<constructor-arg name="curso" ref="primeroSecundaria" />
  	<constructor-arg name="nombre" value="MATEMÁTICAS" />
  </bean>
  ```

  Es equivalente a:

  ```
  Curso primeroSecundaria = new Curso (1, "SECUNDARIA");
  Asignatura matematicasPrimeroSecundaria =
  	new Asignatura(primeroSecundaria, "MATEMÁTICAS");
  ```

- Bean con inyección de propiedades, para lo que se usa el elemento <property>. Por ejemplo:

  ```
  <bean id="primeroSecundaria" class="com.sophistas.memoria.Curso">
  	<property name="nivel" value="1" />
  	<property name="etapa" value="SECUNDARIA" />
  </bean>

  <bean id="matematicasPrimeroSecundaria" class="com.sophistas.memoria.Asignatura">
  	<property name="curso" ref="primeroSecundaria" />
  	<property name="nombre" value="MATEMÁTICAS" />
  ```

```
</bean>
```

Es equivalente a:

```
Curso primeroSecundaria = new Curso();
primeroSecundaria.setNivel(1);
primeroSecundaria.setEtapa("SECUNDARIA");
Asignatura matematicasPrimeroSecundaria = new Asignatura();
matematicasPrimeroSecundaria.setCurso(primeroSecundaria);
matematicasPrimeroSecundaria.setNombre("MATEMÁTICAS");
```

Ambas opciones se pueden combinar.

2.3.3.2 Configuración explícita en Java (JavaConfig)

JavaConfig es la opción preferida para la configuración explícita ya que es más potente, ofrece seguridad de tipos y permite la refactorización.

Conceptualmente se separa de la lógica de negocio de la aplicación. Por ello, aunque no es obligatorio, JavaConfig suele configurarse en un paquete independiente para evitar confusiones a este respecto.[2]

La clave para crear una clase de JavaConfig consiste en anotarla con *@Configuration* (del paquete *org.springframework.context.annotation*), que la identifica como clase de configuración y que se espera que contenga detalles sobre los bean que se van a crear en el contexto de aplicación de Spring.

Para declarar un bean en JavaConfig se debe escribir un método que cree una instancia del tipo deseado y anotarla con *@Bean* (del paquete *org.springframework.context.annotation*).

Siguiendo con los ejemplos usados en el apartado anterior, una definición de bean simple sería:

```
@Bean(name = "matematicasPrimeroSecundaria")
public Asignatura matematicasPrimeroSecundaria() {
	return new Asignatura();
}
```

Esto generará una instancia de un bean que registrar en el contexto de aplicación de Spring, de la clase *Asignatura* y cuyo nombre será *matematicasPrimeroSecundaria*.

A la hora de declarar la DI con JavaConfig para el ejemplo que se está tratando quedaría el siguiente código:

```
package com.sophistas.memoria;

@Configuration
public class Configuracion {

	@Bean(name="primeroSecundaria")
	public Curso primeroSecundaria() {
		Curso curso = new Curso;
		curso.setNivel(1);
		curso.setEtapa("SECUNDARIA");
		return curso;
	}

	@Bean(name = "matematicasPrimeroSecundaria")
	public Asignatura matematicasPrimeroSecundaria() {
		Asignatura asignatura = new Asignatura();
		asignatura.setCurso(primeroSecundaria());
		asignatura.setNombre("MATEMÁTICAS");
		return asignatura;
	}

}
```

[2] En la aplicación "Gestión de Sophistas Academia" se ha creado el paquete *com.sophistas.configuracion* para las clases de configuración.

2.3.3.3 Detección implícita y conexión automática de bean

Éste es el método recomendado siempre que sea posible para evitar los costes de mantenimiento propios de la explícita, además que es la técnica más sencilla.

Spring aborda la conexión automática desde dos frentes:

- Análisis de componentes: Spring detecta automáticamente los bean que debe crear en el contexto de aplicación.
- Conexión automática: Spring satisface automáticamente las dependencias de bean.

Al combinar estas dos técnicas se consigue reducir al mínimo la configuración explícita.

El análisis de componentes no está activado de forma predeterminada. Se debe escribir una configuración explícita para indicarle a Spring que busque clases anotadas con *@Component* (del paquete *org.springframework.stereotype*) y que cree bean para ellas. La anotación *@Component* identifica a una clase como clase de componente y sirve para indicar a Spring que debe crear un bean para la clase.

Por defecto el ID del bean creado va a ser el nombre de la clase anotada con @Component con la primera letra en minúscula. Si se prefiere un ID diferente se debe anotar la clase con @Component("idDelBean").

Para activar el análisis de componentes e indicar a Spring los paquetes que debe analizar (buscar clases anotadas con @Component) existen dos opciones:

- Con la anotación *@ComponentScan*. En el proyecto se ha usado este método, en la clase *WebConfig* del paquete *com.sophistas.configuracion*:

```
@ComponentScan(basePackages={"com.sophistas.negocio", "com.sophistas.accion"})
```

- Habilitación del análisis de componentes en XML. En el elemento raíz `<beans>` se incluye el elemento `<context:component-scan>`.

Si todos los objetos de las aplicaciones fueran independientes y no tuvieran dependencias, solamente sería necesario el análisis de componentes, pero muchos objetos dependen de otros para realizar su labor. Para ello se cuenta con la conexión automática, el otro lado de la configuración automática de Spring.

La conexión automática es una forma de permitir que Spring satisfaga automáticamente las dependencias de un bean buscando a otros bean de la aplicación que coincidan con sus necesidades. Para indicar que debe realizarse la conexión automática, se utiliza la anotación *@Autowired* de Spring (del paquete *org.springframework.beans.factory.annotation*).

Spring intenta satisfacer la dependencia expresada. Si solo hay un bean que coincida, será el que se conecte. Si no hay bean que coincida, Spring generará una excepción cuando se cree el contexto de aplicación. En el proyecto se ha usado de manera extensiva la anotación *@Autowired*.

Si hay varios bean que coincidan, la solución pasa por complementar la anotación *@Autowired* con *@Qualifier* (del paquete *org.springframework.beans.factory.annotation*), para indicar a Spring cuál de los bean coincidentes es el que se debe conectar. Por ejemplo:

```
package com.sophistas.memoria;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

import com.sophistas.memoria.Asignatura;
import com.sophistas.memoria.Curso;

public class Academia {

	@Autowired
	@Qualifier("primeroSecundaria")
	private Curso primeroSecundaria;

	@Autowired
	@Qualifier("matematicasPrimeroSecundaria")
```

```
        private Asignatura primeroSecundaria;

        /* Resto del codigo */

}
```

2.3.3.4 Importar y combinar configuraciones

Ninguna de las tres opciones de configuración vistas es excluyente. Se puede mezclar análisis de componentes y conexión automática con JavaConfig y/o configuración XML. De hecho, es necesario cierta configuración explícita para habilitar el análisis de componentes.

Cuando se combinan estilos de configuración hay que tener en cuenta:

- En lo que respecta a la conexión automática, no importa de donde provenga el bean que conectar. La conexión automática tiene en cuenta todos los bean del contenedor de Spring, independientemente de que se hayan declarado en JavaConfig o XML, o se haya detectado mediante análisis de componentes.

2.4 Programación orientada a aspectos

En el desarrollo de software, las funciones que abarcan varios puntos dentro de una aplicación se denominan preocupaciones transversales. Por lo general, estas preocupaciones se encuentran conceptualmente separadas de la lógica de negocio de la aplicación.

La Programación orientada a aspectos (AOP) entra en acción a la hora de separar estas preocupaciones transversales de la lógica de negocio.

2.4.1 ¿Qué es la Programación orientada a aspectos?

Como se ha indicado con anterioridad, una preocupación transversal puede definirse como cualquier funcionalidad que afecta a varios puntos de una aplicación. Por ejemplo, la seguridad es una preocupación transversal, ya que varios métodos de una aplicación pueden contar con normas de seguridad aplicados a estos.

La AOP permite definir la funcionalidad común en una ubicación y cómo y dónde se va a aplicar, de forma declarativa, sin necesidad de modificar la clase a la que se va a aplicar esta nueva característica.

Las preocupaciones transversales pueden modularizarse en clases especiales llamadas aspectos. Esto ofrece dos ventajas:

- La lógica de cada preocupación se encuentra ahora en una única ubicación, en lugar de encontrarse repartida por todo el código.
- Los módulos de servicio son más claros, ya que solo van a contener el código de su preocupación principal, mientras que las transversales se han transferido a los aspectos.

2.4.2 Terminología de AOP

Los aspectos suelen definirse en términos de consejo, puntos de corte y puntos de cruce. La figura 2-2 muestra cómo se relacionan estos elementos entre sí.

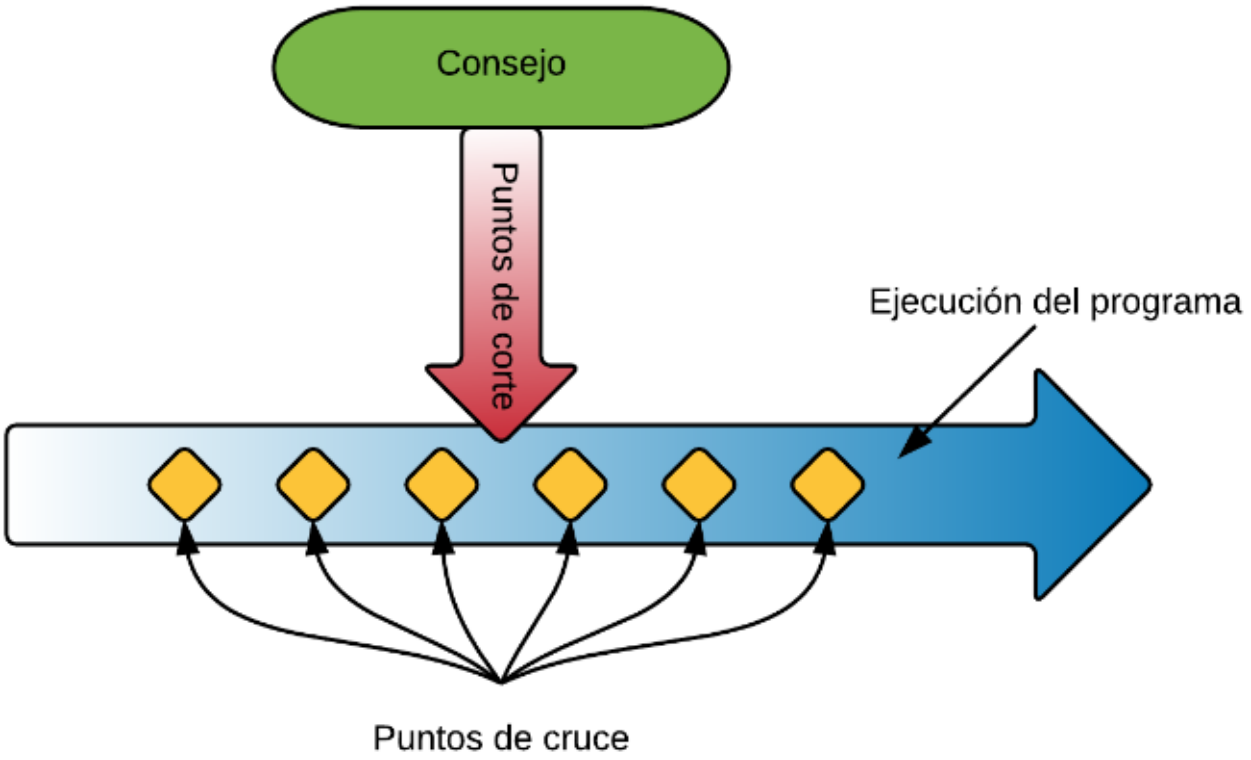


Figura 2-2. Terminología de AOP

2.4.2.1 Consejo

Los aspectos tienen un propósito, es decir, un trabajo para el que se han creado. En términos de AOP, al propósito de un aspecto se le denomina consejo.

Los consejos definen tanto el qué como el cuándo de un aspecto. Además de describir el trabajo que un aspecto debe llevar a cabo, los consejos deben responder a la pregunta de cuándo deben llevarlo a cabo.

Los aspectos de Spring pueden trabajar con cincos tipos de consejos:

- Antes: La funcionalidad del consejo tiene lugar antes de que el método se invoque.
- Después: La funcionalidad del consejo tiene lugar después de que finalice el método, con independencia del resultado.
- Después de la devolución: La funcionalidad del consejo tiene lugar después de que el método se complete con éxito.
- Después de la generación: La funcionalidad del consejo tiene lugar después de que el método genere un error.
- Alrededor: El consejo encapsula el método, lo que proporciona cierta funcionalidad antes y después del método invocado.

2.4.2.2 Puntos de cruce

Un punto de cruce es aquel punto de la ejecución de la aplicación al que puede conectarse un aspecto.

Son los puntos en los que el código de los aspectos puede insertarse en el flujo normal de la aplicación para añadir un nuevo comportamiento.

2.4.2.3 Puntos de corte

Un aspecto no tiene que aconsejar a todos los puntos de cruce de una aplicación. Los puntos de corte ayudan a reducir los puntos de cruce aconsejados por un aspecto.

Si el consejo define el qué y el cuándo de los aspectos, los puntos de corte definen el dónde.

Un aspecto combina los consejos y los puntos de corte. De forma conjunta, definen todo lo que debe hacerse sobre un aspecto: qué hacer, dónde hacerlo y cuándo debe hacerse.

2.4.3 AOP de Spring

A continuación, se describen una serie de puntos claves del marco de trabajo AOP de Spring:

- Todos los consejos que se crean en Spring se escriben en una clase estándar de Java. Los puntos de corte que definen dónde se aplica el consejo pueden especificarse con anotaciones o en XML en el archivo de configuración de Spring.
- Spring aconseja los objetos en tiempo de ejecución.
- Los aspectos de Spring se implementan como proxy que empaquetan el objeto de destino. El proxy gestiona las invocaciones de métodos, aplica lógica de aspectos adicional y, después, invoca el método de destino.
- Spring solo admite puntos de cruce de método. Es una diferencia con respecto a otros marcos de trabajo AOP, que proporcionan puntos de cruce de constructor y de campo además de los de método.

2.4.3.1 Definir puntos de corte

En la siguiente lista se indican los diferentes designadores que se pueden usar en AOP de Spring para definir puntos de corte:

Tabla 2–1 Definir puntos de corte

Designador	Descripción
args()	Limita las coincidencias del punto de cruce a la ejecución de aquellos métodos cuyos argumentos son instancias de los tipos dados.
@args()	Limita las coincidencias del punto de cruce a la ejecución de aquellos métodos cuyos argumentos se anotan con los tipos de anotación proporcionados.
execution()	Hace coincidir los puntos de cruce que son ejecuciones de métodos.
this()	Limita las coincidencias de puntos de cruce a aquellas en las que la referencia de bean del proxy AOP es de un tipo determinado.
target()	Limita las coincidencias de puntos de cruce a aquellas en las que el objeto de destino es de un tipo determinado.
@target()	Limita las coincidencias de puntos de cruce cuando la clase de un objeto en ejecución cuenta con una anotación del tipo dado.
within()	Limita la coincidencia a los puntos de cruce de ciertos tipos.
@within	Limita la coincidencia a los puntos de cruce dentro de ciertos tipos que cuenten con la anotación dada.
@annotation	Limita las coincidencias de puntos de cruce a aquellos en los que el asunto del punto de cruce cuenta con la anotación dada.

2.4.3.2 Definir un aspecto

Para definir un aspecto se debe declarar una clase y anotarla con *@Aspect*. Dentro de la clase, los métodos se

anotan con anotaciones de consejo para indicar cuándo deben invocarse.

A continuación, se indican las diferentes anotaciones para definir aspectos:

Tabla 2–2 Definir un aspecto

Anotación	Descripción
@After	El método de consejo se invoca después de que el método aconsejado devuelva o genere una excepción.
@AfterReturning	El método de consejo se invoca después de que el método aconsejado devuelva un resultado.
@AfterThrowing	El método de consejo se invoca después de que el método aconsejado genere una excepción.
@Around	El método de consejo encapsula al método aconsejado.
@Before	El método de consejo se invoca antes de ejecutarse el método aconsejado.

2.4.3.3 Ejemplo

Para ilustrar el uso de aspectos en Spring, se necesita un elemento que sea el sujeto de los puntos de corte del aspecto.

Para ello, se va a definir la clase *Estudiante*:

```
package com.sophistas.memoria;

public class Estudiante {

	public void realizarExamen() {
		System.out.println("Realizar examen.");
	}

}
```

Por otro lado, se va a definir la clase *PadreEstudiante* como aspecto que se aplica al estudiante:

```
package com.sophistas.memoria;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class PadreEstudiante {

	@Before("execution(* com.sophistas.memoria.Estudiante.realizarExamen(..))")
	public void apuntarAcademia() {
		System.out.println("Apuntar al hijo a una academia que lo prepare.");
	}

	@AfterReturning("execution(* com.sophistas.memoria.Estudiante.realizarExamen(..))"")
	public void felicitarHijo() {
		System.out.println("Felicitar al hijo por la calificación obtenida.");
	}

	@AfterThrowing("execution(* com.sophistas.memoria.Estudiante.realizarExamen(..))")
	public void pedirHijoMayorEsfuerzo() {
		System.out.println("Pedir al hijo mayor esfuerzo para el siguiente examen.");
	}

}
```

PadreEstudiante dispone de tres métodos que describen acciones que los padres de un estudiante pueden realizar a la hora de que sus hijos hagan un examen.

Antes del examen, el padre apunta al hijo a una academia que lo prepare para el mismo. Si en el examen se consigue una buena calificación, el padre felicita al hijo por su resultado. Pero si no es así, el padre le debe pedir al hijo que se esfuerce más en el siguiente examen.

2.4.3.4 Declarar aspectos en XML

Aunque es preferible la configuración basada en anotaciones antes que la basada en XML, también es posible declarar aspectos recurriendo a la configuración XML.

Sin entrar en detalle, y para tener una idea, a continuación, se muestra como quedaría el ejemplo del apartado anterior usando configuración XML.

La clase *PadreEstudiante* quedaría sin anotaciones:

```
package com.sophistas.memoria;

public class PadreEstudiante {

	public void apuntarAcademia() {
		System.out.println("Apuntar al hijo a una academia que lo prepare.");
	}

	public void felicitarHijo() {
		System.out.println("Felicitar al hijo por la calificación obtenida.");
	}

	public void pedirHijoMayorEsfuerzo() {
		System.out.println("Pedir al hijo mayor esfuerzo para el siguiente examen.");
	}

}
```

En el fichero de configuración XML, dentro del elemento raíz `<beans>` se añadiría:

```
<bean id="padreEstudiante" class="com.sophistas.memoria.PadreEstudiante"/>
<aop:config>
   <aop:aspect ref="padreEstudiante">
       <aop:before
           poincut="execution(** com.sophistas.memoria.Estudiante.realizarExamen(..))"
           method="apuntarAcademia"/>
       <aop:after-returning
           poincut="execution(** com.sophistas.memoria.Estudiante.realizarExamen(..))"
           method="felicitarHijo"/>
       <aop:after-throwing
           poincut="execution(** com.sophistas.memoria.Estudiante.realizarExamen(..))"
           method="pedirHijoMayorEsfuerzo"/>
   </aop:aspect>
</aop:config>
```

3 ACCESO A DATOS: MYSQL, HIBERNATE, JPA, SPRING DATA

Always doubt yourself, until the data leave no doubt.

Louis Kronenberg

Nadie concibe una aplicación web sin base de datos. Un requisito que tienen prácticamente todas las aplicaciones es la persistencia de datos. La persistencia es un atributo de los datos que asegura que estarán disponibles incluso más allá de la vida de una aplicación, y aquí entran en juego las bases de datos. Se puede decir que una base de datos es un conjunto de datos persistentes que es utilizado por los sistemas de aplicación de alguna empresa[3] dada.

En este capítulo se va a tratar sobre el sistema de gestión de base de datos utilizado, MySQL; el mapeo objeto-relacional, y la persistencia en aplicaciones basadas en Spring.

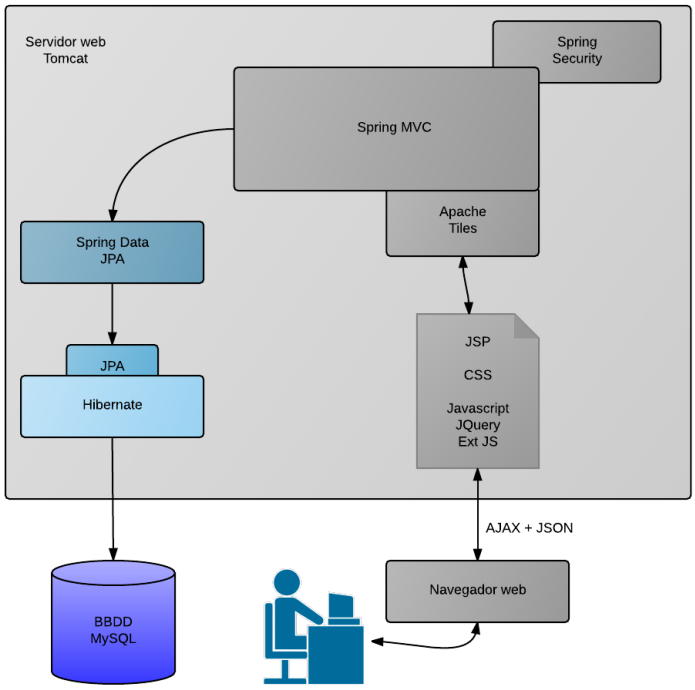


Figura 3-1. Arquitectura del sistema - Acceso a datos: MySQL, Hibernate, JPA, Spring Data

[3] El término empresa es simplemente un término genérico conveniente para identificar a cualquier organización independiente de tipo comercial, técnico, científico u otro.

3.1 Base de datos MySQL

Para la aplicación "Gestión de Sophistas Academia" se ha hecho uso de MySQL. MySQL [3] es un sistema de gestión de bases de datos relacional desarrollado bajo licencia dual GPL/Licencia comercial por Oracle Corporation y está considerada como la base datos *open source* más popular del mundo, y una de las más populares en general junto a Oracle y Microsoft SQL Server, sobre todo para entornos de desarrollo web.

Figura 3-2. Logo MySQL

MySQL es una base de datos muy rápida en la lectura, pero puede provocar problemas de integridad en entornos de alta concurrencia en la modificación. En aplicaciones web hay baja concurrencia en la modificación de datos y en cambio el entorno es intensivo en lectura de datos, lo que hace a MySQL ideal para este tipo de aplicaciones.

3.1.1 Características de MySQL

Entre las características de MySQL destacan:

- Escalabilidad y flexibilidad: Está disponible en gran cantidad de plataformas y sistemas. Ofrece la posibilidad de selección de mecanismos de almacenamiento que ofrecen diferentes velocidades de operación, soporte físico, capacidad, … Además, la naturaleza de código abierto de MySQL permite una personalización completa.
- Alto rendimiento
- Alta disponibilidad
- Apoyo transaccional robusto
- Fuerte protección de datos: Ofrece un sistema de contraseñas y privilegios seguro. Además, viene con soporte incorporado para SSL.
- Facilidad de gestión
- APIs disponibles para múltiples lenguajes de programación

3.1.2 Configuración

Para hacer uso de la base de datos MySQL, hay que incluir en el fichero POM[4] la dependencia del conector de MySQL.

```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<version>${mysql.version}</version>
</dependency>
```

3.2 Mapeo objeto-relacional

Como se ha comentado al inicio del capítulo, la persistencia es un atributo de los datos que asegura que estarán

[4] El fichero POM (*Proyect Object Model*) es el fichero central en que se basa la herramienta de gestión de proyectos Maven, la cual se describe en el Anexo A.

disponibles incluso más allá de la vida de una aplicación. Para un lenguaje orientado a objetos como Java, la persistencia asegura que el estado de un objeto será accesible aunque la aplicación que creó el objeto haya dejado de ejecutarse.

Sin embargo, para conseguir esta persistencia, surge el siguiente problema. La programación orientada a objetos y las bases de datos relacionales son dos paradigmas diferentes. El modelo relacional trata con relaciones y conjuntos. Sin embargo, el paradigma orientado a objetos trata con objetos, sus atributos y asociaciones de unos a otros.

El término de mapeo objeto-relacional (ORM) se refiere a la técnica de mapear una representación de datos desde un modelo de objeto a un modelo de datos relacionales.

3.2.1 Modelo de objeto

En el paquete *com.sophistas.dominio* de la aplicación se encuentran las clases de dominio, llamadas entidades. Estas clases tienen una posición central en la aplicación, representan la información sobre la que la aplicación opera y que se encuentra almacenada en la base de datos.

Los objetos de estas clases (entidades) son los que se corresponden con registros de las tablas de la base de datos. En principio son objetos POJO[5] normales con los cuales se trabaja en la aplicación.

3.2.2 Hibernate y JPA

Hibernate [4] es el *framework* de mapeo objeto-relacional de referencia y una tecnología que está presente en la mayoría de proyectos Java empresariales. Es software libre, distribuido bajo los términos de la licencia GNU LGPL.

Figura 3-3. Logo Hibernate

La inmensa mayoría de los proyectos implementados en lenguajes orientados a objetos utilizan sistemas gestores de bases de datos relacionales, ya que son los más maduros y extendidos en el mercado. *Frameworks* como Hibernate facilitan enormemente tareas repetitivas y tediosas propias de las aplicaciones, como inserción, borrado, actualización, etc., y, por supuesto, abstraen los detalles del gestor de base de datos en la aplicación.

[5] Un POJO (acrónimo de *Plain Old Java Object*) es una sigla utilizada por programadores Java para enfatizar el uso de clases simples y que no dependen de un framework en especial.

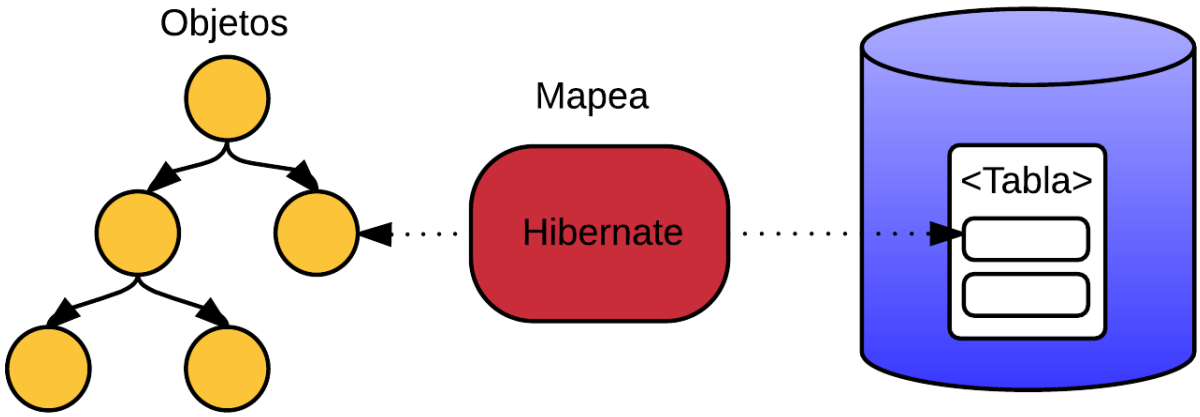


Figura 3-4. Mapeo objeto-relacional

JPA [5] (Java Persistence API) es la API de persistencia para Java EE. Son tres áreas las que contempla principalmente:

- La API definida en el paquete *javax.persistence*.
- JPQL o Java Persistence Query Language: el lenguaje de consulta.
- Metadatos de mapeo, que se pueden expresar en dos formatos: XML y anotaciones Java.

El fin último de esta API es aprovecharse de las ventajas del paradigma de la orientación a objetos, siguiendo el patrón de mapeo modelo-relacional, al interactuar con la base de datos y permitir utilizar los objetos del modelo.

Existen muchas implementaciones diferentes de JPA, entre las cuales Hibernate es la más destacada y es la que se ha utilizado en la aplicación.

3.2.3 Spring y JPA

Actualmente Spring es compatible con JPA. A partir de que se produjera la integración, muchos desarrolladores recomiendan el uso de JPA para la persistencia en aplicaciones basadas en Spring.

3.2.4 Configuración

Para utilizar la API JPA con Hibernate como ORM subyacente se han seguido los siguientes pasos:

- Se han incluido las siguientes dependencias en el fichero POM:

```
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-entitymanager</artifactId>
	<version>${hibernate.version}</version>
</dependency>

<dependency>
	<groupId>javax.transaction</groupId>
	<artifactId>jta</artifactId>
	<version>${jta.version}</version>
</dependency>

<dependency>
	<groupId>org.springframework.data</groupId>
	<artifactId>spring-data-jpa</artifactId>
	<version>${spring-data.version}</version>
</dependency>
```

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-tx</artifactId>
	<version>${spring-framework.version}</version>
</dependency>
```

- En la clase *RootConfig* (dentro del paquete de configuración *com.sophistas.configuracion*) se establece toda la configuración necesaria para la persistencia. Esto incluye los siguientes puntos:
  - Habilitar las transacciones.

    Para ello se usa la anotación de clase *@EnableTransactionManagement* (del paquete *org.springframework.transaction.annotation*).
  - Definir el origen de datos de la aplicación.

    Esta definición se hace mediante el siguiente bean:

```
/*
 * Bean donde se define la base de datos de la aplicacion
 */
@Bean(name = "dataSource")
public DriverManagerDataSource dataSource() {
	DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
	driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver")
	driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/SOPHISTAS");
	driverManagerDataSource.setUsername("sophista");
	driverManagerDataSource.setPassword("S0ph1st4");
	return driverManagerDataSource;
}
```

    Como se puede observar, se indican los diversos parámetros necesarios para establecer una conexión con la base de datos: la clase del *driver* que se va a utilizar, la URL de conexión y unas credenciales.
  - Definir una fábrica de administradores de entidades.

    Esta definición se hace mediante el siguiente bean:

```
/*
 * Bean donde se define una fabrica de administradores de entidades
 */
@Bean
public EntityManagerFactory entityManagerFactory() {
	HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
	vendorAdapter.setGenerateDdl(true);
	vendorAdapter.setShowSql(true);

	LocalContainerEntityManagerFactoryBean factory = new
		LocalContainerEntityManagerFactoryBean();
	factory.setJpaVendorAdapter(vendorAdapter);
	factory.setPackagesToScan("com.sophistas.dominio");
	factory.setDataSource(dataSource());
	factory.afterPropertiesSet();

	return factory.getObject();
}
```

    Para poder utilizar JPA con Spring es necesario configurar una fábrica (*EntityManagerFactory*) de administradores de entidades (*EntityManager*) como bean en el contexto de aplicación de Spring. El *EntityManager* será el encargado de realizar operaciones CRUD (*Create, Read, Update, Delete*) sobre las entidades.

    Como se puede observar, en la configuración se indica:

    - El adaptador de JPA, que en este caso es Hibernate.

- El paquete donde se encuentran las entidades de dominio, que en este caso es *com.sophistas.dominio*.
- El origen de datos.

o Configurar las transacciones de JPA.

Esta tarea se hace mediante el siguiente bean:

```
/*
 * Bean encargado de administrar las transacciones
 */
@Bean
public PlatformTransactionManager transactionManager() {
	JpaTransactionManager txManager = new JpaTransactionManager();
	txManager.setEntityManagerFactory(entityManagerFactory());
	return txManager;
}
```

3.2.5 Mapeo de las clases

Las clases Java a persistir en base de datos (las entidades) deben tener las siguientes características: [6]

- Uno de los constructores debe ser el constructor por defecto, público y sin ningún tipo de argumentos, de manera que Hibernate pueda crear instancias mediante reflectividad.
- La clase no puede ser ni final ni interna (debe ser de primer nivel).
- Si va a ser accedida remotamente debe implementar la interfaz *java.io.Serializable*.
- Por cada propiedad se debe tener un *get/set* asociado.
- Las propiedades son preferiblemente de tipo nullables (*Integer*, *Long*, etc.) mejor que de tipos básicos (*int*, *long*, etc.).
- Para la configuración de mapeo de la clase con la tabla correspondiente en la base de datos se emplean anotaciones que se incorporan en estas clases.

Las anotaciones que se utilizan para el mapeo de las clases vienen contenidas en el paquete *javax.persistence*. Estrictamente las anotaciones obligatorias que se deben incluir son:

- *@Entity*: Se aplica a la clase. Indica que esta clase es una entidad a persistir.
- *@Id*: Se aplica a una propiedad. Indica que este atributo es la clave primaria.

Otras anotaciones (que se emplean en la aplicación) son:

- *@Table*(*name*="NOMBRE_TABLA"): Se aplica a la clase. Indica el nombre de la tabla de la base de datos en la que se persistirá la clase. Si ambos nombres coinciden, no es necesaria ponerla.
- *@Column*(*name*="NOMBRE_COLUMNA"): Se aplica a una propiedad. Indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. Si ambos nombres coinciden, no es necesaria ponerla. Admite otro tipo de atributos como *nullable* o *unique*.
- *@GeneratedValue*: Permite crear un campo identificador en función de una estrategia. Esta anotación se basa en el parámetro *strategy* mediante un *GenerationType* que en nuestro caso va a tomar el valor *IDENTITY*. De esta manera Hibernate se hace responsable de asignar la clave primaria en base a la columna identidad (en MySQL se corresponde con *AUTO_INCREMENT*).

  @GeneratedValue(strategy=GenerationType.IDENTITY)
- *@Temporal*: Es necesaria en propiedades de tipo *java.util.Date* para especificar si es un campo que indica una fecha *(@Temporal*(*TemporalType.DATE*)), una hora *(@Temporal*(*TemporalType.TIME*)), o ambos *(@Temporal*(*TemporalType.TIMESTAMP*)).

Anotaciones para las relaciones

Relación 1:N

En una relación uno a muchos se tienen dos lados: el lado uno (padre) y el lado muchos (hijo). Un elemento de la entidad padre está asociado con muchos elementos de la entidad hijo.

Esta relación puede ser unidireccional (sólo el padre mantiene la referencia a los elementos de la entidad hijo) o bidireccional (también los elementos de la entidad hijo mantienen la referencia a su entidad padre). En la aplicación se han usado relaciones bidireccionales.

Para estas relaciones (bidireccionales) se usan las anotaciones *@OneToMany* y *@ManyToOne*.

- En la entidad padre se usa la siguiente anotación:
    - *@OneToMany*(*mappedBy*="PROPIEDAD_PADRE_EN_HIJO", *cascade*=OPERACIONES_EN_CASCADA, *fetch*=FORMA_RECUPERAR_DATOS, *orphanRemoval=true/false*): Esta anotación se coloca dentro de la entidad padre, en la propiedad que representa a la colección de la entidad hijo.

      El elemento *mappedBy* sirve para indicar la propiedad que en la entidad hijo hace referencia al padre. Este elemento es requerido.

      El elemento *cascade* sirve para especificar qué operaciones se van a hacer en cascada. Se le puede asignar una enumeración de elementos del tipo *javax.persistence.CascadeType*:

        - *CascadeType.MERGE*: Sirve para sincronizar los cambios realizados en un objeto fuera del contexto de Hibernate y volcarlos a la tabla correspondiente de la base de datos.
        - *CascadeType.PERSIST*: Guarda las entidades relacionadas con la entidad salvada.
        - *CascadeType.REFRESH*: Fuerza el refresco de las entidades asociadas cuando se actualiza la entidad principal.
        - *CascadeType.REMOVE*: Elimina las entidades relacionadas con la entidad borrada.
        - *CascadeType.ALL*: Activa todas las cascadas anteriores.

      ** Si no se especifica nada, no se realizan operaciones en cascada.

      El elemento *fetch* sirve para indicar la manera en que se recuperan los datos de la base de datos. Se le puede asignar uno de los dos elementos del tipo *javax.persistence.FetchType*:

        - *FetchType.EAGER*: Al recuperar los datos de la entidad padre, se recuperarán también los datos de los elementos de la entidad hijo, se usen o no.
        - *FetchType.LAZY*: Al recuperar los datos de la entidad padre, no se recuperarán los datos de los elementos de la entidad hijo a menos que se vayan a usar. (Ésta es la manera por defecto).

      El elemento *orphanRemoval* permite borrar de la base de datos los elementos que se hayan quedado huérfanos en una relación al ser eliminados de la lista de elementos de la entidad hijo. Por defecto vale *false*.

- En la entidad hijo se usan las siguientes anotaciones:
    - *@ManyToOne*: Esta anotación se coloca dentro de la entidad hijo, en la propiedad que representa a la entidad padre. (Esta propiedad es la que se debe indicar en el elemento *mappedBy* de la anotación *@OneToMany*).
    - *@JoinColumn*(name="CLAVE_FORANEA_EN_TABLA_DE_ENTIDAD_HIJO"): Esta anotación se coloca dentro de la entidad hijo, en la propiedad que representa a la entidad padre. Establece la clave foránea que une las dos tablas.

Relación M:N

En una relación muchos a muchos, se relaciona un conjunto de entidades con otro conjunto de entidades distintas. Para este tipo de operaciones, en base de datos se usa una tercera tabla que se suele llamar tabla de unión. (Por ejemplo, en la base de datos SOPHISTAS existen dos tablas de unión: CLASE_ALUMNO y CLASE_HORASEMANAL).

El propósito de esta tabla de unión es establecer las relaciones entre las entidades de ambas tablas que están relacionadas. Para mantener esa relación, no se necesita más que los identificadores de las dos tablas como claves foráneas, y con eso se mantiene la tabla.

Al utilizar Hibernate no se necesita gestionar esta tabla ya que el propio *framework* se encarga de ella. Sólo hay que preocuparse de establecer de forma correcta los mapeos para representar este vínculo.

Esta relación puede ser unidireccional (sólo una de las partes mantiene la referencia) o bidireccional (las dos partes mantienen referencia). En la aplicación se han usado relaciones bidireccionales.

En este tipo de relaciones se tienen dos lados: el lado dueño y el lado inverso.

En el lado dueño se usarán las siguientes anotaciones:

- *@ManyToMany*(*cascade*=OPERACIONES_EN_CASCADA, *fetch*=FORMA_RECUPERAR_DATOS): Esta anotación se coloca dentro de la entidad del lado dueño, en la propiedad que representa a la colección de la entidad del lado inverso. Los elementos *cascade* y *fetch* son opcionales y pueden tomar los mismos valores que los indicados anteriormente para la anotación *@OneToMany*.
- *@JoinTable*(*name*="NOMBRE_TABLA_DE_UNION", *joinColumns*= *@JoinColumn*(*name*="CLAVE_FORANEA_EN_TABLA_DE_UNION_LADO_DUEÑO", *referencedColumnName*="CLAVE_PRIMARIA_EN_LADO_DUEÑO"), *inverseJoinColumns*=*@JoinColumn*(*name*="CLAVE_FORANEA_EN_TABLA_DE_UNION_LADO_INVERSO", r*eferencedColumnName*="CLAVE_PRIMARIA_EN_LADO_INVERSO")): Esta anotación se coloca dentro de la entidad del lado dueño, en la propiedad que representa a la colección de la entidad del lado inverso. Establece las relaciones entre la tabla de unión y las dos tablas que ésta une.

- En el lado inverso se usa la siguiente anotación:
  - *@ManyToMany*(*mappedBy*="PROPIEDAD_LADO_INVERSO_EN_LADO_DUEÑO", *cascade*=OPERACIONES_EN_CASCADA, *fetch*=FORMA_RECUPERAR_DATOS): Esta anotación se coloca dentro de la entidad del lado inverso, en la propiedad que representa a la colección de la entidad del lado dueño. Los elementos *cascade* y *fetch* son opcionales y pueden tomar los mismos valores que los indicados anteriormente para la anotación @OneToMany. El elemento *mappedBy* es obligatorio y sirve para indicar la propiedad que en el lado dueño hace referencia al lado inverso.

Ejemplos de clases mapeadas

A continuación, se observa el código de la clase mapeada *Clase* de la aplicación. Es la entidad hijo de una relación 1:N con *Tarifa* y la entidad del lado dueño de una relación M:N con *HoraSemanal*.

```
package com.sophistas.dominio;

import java.io.Serializable;
import java.util.List;

import javax.persistence.*;

@Entity
```

```java
@Table(name="CLASE")
public class Clase implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long id;

	@ManyToOne
	@JoinColumn(name="ID_ASIGNATURA")
	private Asignatura asignatura;

	@ManyToOne
	@JoinColumn(name="ID_PROFESOR")
	private Profesor profesor;

	@ManyToOne
	@JoinColumn(name="ID_TARIFA")
	private Tarifa tarifa;

	@ManyToMany
	@OrderBy("diaIndice, horaIndice")
	@JoinTable(name="CLASE_HORASEMANAL",
	 joinColumns=
	     @JoinColumn(name="ID_CLASE", referencedColumnName="Id"),
	 inverseJoinColumns=
	     @JoinColumn(name="ID_HORASEMANAL", referencedColumnName="Id")
	 )
	private List<HoraSemanal> horasSemanales;

	@ManyToMany(mappedBy="clases")
	private List<Alumno> alumnos;

	public Clase() {
	}

	public Long getId() {
		return this.id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public Asignatura getAsignatura() {
		return asignatura;
	}

	public void setAsignatura(Asignatura asignatura) {
		this.asignatura = asignatura;
	}

	public Profesor getProfesor() {
		return profesor;
	}

	public void setProfesor(Profesor profesor) {
		this.profesor = profesor;
	}

	public Tarifa getTarifa() {
		return tarifa;
	}

	public void setTarifa(Tarifa tarifa) {
		this.tarifa = tarifa;
	}
```

```
	public List<HoraSemanal> getHorasSemanales() {
		return horasSemanales;
	}

	public void setHorasSemanales(List<HoraSemanal> horasSemanales) {
		this.horasSemanales = horasSemanales;
	}

	public List<Alumno> getAlumnos() {
		return alumnos;
	}

	public void setAlumnos(List<Alumno> alumnos) {
		this.alumnos = alumnos;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Clase other = (Clase) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

A continuación, se observa el código de la clase mapeada *Tarifa* de la aplicación. Es la entidad padre de una relación 1:N con *Clase*.

```
package com.sophistas.dominio;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
public class Tarifa implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long id;

	private String descripcion;

	private String nombre;

	private Double precio;

	@OneToMany(mappedBy="tarifa", fetch = FetchType.LAZY)
	private List<Clase> clases;

	public Tarifa() {
```

```
	}

	public Long getId() {
		return this.id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getDescripcion() {
		return this.descripcion;
	}

	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}

	public String getNombre() {
		return this.nombre;
	}

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

	public Double getPrecio() {
		return this.precio;
	}

	public void setPrecio(Double precio) {
		this.precio = precio;
	}

	public List<Clase> getClases() {
		return clases;
	}

	public void setClases(List<Clase> clases) {
		this.clases = clases;
	}

}
```

A continuación, se observa el código de la clase mapeada *HoraSemanal* de la aplicación. Es la entidad del lado inverso de una relación M:N con *Clase*.

```
package com.sophistas.dominio;

import java.io.Serializable;
import javax.persistence.*;
import java.util.List;

@Entity
@Table(name="HORA_SEMANAL")
public class HoraSemanal implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long id;

	private String dia;
```

```
	private String hora;

	@Column(name="DIA_INDICE")
	private Integer diaIndice;

	@Column(name="HORA_INDICE")
	private Integer horaIndice;

	@ManyToMany(mappedBy="horasSemanales")
	private List<Clase> clases;

	public HoraSemanal() {
	}

	public Long getId() {
		return this.id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getDia() {
		return this.dia;
	}

	public void setDia(String dia) {
		this.dia = dia;
	}

	public String getHora() {
		return hora;
	}

	public void setHora(String hora) {
		this.hora = hora;
	}

	public Integer getDiaIndice() {
		return diaIndice;
	}

	public void setDiaIndice(Integer diaIndice) {
		this.diaIndice = diaIndice;
	}

	public Integer getHoraIndice() {
		return horaIndice;
	}

	public void setHoraIndice(Integer horaIndice) {
		this.horaIndice = horaIndice;
	}

	public List<Clase> getClases() {
		return clases;
	}

	public void setClases(List<Clase> clases) {
		this.clases = clases;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
```

```
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		HoraSemanal other = (HoraSemanal) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

3.3 Repositorios con Spring Data JPA

En la aplicación "Gestión de Sophistas Academia", para los repositorios[6], se ha usado Spring Data JPA [7].

Spring Data es un proyecto que se encuentra dentro de la plataforma de Spring. Su propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia. Entre estas tecnologías se encuentra JPA, a la cual se dedica el subproyecto Spring Data JPA.

Para cada entidad JPA se ha creado un repositorio de Spring Data, que no es más que una interfaz que especializa *JpaRepository* (no se necesita crear una clase que la implemente). Esta interfaz proporciona "de serie" las operaciones CRUD más habituales y que por lo tanto no se tienen que implementar. El siguiente diagrama de clases muestra los métodos de los que se disponen:

[6] Los repositorios son las clases usadas para el acceso a la base de datos.

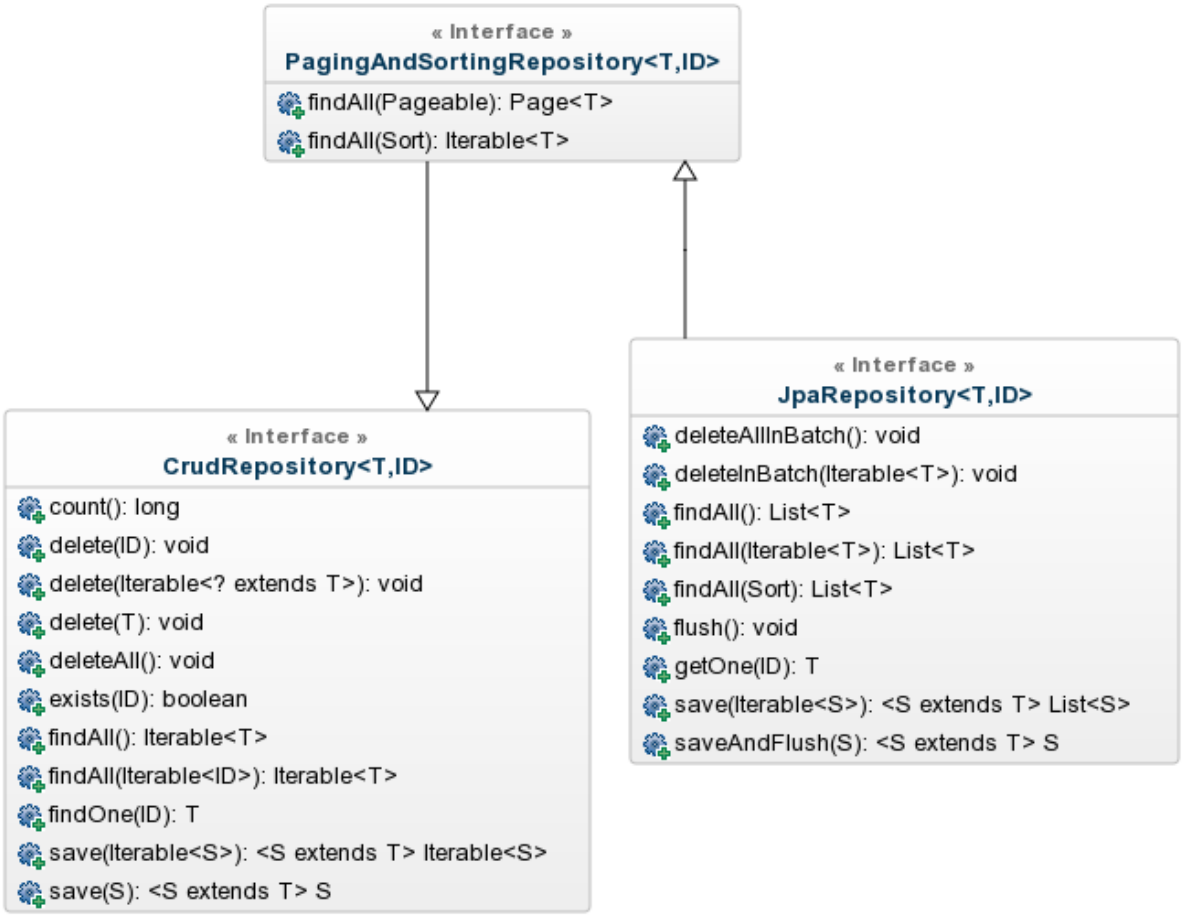


Figura 3-5. Diagrama de clases de Spring Data JPA

A continuación, se describen cada uno de los métodos:

Tabla 3–1 Métodos de la interfaz *JpaRepository<T,ID extends Serializable>*

Método	**Descripción**
deleteAllInBatch(): void	Elimina todas las entidades del tipo de repositorio (en una sóla operación)
deleteInBatch(Iterable<T>): void	Elimina todas las entidades asociadas a los objetos indicados (en una sóla operación)
findAll(): List<T>	Devuelve todas las entidades del tipo de repositorio
findAll(Iterable<ID>): List<T>	Devuelve todas las entidades con los ID indicados
findAll(Sort): List<T>	Recupera una lista ordenada de todas las entidades del tipo de destino
flush(): void	Finaliza todos los cambios pendientes en la base de datos
getOne(ID): T	Devuelve una referencia a la entidad con el ID indicado
save(Iterable<S>): <S extends T> List<S>	Guarda todas las entidades en el objeto *Iterable* indicado
saveAndFlush(S): <S extends T> S	Guarda la entidad indicada y hace un *flush()* instantáneamente

Tabla 3–2 Métodos de la interfaz *PagingAndSortingRepository<T,ID extends Serializable>*

Método	Descripción
findAll(Pageable): Page<T>	Recupera una lista paginada y ordenada de todas las entidades del tipo de destino
findAll (Sort): Iterable<T>	Recupera una lista ordenada de todas las entidades del tipo de destino

Tabla 3–3 Métodos de la interfaz *CrudRepository<T,ID extends Serializable>*

Método	Descripción
count(): long	Devuelve el número de entidades del tipo de repositorio
delete(ID): void	Elimina una entidad por su ID
delete(Iterable<? extends T>): void	Elimina todas las entidades asociadas a los objetos indicados
delete(T): void	Elimina una entidad asociada al objeto indicado
deleteAll(): void	Elimina todas las entidades del tipo de repositorio
exists(ID): boolean	Devuelve *true* si existe una entidad con el ID indicado
findAll(): Iterable<T>	Devuelve todas las entidades del tipo de repositorio
findAll(Iterable<ID>): Iterable<T>	Devuelve todas las entidades con los ID indicados
findOne(ID): T	Devuelve una única entidad con el ID indicado
save(Iterable<S>): <S extends T> Iterable<S>	Guarda todas las entidades en el objeto *Iterable* indicado
save(S): <S extends T> S	Guarda la entidad indicada

La convención a seguir para nombrar el repositorio/interfaz es utilizar el nombre de la entidad con el sufijo "Repository". Asimismo, hay que indicar la entidad para la que se crea el repositorio y el tipo de su clave primaria.

A continuación, se muestra el código de uno de los repositorios de la aplicación:

```
package com.sophistas.persistencia;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sophistas.dominio.ResponsableAlumno;

public interface ResponsableAlumnoRepository extends JpaRepository<ResponsableAlumno, Long>{

}
```

Los repositorios se han colocado todos en el paquete *com.sophistas.persistencia*. Para informar a Spring de la existencia de este paquete se hace uso de la anotación de clase *@EnableJpaRepositories*, del paquete *org.springframework.data.jpa.repository.config*.

Esta anotación se ha colocado en la clase de configuración *RootConfig*:

```
@EnableJpaRepositories("com.sophistas.persistencia")
```

3.3.1 Añadir métodos personalizados a un repositorio

Se ha visto que Spring Data ofrece numerosos métodos para las operaciones con las bases de datos más comunes. Pero normalmente estos no son suficientes. Para ello, Spring Data cuenta con varias formas de añadir métodos

personalizados a un repositorio:

- Usando una convención para el nombre del método definida por Spring Data. Al crear la implementación del repositorio, Spring Data examina los métodos de la interfaz del repositorio, analiza el nombre del método e intenta comprender su cometido en el contexto del objeto para persistencia. Básicamente, lo que hace Spring Data es definir una especie de lenguaje específico del dominio (DSL) en miniatura en el que los detalles de persistencia se expresan en las firmas de los métodos de repositorio.

  Un ejemplo de método sería:

  ```
  List<Tarifa> findTarifaByNombre(String nombre);
  ```

  Este método devolvería una lista con las tarifas cuyo nombre coincida con el que se ha pasado como parámetro.

  Los métodos de repositorio están formados por un verbo, un sujeto opcional, la palabra *By* y un predicado. En el caso del ejemplo, el verbo es *find*, el sujeto es *Tarifa* y el predicado es *Nombre*.

- Usando la anotación *@Query* (del paquete *org.springframework.data.jpa.repository*) para proporcionar a Spring Data la consulta que ejecutar, utilizando para ello el lenguaje JPQL.

  El método del ejemplo anterior quedaría así usando esta técnica:

  ```
  @Query("select tarifa from Tarifa tarifa where tarifa.nombre = :nombre")
  List<Tarifa> buscarPorNombre(@Param("nombre") String nombre);
  ```

  Los parámetros que recibe el método son referenciados en la consulta con el alias que se le asigna a través de la anotación *@Param* (del paquete *org.springframework.data.repository.query*).

  Esta segunda opción ha sido la que se ha empleado en la aplicación siempre que ha sido posible. (Cuando no, se tuvo que bajar de nivel y usar directamente el *EntityManager* "a la antigua usanza" como se indica a continuación.)

Existen ocasiones en las que se necesita una funcionalidad para el repositorio que no se puede describir con las convenciones de nomenclatura de métodos de Spring Data o ni siquiera con una consulta con la anotación *@Query*. En este caso es necesario escribir un método de repositorio usando directamente el *EntityManager*, es decir, trabajar con JPA a un nivel inferior al que ofrece Spring Data JPA.

Cuando Spring Data JPA genera la implementación de una interfaz de repositorio, también busca una clase que tenga el mismo nombre que la interfaz y con el sufijo *Impl*. Si la clase existe, Spring Data JPA fusiona sus métodos con los generados por Spring Data JPA.

Por ejemplo, para la interfaz *ProfesorRepository*, busca la clase *ProfesorRepositoryImpl*. A continuación, se muestra el código de ambas (y de la interfaz *IProfesorCustomRepository*), que constituyen el repositorio para la entidad Profesor de la aplicación:

- Interfaz *ProfesorRepository*

```
package com.sophistas.persistencia;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import com.sophistas.dominio.Profesor;

public interface ProfesorRepository extends IProfesorCustomRepository,
       JpaRepository<Profesor, Long> {

       @Query("select profesor from Profesor profesor where profesor.nif = :nif")
       List<Profesor> buscarPorNif(@Param("nif") String nif);
```

```
	@Query("select count(clase) from Clase clase join clase.horasSemanales horasSemanales
	where clase.profesor.id = :profesor and horasSemanales.id = :horaSemanal")
	Integer estaLibreProfesorEsaHora(@Param("profesor") Long profesor,
		@Param("horaSemanal") Long horaSemanal);

	@Query("select count(clase) from Clase clase join clase.horasSemanales horasSemanales
	where clase.profesor.id = :profesor and horasSemanales.id = :horaSemanal and
	clase.id != :idClase")
	Integer estaLibreProfesorEsaHora(@Param("profesor") Long profesor,
		@Param("horaSemanal") Long horaSemanal, @Param("idClase") Long idClase);

}
```

- Clase *ProfesorRepositoryImpl*

```
package com.sophistas.persistencia;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import org.springframework.stereotype.Repository;

import com.sophistas.dominio.Profesor;

@Repository
public class ProfesorRepositoryImpl implements IProfesorCustomRepository {

	@PersistenceContext
	protected EntityManager em;

	@SuppressWarnings("unchecked")
	@Override
	public List<Profesor> buscarPorNombrePorNif(String nombre, String nif) {

		Map<String, Object> parametros = new HashMap<String, Object>();

		StringBuilder sb = new StringBuilder();
		sb.append(" select profesor ");
		sb.append(" from Profesor profesor ");
		sb.append(" where ");
		sb.append(" 1 = 1 ");

		if (nif != null) {
			sb.append(" and :nif like profesor.nif ");
			parametros.put("nif", nif);
		}

		if (nombre != null) {
			String[] palabras = nombre.split("\\s+");

			sb.append(" and ( ");
			Integer i = 0;
			for (String palabra : palabras) {
				String nombrei = "nombre" + String.valueOf(i);
				if (i > 0) {
					sb.append(" or ");
				}
				sb.append(" profesor.apellido1 like concat('%',:" + nombrei +
					",'%') ");
				sb.append(" or ");
				sb.append(" profesor.apellido2 like concat('%',:" + nombrei +
					",'%') ");
```

```
                                sb.append(" or ");
                                sb.append(" profesor.nombre like concat('%',:" + nombrei +
                                ",'%') ");
                                parametros.put(nombrei, palabra);
                                i++;
                        }
                        sb.append(" ) ");
                }

                Query query = em.createQuery(sb.toString(), Profesor.class);

                for (String key : parametros.keySet()) {
                    query.setParameter(key, parametros.get(key));
                }

                return query.getResultList();
        }

}
```

Como se puede apreciar, *ProfesorRepositoryImpl* no implementa la interfaz *ProfesorRepository*. En su lugar, implementa *IProfesorCustomRepository*, mostrada a continuación:

```
package com.sophistas.persistencia;

import java.util.List;

import com.sophistas.dominio.Profesor;

public interface IProfesorCustomRepository {

        List<Profesor> buscarPorNombrePorNif(String nombre, String nif);

}
```

Es necesario declarar el método *buscarPorNombrePorNif()* en la interfaz *ProfesorRepository*. Para ello se ha hecho que *ProfesorRepository* amplíe *IProfesorCustomRepository* (como se ve en el código de *ProfesorRepository*).

4 SPRING MVC, LOS CONTROLADORES

Divide et impera.

Julio César

El marco de trabajo web de Spring, Spring MVC, está basado en el patrón Modelo-Vista-Controlador (MVC). Ayuda a crear aplicaciones basadas en la web que son flexibles y cuentan con acoplamiento débil, al igual que el propio marco de trabajo Spring.

En este capítulo se va a tratar sobre cómo se ha creado gran parte de la capa web de la aplicación, utilizando para ello Spring MVC.

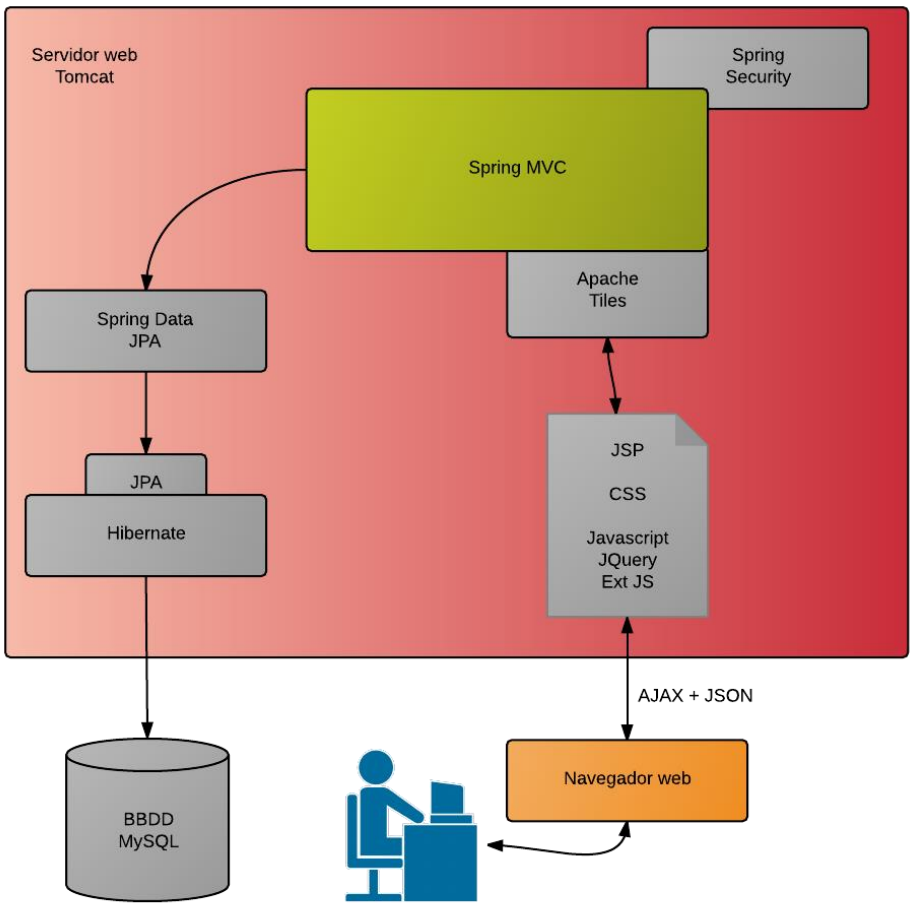


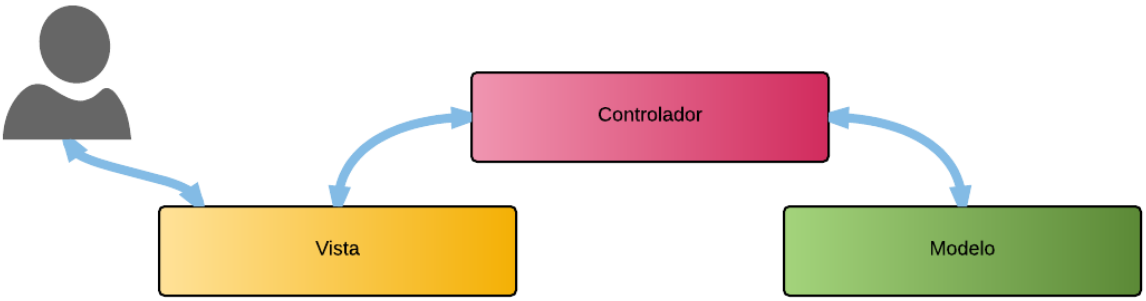
Figura 4-1. Arquitectura del sistema - Spring MVC

4.1 Patrón MVC

El patrón modelo–vista–controlador (MVC) [8] es un patrón de arquitectura de software que separa los diferentes aspectos de una aplicación. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado, define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

De manera genérica, los componentes de MVC se podrían definir como sigue:

- El Modelo: El Modelo es el objeto que representa los datos del programa. Maneja los datos y controla todas sus transformaciones. El Modelo no tiene conocimiento específico de los Controladores o de las Vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el Modelo y sus Vistas, y notificar a las Vistas cuando cambia el Modelo. (En el capítulo anterior se trató de forma exhaustiva este componente.)
- La Vista: La Vista es el objeto que maneja la presentación visual de los datos representados por el Modelo. Genera una representación visual del Modelo y muestra los datos al usuario. Interactúa con el Modelo a través de una referencia al propio Modelo. (En el próximo capítulo se tratará en detalle este componente.)
- El Controlador: Es el intermediario entre la vista y el modelo. Es quien controla las interacciones del usuario solicitando los datos al modelo y entregándolos a la vista para que ésta, lo presente al usuario, de forma "humanamente legible". (En el presente capítulo se tratará en detalle este componente.)



.

Figura 4-2. Patrón MVC

4.2 Ciclo de vida de una solicitud en Spring MVC

A continuación, se describe el ciclo de vida de una solicitud que, desde el cliente, recorre los componentes de Spring MVC [9] para generar, en última instancia, una respuesta que se devuelve al cliente.

Cada vez que un usuario hace clic en un enlace o envía un formulario en su navegador Web, se pone en marcha una solicitud.

La solicitud siempre está ocupada. Desde que deja el navegador hasta que vuelve con una respuesta realiza varias paradas, y en cada una deja información y recoge otra. En la siguiente figura se pueden observar todas las paradas que realiza la solicitud.

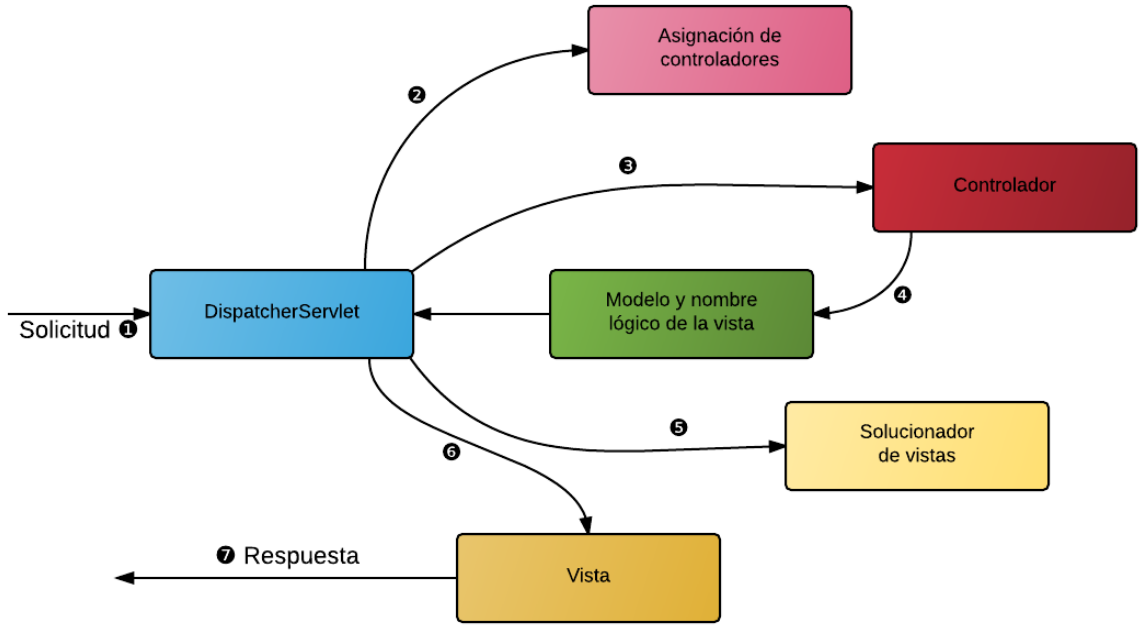


Figura 4-3. Ciclo de vida de una solicitud en Spring MVC

Cuando la solicitud abandona el navegador ❶, lleva información sobre lo que el usuario necesita. Como mínimo, la solicitud va a incluir la URL solicitada. Sin embargo, también puede llevar consigo información adicional, como por ejemplo la enviada por el usuario a través de un formulario.

La primera parada de la solicitud tiene lugar en *DispatcherServlet* de Spring. Spring MVC canaliza las solicitudes a través de este único *servlet*[7] controlador. Un controlador frontal es un patrón común de aplicación web en el que un único servlet delega la responsabilidad de una solicitud a otros componentes de la aplicación para llevar a cabo el procesamiento. En el caso de Spring MVC, *DispatcherServlet* sería el controlador frontal.

La tarea de *DispatcherServlet* es enviar la solicitud a un controlador de Spring MVC. Un controlador es un componente de Spring que procesa la solicitud. Sin embargo, una aplicación típica puede tener varios controladores y *DispatcherServlet* tiene que decidir cuál elegir. Para ello, consulta con una o más asignaciones de controlador para decidir cuál va a ser la siguiente parada de la solicitud ❷. La asignación del controlador va a prestar especial atención a la URL que transporta la solicitud a la hora de tomar la decisión.

Una vez se ha seleccionado un controlador adecuado, *DispatcherServlet* envía la solicitud en su camino hacia el controlador elegido ❸. En éste, la solicitud va a soltar su carga (la información enviada por el usuario) y a esperar mientras el controlador la procesa (en realidad, un controlador bien diseñado no lleva a cabo tareas de procesamiento y, en su lugar, delega la responsabilidad de la lógica de negocio a uno o más objetos de servicio).

A menudo, la lógica llevada a cabo por un controlador implica que parte de la información tiene que enviarse de nuevo al usuario y mostrarse en el navegador. Esta información recibe el nombre de modelo. Sin embargo, enviar información sin procesar al usuario no es suficiente: necesita un formato para que éste pueda consultarla. Para ello, a la información se le tiene que asignar una vista, normalmente una JSP.

Una de las últimas tareas que lleva a cabo un controlador es empaquetar los datos del modelo e identificar el nombre de la vista que debe generar el resultado. A continuación, envía la solicitud, junto con el modelo y el nombre de la vista, de vuelta a *DispatcherServlet* ❹.

Hasta el momento, el controlador no se acopla a ninguna vista y el nombre de la vista devuelto a *DispatcherServlet* no identifica una JSP específica (en concreto, ni siquiera tiene que sugerir que la vista sea una

[7] Un *servlet* es un componente web que se ejecuta dentro de un contenedor web y genera contenido dinámico que viaja al navegador web para ser visualizado.

JSP). En su lugar, solo cuenta con un nombre lógico que se va a utilizar para examinar la vista que va a generar el resultado. *DispatcherServlet* va a consultar a un solucionador de vistas para asignar el nombre de la vista lógica a una implementación de vista específica, que puede ser o no una JSP ❺.

Ahora que *DispatcherServlet* sabe qué vista va a procesar el resultado, el trabajo de la solicitud está prácticamente terminado. Su última parada es la implementación de la vista ❻, por lo general una JSP, donde entrega los datos del modelo. La vista va a utilizarlos para generar el resultado que el objeto de respuesta va a devolver al cliente ❼.

4.3 Configuración de Spring MVC

4.3.1 Dependencia Maven

Para utilizar Spring MVC se debe incluir la siguiente dependencia en el fichero POM:

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-webmvc</artifactId>
	<version>${spring-framework.version}</version>
</dependency>
```

4.3.2 Configurar *DispatcherServlet*

En el corazón de Spring MVC se encuentra *DispatcherServlet*, el primer contacto de la solicitud con el marco de trabajo, y responsable de dirigir la solicitud a través de los demás componentes.

En la clase *SophistasWebAppInitializer* (dentro del paquete de configuración *com.sophistas.configuracion*) se establece toda la configuración necesaria para el *DispatcherServlet*.

```
package com.sophistas.configuracion;

import javax.servlet.Filter;

import org.springframework.web.filter.CharacterEncodingFilter;
import org.springframework.web.servlet.support.
	AbstractAnnotationConfigDispatcherServletInitializer;

public class SophistasWebAppInitializer extends
	AbstractAnnotationConfigDispatcherServletInitializer {

	@Override
	protected Class<?>[] getRootConfigClasses() {
		return new Class<?>[] {RootConfig.class, SecurityConfig.class};
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class<?>[] {WebConfig.class};
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}

	@Override
	protected Filter[] getServletFilters() {
		CharacterEncodingFilter utf8Filter = new CharacterEncodingFilter();
		utf8Filter.setEncoding("UTF-8");
		utf8Filter.setForceEncoding(true);
		return new Filter[] { utf8Filter };
	}
```

```
}
```

Cualquier clase que amplíe *AbstractAnnotationConfigDispatcherServletInitializer* (como hace *SophistasWebAppInitializer*) se utilizará automáticamente para configurar *DispatcherServlet*.

La clase *SophistasWebAppInitializer* sobreescribe tres métodos abstractos:

- *getRootConfigClasses()*: Debe devolver un *array* de clases que configuren el contexto raíz. En nuestro caso este *array* contiene las clases *RootConfig* y *SecurityConfig*.
- *getServletConfigClasses()*: Debe devolver un *array* de clases que configuren el contexto *servlet*. En nuestro caso este *array* contiene la clase *WebConfig*.
- *getServletMappings()*: Identifica una o varias rutas al *DispatcherServlet* al que se va a asignar. En este caso se asigna a / para indicar que será el *servlet* predeterminado de la aplicación. Procesará todas las solicitudes que reciba la aplicación.

4.3.3 Dos contextos de aplicación

AbstractAnnotationConfigDispatcherServletInitializer crea tanto *DispatcherServlet* como *ContextLoaderListene*r.

Cuando se inicia *DispatcherServlet*, crea un contexto de aplicación de Spring y lo carga con los *bean* declarados en las clases de configuración que ha recibido. Con el método *getServletConfigClasses()* se le pide a ese *DispatcherServlet* que cargue su contexto de aplicación, el contexto *servlet*, con los *bean* definidos en la clase de configuración *WebConfig*.

Pero en las aplicaciones web de Spring hay otro contexto de aplicaciones, el contexto raíz, creado por *ContextLoaderListener*. Con el método *getRootConfigClasses()* se le pide a ese *ContextLoaderListener* que cargue su contexto de aplicación, el contexto raíz, con los *bean* definidos en la clase de configuración *RootConfig* y *SecurityConfig*.

Mientras que *DispatcherServlet* debe cargar los *bean* que contengan componentes Web como controladores, solucionadores de vistas y asignaciones de controlador, *ContextLoaderListener* debe cargar los demás *bean* de la aplicación. Estos *bean* suelen ser componentes del nivel intermedio y del nivel de datos.

Estos dos contextos de aplicación son del tipo *AnnotationConfigWebApplicationContext*.

El código de las tres clases de configuración (*RootConfig*, *WebConfig* y *SecurityConfig*) se muestra a continuación y se explica a lo largo de esta primera parte de la memoria.

- Clase *RootConfig*

```
package com.sophistas.configuracion;

import javax.persistence.EntityManagerFactory;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableJpaRepositories("com.sophistas.persistencia")
@EnableTransactionManagement
public class RootConfig {

	/*
	 * Bean donde se define la base de datos de la aplicacion
```

```
	 */
	@Bean(name = "dataSource")
	public DriverManagerDataSource dataSource() {
		DriverManagerDataSource driverManagerDataSource = new
		DriverManagerDataSource();
		driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver");
		driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/SOPHISTAS");
		driverManagerDataSource.setUsername("sophista");
		driverManagerDataSource.setPassword("S0ph1st4");
		return driverManagerDataSource;
	}

	/*
	 * Bean donde se define una fabrica de administradores de entidades
	 */
	@Bean
	public EntityManagerFactory entityManagerFactory() {
		HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
		vendorAdapter.setGenerateDdl(true);
		vendorAdapter.setShowSql(true);

		LocalContainerEntityManagerFactoryBean factory = new
		LocalContainerEntityManagerFactoryBean();
		factory.setJpaVendorAdapter(vendorAdapter);
		factory.setPackagesToScan("com.sophistas.dominio");
		factory.setDataSource(dataSource());
		factory.afterPropertiesSet();

		return factory.getObject();
	}

	/*
	 * Bean encargado de administrar las transacciones
	 */
	@Bean
	public PlatformTransactionManager transactionManager() {
		JpaTransactionManager txManager = new JpaTransactionManager();
		txManager.setEntityManagerFactory(entityManagerFactory());
		return txManager;
	}

}
```

- Clase *WebConfig*

```
package com.sophistas.configuracion;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.tiles3.TilesConfigurer;
import org.springframework.web.servlet.view.tiles3.TilesViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages={"com.sophistas.negocio", "com.sophistas.accion"})
public class WebConfig extends WebMvcConfigurerAdapter {

	/*
```

```
	 * Bean encargado de localizar y cargar definiciones de mosaico y de la
	 * coordinacion general de Tiles
	 */
	@Bean
	public TilesConfigurer tilesConfigurer() {
		TilesConfigurer tiles = new TilesConfigurer();
		tiles.setDefinitions(new String[] { "/WEB-INF/layout/tiles.xml" });
		tiles.setCheckRefresh(true);
		return tiles;
	}

	/*
	 * Bean encargado de resolver los nombres logicos de vistas en definiciones
	 * de mosaico
	 */
	@Bean
	public ViewResolver viewResolver() {
		return new TilesViewResolver();
	}

	/*
	 * Metodo para que DispatcherServlet dirija las solicitudes
	 * de recursos estaticos al servlet predeterminado del
	 * contenedor de servlet y que no las procese personalmente
	 */
	@Override
	public void configureDefaultServletHandling (
			DefaultServletHandlerConfigurer configurer) {
		configurer.enable();
	}

	/*
	 * Metodo para indicar como se van a localizar los recursos
	 * desde la peticion HTTP, y donde se van a encontrar los recursos
	 */
	@Override
	public void addResourceHandlers(ResourceHandlerRegistry registry) {
		registry.addResourceHandler("/estilo/**").addResourceLocations("/estilo/");
		registry.addResourceHandler("/ext/**").addResourceLocations("/ext/");
		registry.addResourceHandler("/fuentes/**").addResourceLocations("/fuentes/");
		registry.addResourceHandler("/imagenes/**")
			.addResourceLocations("/imagenes/");
		registry.addResourceHandler("/jquery/**").addResourceLocations("/jquery/");
		registry.addResourceHandler("/js/**").addResourceLocations("/js/");
	}

}
```

- Clase *SecurityConfig*

```
package com.sophistas.configuracion;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.encoding.Md5PasswordEncoder;
import org.springframework.security.config.annotation.authentication.
	builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.
	WebSecurityConfigurerAdapter;

@Configuration
```

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

	@Autowired
	DataSource dataSource;

	private static final String USERSBYUSERNAMEQUERY =
		"SELECT USUARIO, CLAVE, TRUE FROM USUARIO WHERE USUARIO = ?";
	private static final String AUTHORITIESBYUSERNAMEQUERY =
		"SELECT USUARIO, 'ROLE_USER' FROM USUARIO WHERE USUARIO = ?";

	/*
	 * Configura servicios de detalles de usuarios
	 */
	@Override
	protected void configure(AuthenticationManagerBuilder auth) throws Exception {
		auth.jdbcAuthentication()
			.dataSource(dataSource)
			.usersByUsernameQuery(USERSBYUSERNAMEQUERY)
			.authoritiesByUsernameQuery(AUTHORITIESBYUSERNAMEQUERY)
			.passwordEncoder(new Md5PasswordEncoder());
	}

	/*
	 * Configura la protección de las solicitudes por
	 * parte de los interceptores
	 */
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.authorizeRequests()
			.antMatchers("/login**").permitAll()
			.anyRequest().access("hasRole('ROLE_USER')")
		.and()
			.formLogin()
			.loginPage("/login")
			.defaultSuccessUrl("/home", true)
			.failureUrl("/login?error")
			.usernameParameter("usuario")
			.passwordParameter("clave")
		.and()
			.logout()
			.logoutSuccessUrl("/login?logout")
		.and()
			.csrf().disable();
	}

	/*
	 * Configura la secuencia de filtros de Spring Security
	 */
	@Override
	public void configure(WebSecurity web) throws Exception {
		web.ignoring().antMatchers("/estilo/**");
		web.ignoring().antMatchers("/ext/**");
		web.ignoring().antMatchers("/fuentes/**");
		web.ignoring().antMatchers("/imagenes/**");
		web.ignoring().antMatchers("/jquery/**");
		web.ignoring().antMatchers("/js/**");
	}

}
```

4.3.4 Habilitar Spring MVC

Para habilitar Spring MVC se usa la anotación *@EnableWebMvc* (del paquete

org.springframework.web.servlet.config.annotation). Esta anotación se ha añadido a la clase *WebConfig*.

4.4 Los controladores

El controlador es el objeto que proporciona significado a las órdenes del usuario, actuando sobre los datos representados por el Modelo.

Las tareas de un controlador consisten en:

- Interceptar las solicitudes HTTP del cliente.
- Traducir cada solicitud en una operación específica de negocio a ser realizada.
- Ayudar a seleccionar la siguiente vista a mostrar al cliente.
- Devolver la vista al cliente.

4.4.1 Definición de un controlador con *@Controller*

La anotación *@Controller* indica que una clase particular toma el papel de controlador.

@Controller es un estereotipo de anotación, basado en la anotación *@Component* (que se vió en el capítulo 2, en el apartado sobre inyección de dependencias). Su presencia está relacionada con el análisis de componentes. El analizador de componentes seleccionará automáticamente las clases anotadas con *@Controller* y las declarará como *bean* en el contexto de aplicaciones de Spring.

Se podría usar la anotación *@Component* y el efecto sería el mismo, pero no indicaría tan claramente el tipo de componente que es.

En la aplicación "Gestión de Sophistas Academia" los controladores se encuentran en el paquete *com.sophistas.accion*. En la clase *WebConfig* se ha indicado que este paquete debe ser analizado con la anotación *@ComponentScan*.

4.4.2 Declaración de solicitudes procesadas por un controlador

Para gestionar las diversas solicitudes del cliente, el controlador debe añadir los correspondientes métodos para cada tipo de solicitud.

Estos métodos deben ser anotados con *@RequestMapping* (del paquete *org.springframework.web.bind.annotation*). Los principales atributos de esta anotación son:

- *value*: El atributo *value* especifica la ruta de solicitud que procesará el método.
- *method*: El atributo *method* detalla el método HTTP que el método del controlador puede procesar.

A continuación, se muestra un ejemplo de controlador (muy similar a uno de la aplicación "Gestión de Sophistas Academia"):

```
package com.sophistas.memoria;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import com.sophistas.dominio.Tarifa;

@Controller
public class MantenimientoTarifasController {

	@RequestMapping(value="/mantenimientoTarifas", method=RequestMethod.GET)
	public String cargarMantenimientoTarifas() {
		// El nombre de la vista es mantenimientoTarifas
		return "mantenimientoTarifas";
```

```
        }

}
```

En este caso, siempre que llegara una solicitud GET HTTP para */mantenimientoTarifas*, se invocaría el método *cargarMantenimientoTarifas()*. Este método devuelve el valor de cadena "mantenimientoTarifas", que Spring MVC interpretará como el nombre de la vista que representar. *DispatcherServlet* pedirá al solucionador de vistas que resuelva este nombre de vista lógico en una vista real (este proceso se describe en el siguiente capítulo).

Nota: La anotación *@RequestMapping* puede ir también a nivel de clases. Siempre que en una clase de controlador hay un *@RequestMapping* de nivel de clase, se aplica a todos los métodos de control del controlador. Después, las anotaciones *@RequestMapping* en los métodos de control complementarán el *@RequestMapping* de nivel de clase.

4.4.3 Pasar datos del modelo a la vista

Cuando el controlador necesita pasar datos del modelo a la vista, no le es suficiente con devolver una cadena con el nombre lógico de la vista. En estos casos deberá retornar un objeto *ModelAndView*.

Como su nombre indica, un *ModelAndView* es una composición del modelo (lógica de datos) y de la vista. El modelo va a ser básicamente un mapa, es decir, una colección de pares de clave y valor.

A continuación, se muestra la clase *HomeController*, que es un controlador de la aplicación y contiene un método que devuelve un *ModelAndView*.

```
package com.sophistas.accion;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.sophistas.accion.interfaces.IHomeController;
import com.sophistas.negocio.interfaces.IEstadisticasManager;

@Controller
public class HomeController implements IHomeController {

	@Autowired
	private IEstadisticasManager estadisticasManager;

	@Override
	@RequestMapping(value={"/", "/home"}, method=RequestMethod.GET)
	public ModelAndView cargarHome() {
		// Resultado
		ModelAndView model = new ModelAndView("home");

		ObjectMapper mapper = new ObjectMapper();
		try {
			model.addObject("estadisticasEtapas", mapper.writeValueAsString
			 (estadisticasManager.calcularEstadisticasEtapas()));
			model.addObject("estadisticasAltasBajas", mapper.writeValueAsString
			 (estadisticasManager.calcularEstadisticasAltasBajas()));
		} catch (JsonProcessingException e) {
			// No se pueden obtener las graficas
		}

		// Devolvemos resultados
		return model;
	}

}
```

Se puede observar como se construye el objeto *ModelAndView* pasándole al constructor el nombre lógico de la vista "home":

```
ModelAndView model = new ModelAndView("home");
```

Y con el método *addObject()* se construye el modelo:

```
model.addObject("estadisticasEtapas",
	mapper.writeValueAsString(estadisticasManager.calcularEstadisticasEtapas()));
model.addObject("estadisticasAltasBajas",
	mapper.writeValueAsString(estadisticasManager.calcularEstadisticasAltasBajas()));
```

Los datos son "estadisticasEtapas" y "estadisticasAltasBajas", necesarios para representar la vista.

Aunque este capítulo no trate sobre las vistas, es necesario indicar aquí como accede la vista a estos datos. Si la vista es una JSP (como en la aplicación) los datos del modelo se copian en la solicitud como atributos de solicitud. Por lo tanto, se puede usar `${estadisticasEtapas}` y `${estadisticasAltasBajas}` desde la JSP para acceder a estos datos.

4.4.4 Cambiar dinámicamente la interfaz sin cambiar de vista

En las aplicaciones web "clásicas", el funcionamiento era como el visto hasta ahora. Cada comunicación con el servidor implicaba un cambio de página. Esto es, los controladores siempre devuelven un nombre lógico de vista (ya sea en *String* o en *ModelAndView*), esta vista se implementa y como resultado final se carga toda la página de nuevo en el navegador.

La gran mayoría de páginas web actuales no funcionan así. Casi todas usan AJAX[8] y Javascript en el cliente de manera intensiva, lo que permite comunicarse con el servidor y cambiar dinámicamente la interfaz sin cambiar de página, a no ser que fuera estrictamente necesario. En estos casos el método del controlador no va a devolver el nombre lógico de una vista (y el modelo si es el caso), sino un objeto Java con la información necesaria en el cliente.

Aquí entra en juego la conversión de mensajes de Spring. Un conversor de mensajes transforma un objeto devuelto por el controlador en una representación que entregar al cliente. Al usar la conversión de mensajes, *DispatcherServlet* no se preocupa de plasmar los datos del modelo en una vista. De hecho, no hay modelo y no hay vista. Solamente hay datos generados por el controlador y una representación de recursos generada cuando un conversor de mensajes transforma esos datos.

En estos métodos de controlador se usa la anotación *@ResponseBody* (del paquete *org.springframework.web.bind.annotation*). Esta anotación indica a Spring que se quiere enviar el objeto devuelto como recurso al cliente, convertido en un formato representacional que el cliente puede aceptar.

Por defecto, lo más sencillo en Spring es generar JSON[9]. Para ello, basta con incluir en el fichero POM la dependencia de la librería Jackson, una librería Java para convertir a/desde JSON que no es propia de Spring, pero con la que el *framework* está preparado para integrarse:

```
<dependency>
	<groupId>com.fasterxml.jackson.core</groupId>
	<artifactId>jackson-databind</artifactId>
	<version>${jackson.version}</version>
</dependency>
```

Un ejemplo (de la aplicación "Gestión de Sophistas Academia") sería:

```
@ResponseBody
@RequestMapping(value = "obtenerTarifasCombo")
public ResultadoObtenerTarifasComboDTO obtenerTarifasCombo() {

	ResultadoObtenerTarifasComboDTO resultado = new ResultadoObtenerTarifasComboDTO();
```

[8] AJAX (*Asynchronous JavaScript And XML*): Es una técnica de desarrollo web para crear aplicaciones que se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene una comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.
[9] JSON (*JavaScript Object Notation*): Es un formato de texto ligero para el intercambio de datos.

```
        List<Tarifa> lista = tarifaManager.listar();
        List<ComboTarifaDTO> listaDTO = new LinkedList<ComboTarifaDTO>();

        for (Tarifa tarifa : lista) {
                ComboTarifaDTO dto = new ComboTarifaDTO(tarifa);
                listaDTO.add(dto);
        }
        resultado.setTarifas(listaDTO);
        resultado.setNumeroResultados(listaDTO.size());
        resultado.setExito(true);
        resultado.setMensaje("OK");

        // Devolvemos resultados
        return resultado;

}
```

A este método de controlador se le va a llamar para cargar un combo con todas las tarifas del sistema.

Este método devuelve un objeto de la clase *ResultadoObtenerTarifasComboDTO*, la cual se muestra a continuación (esta clase realmente extiende *ResultadoDTO*, pero para facilitar el texto se han incluido en ella directamente todos las propiedades y métodos):

```
package com.sophistas.accion.dto;

import java.util.List;

public class ResultadoObtenerTarifasDTO {

        private static final long serialVersionUID = 1L;

        private Boolean exito;
        private String mensaje;
        private Integer numeroResultados;
        private List<ComboTarifaDTO> tarifas;

        public Boolean getExito() {
                return exito;
        }

        public void setExito(Boolean exito) {
                this.exito = exito;
        }

        public String getMensaje() {
                return mensaje;
        }

        public void setMensaje(String mensaje) {
                this.mensaje = mensaje;
        }

        public Integer getNumeroResultados() {
                return numeroResultados;
        }

        public void setNumeroResultados(Integer numeroResultados) {
                this.numeroResultados = numeroResultados;
        }

        public List<ComboTarifaDTO> getTarifas() {
                return tarifas;
        }

        public void setTarifas(List<ComboTarifaDTO> tarifas) {
                this.tarifas = tarifas;
        }
```

```
}
```

Las propiedades de esta clase son:

- *exito*: *true* o *false* dependiendo si la ejecución del controlador es exitosa o no.
- *mensaje*: Cadena que será "OK" si la ejecución es exitosa, o una descripción del error si la ejecución no lo es.
- *numeroResultados*: Número de resultados, esto es, número de tarifas que hay en la lista *tarifas*.
- *tarifas*: Lista de objetos de la clase *ComboTarifaDTO*, que representa cada opción del combo.

La clase *ComboTarifaDTO* se muestra a continuación:

```
package com.sophistas.accion.dto;

import java.io.Serializable;

import com.sophistas.dominio.Tarifa;

public class ComboTarifaDTO implements Serializable {

	private static final long serialVersionUID = 1L;

	private Long id;
	private String nombre;

	public ComboTarifaDTO() {
		super();
	}

	public ComboTarifaDTO(Tarifa tarifa) {
		this.id = tarifa.getId();
		this.nombre = tarifa.getNombre();
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getNombre() {
		return nombre;
	}

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

}
```

Si llega al servidor una solicitud para *obtenerTarifasCombo*, se recibirá una respuesta como ésta:

```
{
   "exito":true,
   "mensaje":"OK",
   "numeroResultados":4,
   "tarifas":[
      {
         "id":4,
         "nombre":"L"
      },
      {
         "id":3,
```

```
            "nombre":"M"
        },
        {
            "id":2,
            "nombre":"S"
        },
        {
            "id":1,
            "nombre":"XS"
        }
    ]
}
```

El conversor de mensajes ha convertido el objeto devuelto por el método del controlador en un documento JSON que se escribe en el cuerpo de la respuesta. En el lado del cliente, el Javascript toma el objeto enviado por Spring y lo usará para rellenar el combo con las tarifas devueltas[10].

4.4.5 Pasar datos a un método de controlador

Muchas aplicaciones web permiten al usuario enviar datos al servidor. Spring MVC ofrece varias formas para que un cliente pase datos a un método de controlador, entre estas formas destacan:

[10] Se muestra aquí como se ha rellenado el combo de tarifas usando la biblioteca Ext JS (que se describirá en el siguiente capítulo) haciendo una solicitud para *obtenerTarifasCombo*. Se define un almacén de tarifas y se carga con las tarifas:

```
// Almacen de tarifas
var readerInfoTarifas = Ext.create('Ext.data.JsonReader', {
        successProperty : 'exito',
        messageProperty : 'mensaje',
        totalProperty : 'numeroResultados',
        rootProperty : 'tarifas'
});

Ext.define('modeloInfoTarifas', {
        extend : 'Ext.data.Model',
        fields : [{
            name : 'id'
        }, {
            name : 'nombre'
        }],
        idProperty : 'id'
});

storeTarifas = Ext.create('Ext.data.Store', {
        autoDestroy : true,
        proxy: {
                type : 'ajax',
                url : 'obtenerTarifasCombo',
                reader : readerInfoTarifas,
        },
        model: modeloInfoTarifas,
        sorters: 'nombre'
});

// Se cargan las tarifas
storeTarifas.load();
```

Se define el elemento *tarifa* que es un combo en un formulario:

```
{
        xtype : 'combo',
        id : 'tarifa',
        name : 'tarifa',
        fieldLabel : '*Tarifa',
        editable : false,
        emptyText : 'Indique una tarifa',
        queryMode : 'local',
        store : storeTarifas,
        valueField : 'id',
        displayField : 'nombre'
}
```

- Variables de ruta (Patrones de plantilla URI)
- Parámetros de consulta
- Parámetros de formulario

4.4.5.1 Variables de ruta (Patrones de plantilla URI)

Las plantillas URI (*Uniform Resource Identifier*) se pueden utilizar para facilitar el acceso a determinadas partes de una URL en un método *@RequestMapping*.

Una plantilla URI es una cadena que contiene uno o más nombres de variables. Al sustituir los valores de estas variables, la plantilla se convierte en un URI. Las plantillas URI están definidas como un URI parametrizado. Por ejemplo, la plantilla URI "http://servidor/ consultaDetalleTarifa /{idTarifa}" contiene la variable *idTarifa*.

En Spring MVC puede utilizar la anotación *@PathVariable* (del paquete *org.springframework.web.bind.annotation*) en un argumento de un método de controlador para hacer que tome el valor de una variable de la plantilla URI:

```
@ResponseBody
@RequestMapping(value = "/consultaDetalleTarifa/{idTarifa}", method = RequestMethod.GET)
public ConsultaDetalleTarifaDTO consultaDetalleTarifa(
        @PathVariable("idTarifa") Long idTarifa) {

        // Resultado
        ConsultaDetalleTarifaDTO resultado = new ConsultaDetalleTarifaDTO();

        // Consultamos si existe la tarifa
        Tarifa tarifa = tarifaManager.buscarPorId(idTarifa);

        // Devolvemos el resutado
        resultado.setSuccess(true);
        resultado.setFormularioDetalleTarifaDTO(new FormularioDetalleTarifaDTO(tarifa));
        return resultado;

}
```

Por ejemplo, cuando llegue una petición para "/consultaDetalleTarifa/2", *idTarifa* tomará el valor 2.

(Este método para pasar datos a un método de controlador no se ha usado en la aplicación "Gestión de Sophistas Academia".)

4.4.5.2 Parámetros de consulta

Para pasar datos al método de controlador empleando esta forma se usa la anotación *@RequestParam* (del paquete *org.springframework.web.bind.annotation*). Con la notación *@RequestParam* se hace referencia a los parámetros de la petición, es decir, a los parámetros que se envían junto con la petición HTTP.

A continuación, se muestra un ejemplo (de la aplicación "Gestión de Sophistas Academia"):

```
@ResponseBody
@RequestMapping(value = "/consultaDetalleTarifa", method = RequestMethod.GET)
public ConsultaDetalleTarifaDTO consultaDetalleTarifa(
        @RequestParam(required = true, value = "idTarifa") Long idTarifa) {

        // Resultado
        ConsultaDetalleTarifaDTO resultado = new ConsultaDetalleTarifaDTO();

        // Consultamos si existe la tarifa
        Tarifa tarifa = tarifaManager.buscarPorId(idTarifa);

        // Devolvemos el resutado
        resultado.setSuccess(true);
        resultado.setFormularioDetalleTarifaDTO(new FormularioDetalleTarifaDTO(tarifa));
        return resultado;
```

```
}
```

Por ejemplo, cuando llegue una petición para "/consultaDetalleTarifa?idTarifa=2", *idTarifa* tomará el valor 2.

Se puede especificar si el parámetro es obligatorio o no con el elemento *required* de la anotación *@RequestParam*. Además, se puede indicar un valor por defecto para el parámetro si este no viene en la petición, para ello se usa el elemento *defaultValue*.

4.4.5.3 Parámetros de formulario

Este caso es más complejo ya que implica varios pasos:

- El usuario introduce los datos a través de un formulario HTML.
- Los datos se validan, y en caso de no ser correctos se vuelve a mostrar el formulario, indicando los errores de validación, para que el usuario pueda corregirlos.
- En caso de pasar la validación, los datos se "empaquetan" en un objeto Java para que el controlador pueda acceder a ellos de modo más sencillo que a través de la petición HTTP. La clase de este objeto tendrá tantas propiedades como campos tenga el formulario, y los nombres de estas propiedades serán los mismos que los nombres de los campos del formulario.
- El controlador se ejecuta, toma los datos, realiza la tarea y cede el control para que se muestre la vista.

A continuación, a modo de ejemplo para desarrollar este apartado, se muestra el formulario usado en la aplicación "Gestión de Sophistas Academia" para crear una tarifa nueva:

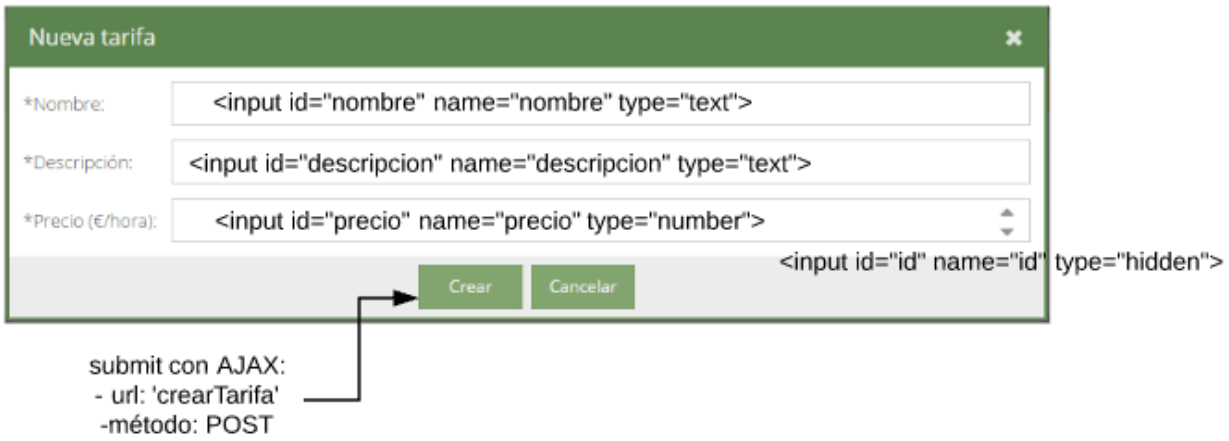


Figura 4-4. Formulario Nueva tarifa

Para crear una tarifa nueva, el usuario debe introducir el nombre de la tarifa, una descripción y el precio de ésta (en €/hora). Una vez introducidos debe pulsar el botón "Crear".

Se ha comentado que el siguiente punto era la validación de los datos. Antes de describir cómo se ha validado el formulario, se va a comentar los tipos de validaciones y dónde se pueden/deben hacer las validaciones.

Existen dos tipos de validaciones:

- "Validaciones simples": Los parámetros de validación están dentro del mismo código del cliente, por ejemplo, comprobar que un campo no esté vacío, esté bien formateado, …
- "Validaciones dinámicas": Los parámetros de validación no están dentro del mismo código del cliente, y sea necesario invocar a un servicio web o hacer una consulta a la base de datos; por ejmplo, comprobar que no haya NIF duplicados.

Las validaciones pueden hacerse en:

- El lado del cliente. Por motivos obvios, en el lado del cliente se hacen las "validaciones simples". No es obligatorio hacer validaciones en el lado del cliente, pero éstas evitan comunicaciones innecesarias con el servidor cuando el usuario introduce datos mal formateados, deja campos vacíos que son obligatorios…
- El lado del servidor. Aquí son obligatorias ambas validaciones. Lo más común es hacer primero las validaciones simples y luego las dinámicas que son más lentas (y no son necesarias si alguna "validación simple" no ha sido satisfactoria).

  Las simples hay que hacerlas en el lado del servidor obligatoriamente, aunque se hayan hecho también en el lado del cliente, ya que existen muchas formas de enviar peticiones de manera malintencionada directamente al servidor, sin pasar por la validación en el cliente.

Por tanto, tras pulsar el botón "Crear" se lleva a cabo la validación en el lado del cliente, antes de enviar los datos al servidor[11].

Si pasa esta primera validación, el formulario se envía al servidor empaquetado en un objeto Java de la clase *FormularioDetalleTarifaDTO* y es procesado por el método *crearTarifa()* de la clase controlador siguiente:

```
package com.sophistas.accion;

import java.util.LinkedList;
import java.util.List;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

[11] Se muestra aquí el método que se usa en el cliente para validar el formulario:

```
// Valida el formulario de creacion y edicion de tarifa
function validarFormularioDetalleTarifa() {
        var mensaje = '';
        var valido = true;

        // Se comprueba que se ha indicado un nombre
        if (Ext.getCmp('nombre').getValue() == null || Ext.getCmp('nombre').getValue().trim()
        == '') {
                mensaje = mensaje + 'Debe rellenar el campo \'Nombre\'.<br/>';
                valido = false;
        }

        // Se comprueba que se ha indicado una descripcion
        if (Ext.getCmp('descripcion').getValue() == null || Ext.getCmp('descripcion').getValue().trim()
        == '') {
                mensaje = mensaje + 'Debe rellenar el campo \'Descripción\'.<br/>';
                valido = false;
        }

        // Se comprueba que se ha indicado un precio
        if (Ext.getCmp('precio').getValue() == null) {
                mensaje = mensaje + 'Debe rellenar el campo \'Precio\'.<br/>';
                valido = false;
        }

        if (!valido) {
                mostrarMensajeErrorValidacion(mensaje);
        }

        return valido;
}
```

Comprueba que se han rellenado los tres campos, y si falta alguno se muestra al usuario el mensaje indicándoselo.

```
import org.springframework.web.bind.annotation.ResponseBody;

import com.sophistas.accion.dto.FormularioDetalleTarifaDTO;
import com.sophistas.accion.dto.ResultadoCreacionModificacionBorradoDTO;
import com.sophistas.accion.interfaces.IMantenimientoTarifasController;
import com.sophistas.negocio.interfaces.ITarifaManager;
import com.sophistas.negocio.validacion.FormularioDetalleTarifaDTOValidator;
import com.sophistas.utilidades.Utiles;

@Controller
public class MantenimientoTarifasController implements IMantenimientoTarifasController {

	private static final String[] CAMPOS_ERROR = {"nombre", "descripcion", "precio"};

	@Autowired
	private ITarifaManager tarifaManager;

	@Autowired
	private FormularioDetalleTarifaDTOValidator formularioDetalleTarifaDTOValidator;

	@InitBinder("formularioDetalleTarifaDTO")
	public void dataBinding(WebDataBinder binder) {
		binder.addValidators(formularioDetalleTarifaDTOValidator);
	}

	@Override
	@RequestMapping(value="/crearTarifa", method=RequestMethod.POST)
	@ResponseBody
	public ResultadoCreacionModificacionBorradoDTO crearTarifa(
			@Valid FormularioDetalleTarifaDTO formularioDetalleTarifaDTO,
			BindingResult bindingResult) {

		// Se validan los datos que vienen del formulario
		if (bindingResult.hasErrors()) {
			return Utiles.obtenerResultadoSiErrorValidacion(bindingResult,
				CAMPOS_ERROR);
		}

		// Se inserta la tarifa en la BBDD
		tarifaManager.crearTarifa(formularioDetalleTarifaDTO);

		ResultadoCreacionModificacionBorradoDTO resultado = new
			ResultadoCreacionModificacionBorradoDTO();
		resultado.setSuccess(true);
		resultado.setMensaje("");
		return resultado;

	}

	// Resto de la clase

}
```

Referente a la validación se observa en el método:

- La anotación *@Valid* del paquete (*javax.validation*) en el parámetro *formularioDetalleTarifaDTO*. Esta anotación indica a Spring que el objeto tiene limitaciones de validación que tiene que aplicar.
- El parámetro *bindingResult* de la clase *org.springframework.validation.BindingResult*. A través de este parámetro es accesible el resultado de la validación.

En el servidor, se van a tener que hacer, como se comentó anteriormente, tanto las "validaciones simples" como las "validaciones dinámicas".

Validaciones simples (en el lado del servidor)

Para estas validaciones se va a aprovechar la compatibilidad de Spring con el API de validación de Java [10]. Spring admite este API en Spring MVC. No se necesita configuración adicional para que funcione, basta con asegurarse de incluir una implementación del API de Java, como Hibernate Validator [11].

Para ello se incluye en el fichero POM la siguiente dependencia:

```
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-validator</artifactId>
	<version>${hibernate-validator.version}</version>
</dependency>
```

El API de validación de Java define varias anotaciones que se pueden añadir a las propiedades de la clase del objeto empaquetado que se pasa al método de controlador para limitar sus valores.

A continuación, como ejemplo, se muestra la clase *FormularioDetalleTarifaDTO*. Se puede observar como coinciden el nombre de las propiedades con el nombre de los campos del formulario:

```
package com.sophistas.accion.dto;

import java.io.Serializable;

import javax.validation.GroupSequence;
import javax.validation.constraints.NotNull;

import org.hibernate.validator.constraints.NotBlank;

import com.sophistas.dominio.Tarifa;
import com.sophistas.negocio.validacion.NoValido;
import com.sophistas.negocio.validacion.Vacio;

@GroupSequence({ Vacio.class, NoValido.class, FormularioDetalleTarifaDTO.class})
public class FormularioDetalleTarifaDTO implements Serializable {

	private static final long serialVersionUID = 1L;

	private Long id;

	@NotBlank(message="Debe rellenar el campo 'Nombre'.", groups=Vacio.class)
	private String nombre;

	@NotBlank(message="Debe rellenar el campo 'Descripción'.", groups=Vacio.class)
	private String descripcion;

	@NotNull(message="Debe rellenar el campo 'Precio'.", groups=Vacio.class)
	private Double precio;

	public FormularioDetalleTarifaDTO() {
		super();
	}

	public FormularioDetalleTarifaDTO(Tarifa tarifa) {
		this.id = tarifa.getId();
		this.nombre = tarifa.getNombre();
		this.descripcion = tarifa.getDescripcion();
		this.precio = tarifa.getPrecio();
	}

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getNombre() {
```

```
		return nombre;
	}

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

	public String getDescripcion() {
		return descripcion;
	}

	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}

	public Double getPrecio() {
		return precio;
	}

	public void setPrecio(Double precio) {
		this.precio = precio;
	}

}
```

Se han usado las anotaciones *@NotBlank* en las propiedades nombre y descripcion y *@NotNull* en la propiedad precio. *@NotNull* es propia del API de validación de Java y *@NotBlank* la ofrece adicionalmente la implementación Hibernate Validator. También es posible definir limitaciones propias.

En la siguiente tabla se muestran las anotaciones proporcionadas por el API de validación de Java:

Tabla 4–1 Anotaciones proporcionadas por el API de validación de Java

Anotación	Descripción
@AssertFalse	El elemento anotado debe ser un tipo *Booleano* y *false*.
@AssertTrue	El elemento anotado debe ser un tipo *Booleano* y *true*.
@DecimalMax	El elemento anotado debe ser un número con un valor menor o igual al valor *BigDecimalString* proporcionado.
@DecimalMin	El elemento anotado debe ser un número con un valor mayor o igual al valor *BigDecimalString* proporcionado.
@Digits	El elemento anotado debe ser un número cuyo valor tenga el número de dígitos especificado.
@Future	El valor del elemento anotado debe ser una fecha futura.
@Max	El elemento anotado debe ser un número con un valor menor o igual al valor proporcionado.
@Min	El elemento anotado debe ser un número con un valor mayor o igual al valor proporcionado.
@NotNull	El valor del elemento anotado no puede ser *null*.
@Null	El valor del elemento anotado debe ser *null*.
@Past	El valor del elemento anotado debe ser una fecha pasada.
@Pattern	El valor del elemento anotado debe coincidir con la expresión regular proporcionada.
@Size	El valor del elemento anotado debe ser una cadena, una colección o una matriz cuya longitud coincida con el intervalo proporcionado.

Con estas anotaciones en la clase *FormularioDetalleTarifaDTO* y la anotación *@Valid* en el parámetro del método de controlador se cubren las "validaciones simples". El resultado irá en el parámetro *bindingResult* del método de controlador y se describirá más adelante.

Validaciones dinámicas

Para estas validaciones se ha implementado un tipo de servicio validador (uno por formulario). El validador va a ser una clase que implemente la interfaz *Validator* de Spring. Esta interfaz obliga a implementar dos métodos: *supports()* y *validate()*.

El primero de ellos establece si el validador puede validar instancias de la clase sumimistrada.

El método *validate()* es el que va a llevar a cabo todas las validaciones que se consideren necesarias para el formulario (serán las "validaciones dinámicas" aunque también se podrían incluir aquí "validaciones simples").

El validador que se ha implementado para el formulario que se está tratando es:

```
package com.sophistas.negocio.validacion;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

import com.sophistas.accion.dto.FormularioDetalleTarifaDTO;
import com.sophistas.negocio.interfaces.ITarifaManager;

@Service[12]
public class FormularioDetalleTarifaDTOValidator implements Validator {

	@Autowired
	private ITarifaManager tarifaManager;

	@Override
	public boolean supports(Class<?> clase) {
		return FormularioDetalleTarifaDTO.class.isAssignableFrom(clase);
	}

	@Override
	public void validate(Object target, Errors errores) {

		FormularioDetalleTarifaDTO formularioDetalleTarifaDTO =
			(FormularioDetalleTarifaDTO) target;

		// Se comprueba que no exista ya la tarifa
		if (tarifaManager.existeTarifa(formularioDetalleTarifaDTO.getId(),
				formularioDetalleTarifaDTO.getNombre().trim().toUpperCase())) {
			errores.reject("tarifaDuplicada", "Ya existe la tarifa introducida.");
		}

	}

}
```

Este validador comprueba que no haya ya una tarifa almacenada con el nombre que se está pasando en el formulario. Hay que hacer una búsqueda para ello en la base de datos, y por eso es una "validación dinámica".

Este validador debe registrarse en la clase controlador, y para ello se usa el método *initBinder()*. Se repite aquí el fragmento del controlador donde se registra el validador:

```
@Autowired
private FormularioDetalleTarifaDTOValidator formularioDetalleTarifaDTOValidator;

@InitBinder("formularioDetalleTarifaDTO")
public void dataBinding(WebDataBinder binder) {
	binder.addValidators(formularioDetalleTarifaDTOValidator);
}
```

El resultado de la validación irá en el parámetro *bindingResult* del método de controlador y se describirá a

[12] La anotación *@Service* del *paquete org.springframework.stereotype* indica que la clase toma el papel de servicio. Es un estereotipo de anotación, basado en la anotación *@Component*.

continuación.

Resultado de la validación, *BindingResult*

La clase *BindingResult* tiene métodos para averiguar qué errores se han producido. A continuación, se muestran algunos métodos destacados de ésta:

Tabla 4–2 Métodos de *BindingResult*

Método	Descripción
getAllErrors(): List<ObjectError>	Devuelve una lista con todos los errores, tanto globales como de campo.
getErrorCount(): int	Devuelve el número total de errores
getFieldErrorCount(): int	Devuelve el número de errores de campo
getFieldErrors(): List<FieldError>	Devuelve una lista con todos los errores de campo
getGlobalErrorCount():int	Devuelve el número de errores globales
getGlobalErrors(): List<ObjectError>	Devuelve una lista con todos los errores globales
hasErrors(): void	Devuelve si hubo algún error
hasFieldErrors(): void	Devuelve si hubo algún error de campo
hasGlobalErrors(): void	Devuelve si hubo algún error global

Un error de campo se refiere a un error en el valor de un campo específico (por ejemplo, que el campo *nombre* venga vacío). Un error global se refiere a un valor por el que se rechaza el objeto entero (por ejemplo, que la tarifa que se va a almacenar ya esté guardada).

Al principio del método de controlador se comprueba si el formulario viene con errores. Se repite a continuación el fragmento del código:

```
// Se validan los datos que vienen del formulario
if (bindingResult.hasErrors()) {
	return Utiles.obtenerResultadoSiErrorValidacion(bindingResult, CAMPOS_ERROR);
}
```

Si hay errores se llama al método *obtenerResultadoSiErrorValidacion()* de la clase *Utiles* que se muestra a continuación:

```
public static ResultadoCreacionModificacionBorradoDTO obtenerResultadoSiErrorValidacion
	 (BindingResult bindingResult, String[] camposError) {
	ResultadoCreacionModificacionBorradoDTO resultado = new
		ResultadoCreacionModificacionBorradoDTO();
	resultado.setSuccess(false);
	resultado.setTipoError("Validacion");
	StringBuilder mensaje = new StringBuilder();
	// Se consulta si hay errores de campo
	if(bindingResult.getFieldErrorCount() > 0) {
		HashMap<String, String> mapaErrores = new HashMap<String, String>();
		for(FieldError error : bindingResult.getFieldErrors()) {
			mapaErrores.put(error.getField(), error.getDefaultMessage());
		}
		for(String campo : camposError) {
			if (mapaErrores.containsKey(campo)) {
				mensaje.append(mapaErrores.get(campo)).append("<br/>");
			}
		}
	}
	// Si no hay errores de campo se muestran los errores globales
	else if(bindingResult.getGlobalErrorCount() > 0) {
		for(ObjectError error: bindingResult.getGlobalErrors()) {
			mensaje.append(error.getDefaultMessage()).append("<br/>");
		}
```

```
        }

        resultado.setMensaje(mensaje.toString());
        return resultado;
}
```

En primer lugar, se comprueban los errores de campo y se devuelven si hay. Si no, se comprueban los errores globales.

5 VISTAS

Sólo se ve bien con el corazón; lo esencial es invisible a los ojos.

Antoine de Saint-Exupéry

Las vistas son las encargadas de la representación de los datos, contenidos en el modelo, al usuario.

La vista sólo necesita la información requerida del modelo para realizar un despliegue. Cada vez que se realiza una actuación, que implica una modificación del modelo de dominio, la vista cambia a través de notificaciones generadas por el modelo de la aplicación.

Sencillamente, es la representación visual del modelo que redibuja las partes necesarias cuando se produce una modificación del mismo.

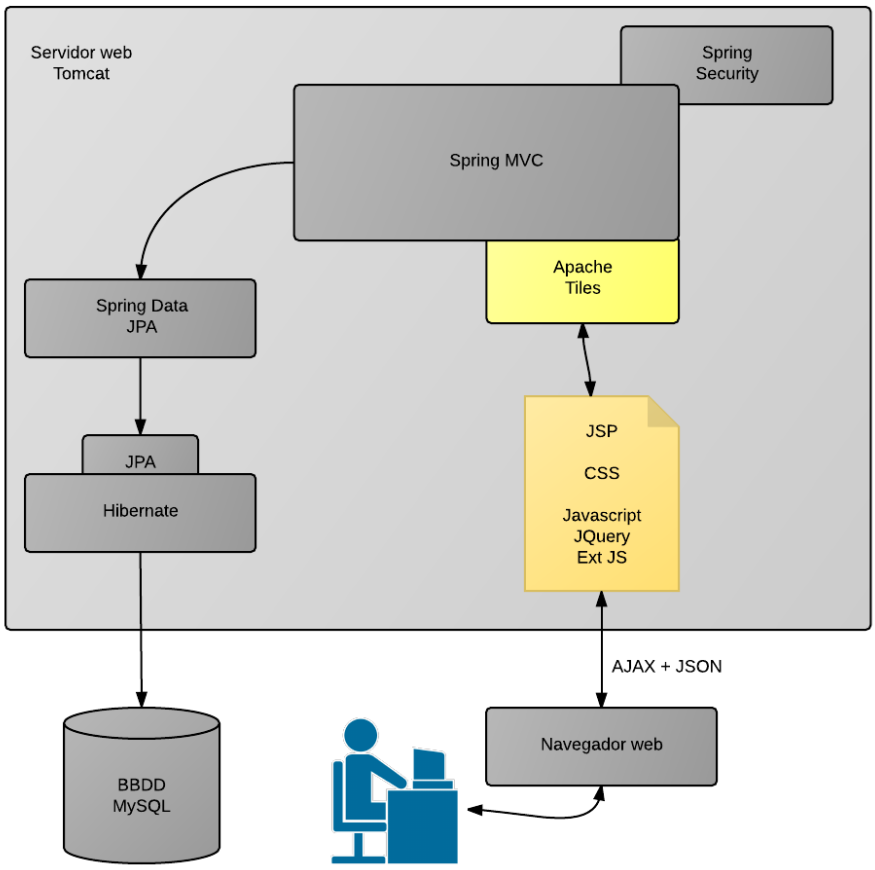


Figura 5-1. Arquitectura del sistema – Vista

5.1 Resolución de vistas

Recordando del capítulo anterior, los últimos pasos del ciclo de vida de una solicitud en Spring MVC eran:

- El controlador empaqueta los datos del modelo e identifica el nombre de la vista que debe generar el resultado.
- El controlador envía la solicitud, junto con el modelo y el nombre de la vista, de vuelta a *DispatcherServlet*. (El controlador no se acopla a ninguna vista y el nombre de la vista devuelto a *DispatcherServlet* no identifica un fichero específico, normalmente un .jsp)
- *DispatcherServlet* va a consultar a un solucionador de vistas para asignar el nombre de la vista lógica a una implementación de vista específica.
- La última parada de la solicitud es la implementación de la vista, donde *DispatcherServlet* entrega los datos del modelo. La vista va a utilizarlos para generar el resultado que el objeto de respuesta va a devolver al cliente.

Para la resolución de vistas es por tanto necesario configurar un solucionador de vistas, que le sirva a Spring para traducir nombres lógicos de vistas en implementaciones de vistas, que en la presente aplicación van a ser páginas JSP.

El MVC de Spring define la interfaz *ViewResolver*, que tiene este aspecto:

```
public interface ViewResolver {
	View resolveViewName(String viewName, Locale locale) throws Exception;
}
```

Cuando se asigna un nombre de vista y un *Locale* al método *resolveViewName()*, devuelve una instancia *View*, otra interfaz que tiene este aspecto:

```
public interface View {
	String getContentType();
	void render(Map<String, ?> model, HttpServletRequest request,
	HttpServletResponse response) throws Exception;
}
```

La labor de la interfaz *View* es aceptar el modelo, además de la solicitud de *servlet* y los objetos de respuesta, y representar un resultado en la respuesta.

Aunque es posible crear implementaciones propias de *ViewResolver* y *View*, y en algunos casos concretos es la solución adecuada, por lo general no es necesario. Spring ofrece diversas implementaciones que sirven para la mayoría de los casos.

A continuación, se nombran las más relevantes usadas por la mayoría de desarrolladores Java.

Tabla 5–1 Solucionadores de vista

Solucionador de vista	Descripción
InternalResourceViewResolver	Resuelve vistas como recursos internos a la aplicación web (por lo general JSP)
TilesViewResolver	Resuelve vistas como definiciones de Apache Tiles, en las que el ID del mosaico es igual que el nombre de la vista.

En la presente aplicación se va a hacer uso del solucionador *TilesViewResolver,* que se describe en el siguiente apartado.

5.2 *TilesViewResolver*, vistas Apache Tiles

Apache Tiles [12] es un motor de diseño que permite a los desarrolladores definir fragmentos de páginas que pueden ser ensamblados para formar una página completa en tiempo de ejecución.

Estos fragmentos o mosaicos, pueden ser usados simplemente para reducir la duplicación de elementos comunes o embebidos en otros fragmentos, para así desarrollar una serie de plantillas reusables. Estas plantillas dinamizan todo el desarrollo de toda una aplicación.

Como en la aplicación "Gestión de Sophistas Academia" todas las páginas tienen un encabezado y un pie de página comunes se ha optado por el uso de este motor de diseño, con el que se ha definido un diseño de página común para aplicarlo a todas las páginas.

Para usar Apache Tiles con Spring se necesita configurar un par de *bean*:

- *TilesConfigurer*: Encargado de localizar y cargar definiciones de mosaico y de la coordinación general de Tiles.
- *TilesViewResolver*: Encargado de resolver nombres lógicos de vista en definiciones de mosaico.

Para ello en la clase *WebConfig* (*com.sophistas.configuracion*) del paquete se han añadido las siguientes líneas:

```
/*
 * Bean encargado de localizar y cargar definiciones de mosaico y de la
 * coordinacion general de Tiles
 */
@Bean
public TilesConfigurer tilesConfigurer() {
	TilesConfigurer tiles = new TilesConfigurer();
	tiles.setDefinitions(new String[] { "/WEB-INF/layout/tiles.xml" });
	tiles.setCheckRefresh(true);
	return tiles;
}

/*
 * Bean encargado de resolver los nombres logicos de vistas en definiciones
 * de mosaico
 */
@Bean
public ViewResolver viewResolver() {
	return new TilesViewResolver();
}
```

Al configurar *TilesConfigurer*, la propiedad más importante es *definitions*. Acepta una matriz de cadenas en la que cada entrada especifica la ubicación de los archivos XML de definición de mosaicos. En este caso, la definición de mosaicos se encuentra en el archivo tiles.xml del directorio /WEB-INF/layout/.

La configuración de *TilesViewResolver* es una definición de *bean* muy básica, sin propiedades que configurar.

Apache Tiles cuenta con una definición de tipo de documento (DTD) para especificar definiciones de mosaico en un archivo XML. A continuación, se muestra el fichero tiles.xml que se ha usado:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
	"-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
	"http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>

	<definition name="base"
		template="/WEB-INF/layout/page.jsp">
		<put-attribute name="menu"
			value="/WEB-INF/layout/menu.jsp" />
		<put-attribute name="seccioninferior"
			value="/WEB-INF/layout/seccioninferior.jsp" />
	</definition>
```

```
	<definition name="login" extends="base">
		<put-attribute name="content"
			value="/login.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Identificación" />
		<put-attribute name="javascript"
			value="js/login.js" />
	</definition>

	<definition name="home" extends="base">
		<put-attribute name="content"
			value="/home.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Inicio" />
		<put-attribute name="javascript"
			value="js/home.js" />
	</definition>

	<definition name="profesores" extends="base">
		<put-attribute name="content"
			value="/profesores.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Profesorado" />
		<put-attribute name="javascript"
			value="js/profesores.js" />
	</definition>

	<definition name="alumnos" extends="base">
		<put-attribute name="content"
			value="/alumnos.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Alumnado" />
		<put-attribute name="javascript"
			value="js/alumnos.js" />
	</definition>

	<definition name="clases" extends="base">
		<put-attribute name="content"
			value="/clases.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Clases" />
		<put-attribute name="javascript"
			value="js/clases.js" />
	</definition>

	<definition name="mantenimientoCursos" extends="base">
		<put-attribute name="content"
			value="/mantenimientoCursos.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Mantenimiento de Cursos" />
		<put-attribute name="javascript"
			value="js/mantenimientoCursos.js" />
	</definition>

	<definition name="mantenimientoAsignaturas" extends="base">
		<put-attribute name="content"
			value="/mantenimientoAsignaturas.jsp" />
		<put-attribute name="title"
			value="SOPHISTAS - Mantenimiento de Asignaturas" />
		<put-attribute name="javascript"
			value="js/mantenimientoAsignaturas.js" />
	</definition>

	<definition name="mantenimientoTarifas" extends="base">
		<put-attribute name="content"
```

```
                value="/mantenimientoTarifas.jsp" />
            <put-attribute name="title"
                value="SOPHISTAS - Mantenimiento de Tarifas" />
            <put-attribute name="javascript"
                value="js/mantenimientoTarifas.js" />
        </definition>

</tiles-definitions>
```

Cada definición está formada por un elemento `<definition>` que suele tener uno o varios elementos `<put-attribute>`.

Cada elemento `<definition>` define un mosaico que, en última instancia, hace referencia a una plantilla JSP. En el caso del mosaico "base", la plantilla a la que se hace referencia se encuentra en /WEB-INF/layout/page.jsp.

Un mosaico también puede hacer referencia a otras plantillas JSP que incrustar en la principal. Por ejemplo, el mosaico "base", que hace referencia a una plantilla JSP de menú y a otra de sección inferior.

A continuación, se muestra la plantilla page.jsp a la que hace referencia el mosaico "base".

```
<!DOCTYPE HTML>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>

<html>
        <head>
                <meta charset="UTF-8">
                <meta http-equiv="X-UA-Compatible" content="IE=edge">
                <meta name="viewport" content="width=device-width, initial-scale=1,
                        maximum-scale=1, user-scalable=no">
                <link rel="icon" type="image/png" href="imagenes/buho_sophistas_icono.png" />
                <jsp:include page="scripts.jsp" />
                <script type="text/javascript" src="<tiles:insertAttribute
                        name="javascript"/>"></script>
                <title><tiles:insertAttribute name="title"/></title>
        </head>
        <body>
                <div id="bg">
                        <div id="page">
                                <tiles:insertAttribute name="menu" />
                                <tiles:insertAttribute name="content" />
                                <div> </div>
                        </div>
                        <tiles:insertAttribute name="seccioninferior" />
                </div>
        </body>
</html>
```

Cabe destacar cómo se usa la etiqueta JSP `<tiles:insertAttribute>` de la bibliotecta de etiquetas Tiles para insertar los atributos:

- *javascript*: ruta al fichero que contiene el código Javascript usado específicamente en la página.
- *title:* título de la página
- *menu*: plantilla JSP que contiene el menú de la aplicación
- *content*: plantilla JSP con el contenido específico de la página
- *seccioninferior*: plantilla JSP que contiene la sección inferior de la página

Los atributos *menu* y *seccioninferior* se establecen en la definición del mosaico "base" para apuntar a /WEB-INF/layout/menu.jsp y /WEB-INF/layout/seccioninferior.jsp respectivamente.

El mosaico "base" no se utiliza de forma independiente. Sirve de definición base (de ahí su nombre) que el resto de definiciones de mosaico amplían. Esto significa que heredan los valores de los atributos *menu* y *seccioninferior* (aunque podrían haberlos reemplazado), pero también establecen valores para el resto de

atributos (*javascript*, *title* y *content*).

```
<definition name="base" template="/WEB-INF/layout/page.jsp">
	<put-attribute name="menu" value="/WEB-INF/layout/menu.jsp" />
	<put-attribute name="seccioninferior" value="/WEB-INF/layout/seccioninferior.jsp" />
</definition>
```

A continuación, se muestra una de las definiciones que hereda del mosaico "base":

```
<definition name="home" extends="base">
	<put-attribute name="content" value="/home.jsp" />
	<put-attribute name="title" value="SOPHISTAS - Inicio" />
	<put-attribute name="javascript" value="js/home.js" />
</definition>
```

Se observa que el atributo *content* apunta a /home.jsp, el atributo *title* toma el valor "SOPHISTAS - Inicio" y el atributo *javascript* apunta a js/home.js.

5.3 JSP, JavaServer Pages

En la aplicación "Gestión de Sophistas Academia", se ha usado JSP para la tecnología de vista.

JSP [13] es un acrónimo de *JavaServer Pages*, que en castellano vendría a decir algo como Páginas de Servidor Java. Es, pues, una tecnología orientada a crear páginas web con programación en Java.

Una página JSP es una página (X)HTML[13] que incorpora ciertos elementos dinámicos: etiquetas especiales y pequeños fragmentos de código.

El código HTML aparece a la salida sin modificaciones.

Los elementos dinámicos se evalúan o ejecutan en el servidor en el momento de construcción de la respuesta.

La especificación JSP fue desarrollada mediante una iniciativa industrial de Sun Microsystems.

5.4 Otras tecnologías de cliente

5.4.1 JavaScript

JavaScript [14] es un lenguaje de programación interpretado orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. JavaScript permite mejoras en la interfaz de usuario y aporta dinamismo a las páginas web que lo incluyen.

El navegador es el encargado de interpretar las instrucciones Javascript y ejecutarlas para realizar estos efectos e interactividades.

5.4.2 jQuery

jQuery [15] es una biblioteca de código abierto escrita en JavaScript que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM[14], manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web.

jQuery ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código, es decir, con las funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio.

[13] HTML (*HyperText Markup Language*): Lenguaje de marcado para la elaboración de páginas web especificado por el *World Wide Web Consortium* (W3C).
[14] DOM (*Document Object Model*): Define la manera en que objetos y elementos se relacionan entre sí en el navegador y en el documento.

5.4.3 Ext JS

Ext JS [16] es una biblioteca de Javascript desarrollada por Sencha que incluye:

- Componentes UI de alto rendimiento y personalizables
- Modelo de componentes extensible
- Un API fácil de usar
- Licencias *Open Source* y comercial

Ext JS ofrece al desarrollador un gran conjunto de componentes (como *grid*, ventanas de diálogo, …) plenamente integrados y un API para conseguir interfaces web dinámicas e interactivas con el usuario.

Ext JS utiliza implícitamente tecnologías web 2.0 agregando características como las siguientes:

- Compatibilidad. Es compatible con los navegadores más importantes del mercado.
- Comunicación asíncrona (AJAX).
- DOM. Maximiza la capacidad de agregar, editar, cambiar, eliminar elementos del árbol DOM de manera dinámica.
- Efectos visuales. Utilizando la manipulación de los elementos, se pueden crear efectos visuales y animaciones.
- Almacenamiento del lado del cliente. Permite a las aplicaciones web guardar datos del lado del cliente.
- Manejo JSON.
- Manejo de Eventos. Permite reaccionar de una manera u otra dependiendo de las acciones del usuario.

5.4.4 CSS

Las hojas de estilo en cascada, conocidas como CSS [17], especificado por el W3C, es un lenguaje utilizado para escribir la información referente al estilo de presentación de la estructura del documento HTML (como tamaño, color, fondo, bordes, ...).

A pesar de que cada navegador garantiza estilos por defecto para cada uno de los elementos HTML, estos estilos no necesariamente satisfacen los requerimientos. Por ello, se debe aplicar los propios estilos para obtener la organización y el efecto visual deseados.

El mejor procedimiento para utilizar CSS consiste en escribir la información de presentación en un fichero con extensión .css y a continuación declarar una etiqueta que lo incluya en la cabecera de los ficheros HTML en los que sea necesario aplicar sus propiedades, y así se ha hecho en la aplicación.

6 SEGURIDAD: SPRING SECURITY

Uncertainty is the only certainty there is, and knowing how to live with insecurity is the only security.

John Allen Paulos

En la actualidad, la información es uno de los elementos más valiosos con los que contamos. Al desarrollar una aplicación, se deben adoptar los pasos necesarios para proteger la información almacenada.

Spring Security [18] es un marco de trabajo que proporciona seguridad declarativa para las aplicaciones basadas en Spring. Constituye una solución de seguridad integral que gestiona la autenticación y la autorización, tanto a nivel de las solicitudes Web como a nivel de invocación de métodos.

Para proteger solicitudes web y restringir el acceso al nivel de URL, utiliza filtros de *servlet*. Para proteger las invocaciones de métodos utiliza AOP de Spring, aplicando *proxy* sobre objetos y aplicando consejos que garanticen que el usuario cuenta con la autoridad adecuada para invocar métodos protegidos.

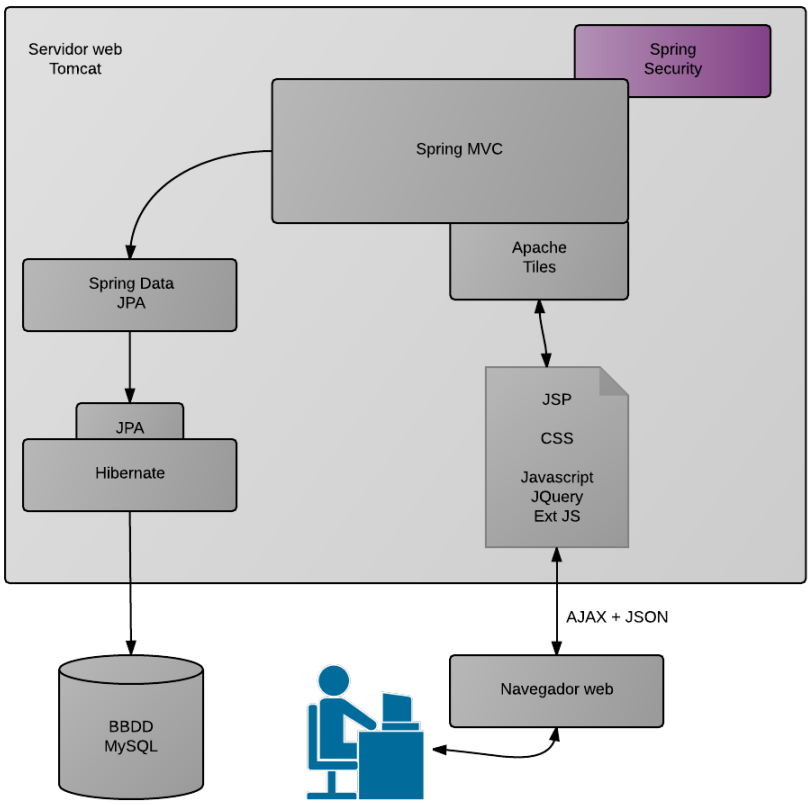


Figura 6-1. Arquitectura del sistema – Spring Security

6.1 Módulos de Spring Security

Spring Security se divide en once módulos, recogidos en la siguiente tabla.

Tabla 6–1 Módulos de Spring Security

Módulo	Descripción
ACL	Permite el uso de seguridad en objetos de dominio mediante Listas de Control de Acceso (ACL).
Aspectos	Un pequeño módulo para admitir aspectos basados en AspectJ en lugar de AOP estándar de Spring a la hora de usar anotaciones de Spring Security.
Cliente CAS	Permite la integración con el Servicio de Autenticación Central (CAS) de Jasig.
Configuración	Permite configurar Spring Security con XML y Java.
Núcleo	Proporciona la biblioteca esencial de Spring Security.
Criptografía	Permite usar encriptación y codificación con contraseñas.
LDAP	Permite la autenticación basada en LDAP.
OpenID	Permite la autenticación centralizada con OpenID.
Dispositivos remotos	Permite la integración con Spring Remoting.
Biblioteca de etiquetas	La biblioteca de etiquetas JSP de Spring Security.
Web	Proporciona el soporte para la seguridad Web basada en filtros de Spring Security.

En el proyecto se han incluido los módulos de núcleo, de configuración (los mínimos necesarios) y el módulo web y biblioteca de etiquetas.

Para ello en el fichero POM se incluye:

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

6.2 Protección de solicitudes web

Para proteger solicitudes web y restringir el acceso al nivel de URL, Spring Security utiliza filtros de *servlet*.

Spring facilita la configuración de seguridad y sólo es necesario configurar un filtro, *DelegatingFilterProxy*. Se trata de un filtro de *servlet* especial que delega a una implementación de *javax.servlet.Filter* registrada como *bean* en el contexto de aplicación de Spring.

Para dicha configuración se ha creado una clase que amplía *AbstractSecurityWebApplicationInitializer*:

```
package com.sophistas.configuracion;

import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebInitializer extends AbstractSecurityWebApplicationInitializer{
```

```
}
```

AbstractSecurityWebApplicationInitializer implementa *WebApplicationInitializer*, por lo que será detectable por Spring y se usará para registrar *DelegatingFilterProxy* con el contenedor Web. *DelegatingFilterProxy* interceptará las solicitudes entrantes en la aplicación y las delegará al bean con el ID *springSecurityFilterChain*.

Respecto al propio bean *springSecurityFilterChain*, se trata de otro filtro especial denominado *FilterChainProxy*. Es un filtro único que conecta entre sí uno o más filtros adicionales, que proporcionan las características de seguridad de Spring. Sin embargo, estos filtros no hay que declararlos de manera explícita ya que se crearán de manera automática al habilitar la seguridad web.

6.2.1 Habilitar la seguridad web para Spring MVC

A continuación, se muestra la clase usada en el proyecto para habilitar la seguridad web:

```
package com.sophistas.configuracion;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.encoding.Md5PasswordEncoder;
import org.springframework.security.config.annotation.authentication.
       builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.
       configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.
       configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

       @Autowired
       DataSource dataSource;

       private static final String USERSBYUSERNAMEQUERY = "SELECT USUARIO, CLAVE, TRUE FROM
       USUARIO WHERE USUARIO = ?";
       private static final String AUTHORITIESBYUSERNAMEQUERY = "SELECT USUARIO, 'ROLE_USER'
       FROM USUARIO WHERE USUARIO = ?";

       /*
        * Configura servicios de detalles de usuarios
        */
       @Override
       protected void configure(AuthenticationManagerBuilder auth) throws Exception {
              auth.jdbcAuthentication()
                     .dataSource(dataSource)
                     .usersByUsernameQuery(USERSBYUSERNAMEQUERY)
                     .authoritiesByUsernameQuery(AUTHORITIESBYUSERNAMEQUERY)
                     .passwordEncoder(new Md5PasswordEncoder());
       }

       /*
        * Configura la protección de las solicitudes por
        * parte de los interceptores
        */
       @Override
       protected void configure(HttpSecurity http) throws Exception {
              http.authorizeRequests()
                     .antMatchers("/login**").permitAll()
                     .anyRequest().authenticated()
        .and()
            .formLogin()
```

```
            .loginPage("/login")
            .defaultSuccessUrl("/home", true)
            .failureUrl("/login?error")
            .usernameParameter("usuario")
            .passwordParameter("clave")
        .and()
            .logout()
            .logoutSuccessUrl("/login?logout")
        .and()
              .csrf().disable();
       }

       /*
        * Configura la secuencia de filtros de Spring Security
        */
       @Override
       public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/estilo/**");
        web.ignoring().antMatchers("/ext/**");
        web.ignoring().antMatchers("/fuentes/**");
        web.ignoring().antMatchers("/imagenes/**");
        web.ignoring().antMatchers("/jquery/**");
        web.ignoring().antMatchers("/js/**");
       }

}
```

Es necesario configurar Spring Security en un bean que implemente *WebSecurityConfigurer* o (por comodidad) amplíe *WebSecurityConfigurerAdapter*, como es el caso.

La anotación *@EnableWebMvcSecurity* (del paquete *org.springframework.security.config.annotation.web.configuration*) habilita la seguridad web. Además, configura un solucionador de argumentos de Spring MVC para que los métodos de controlador puedan recibir el principal (o nombre de usuario) del usuario autenticado.

Para la configuración se han sobreescrito los tres métodos de *WebSecurityConfigurerAdapter*. A continuación, se describen para qué sirve cada uno de estos métodos:

Tabla 6–2 Métodos para la configuración de la seguridad web para Spring MVC

Método	Descripción
configure(AuthenticationManagerBuilder)	Se reemplaza para configurar servicios de detalles de usuarios.
configure(HttpSecurity)	Se reemplaza para configurar la protección de las solicitudes por parte de los interceptores.
configure(WebSecurity)	Se reemplaza para configurar la secuencia de filtros de Spring Security.

6.2.1.1 Servicios de detalles de usuarios

Es necesario un repositorio de usuarios en el que se guarden nombres de usuarios, contraseñas y otros datos, y desde el que se recuperen al tomar decisiones de autenticación.

Spring Security es muy flexible en este aspecto ofreciendo varias opciones:

- Repositorio de usuarios en memoria: Un repositorio de este tipo es muy útil para tareas de depuración y pruebas de desarrollo, pero no es la opción más acertada para una aplicación de producción.
- Repositorio de usuarios en una base de datos relacional: Es muy habitual que los datos de usuarios se almacenen en una base de datos relacional, como se ha hecho en la presente aplicación. Se describe en detalle en el siguiente apartado.

- Repositorio de usuarios en un directorio LDAP: Este sistema es útil si ya se tiene un servicio de directorio y no se quiere mantener cuentas de usuario y contraseñas adicionales para el acceso web.
- Servicio de usuario personalizado: Si las anteriores opciones no cubren las necesidades de autenticación para una cierta aplicación web, es posible crear y configurar un servicio personalizado de detalles de usuario.

Repositorio de usuarios en una base de datos relacional

En la aplicación se ha usado esta opción para el repositorio de usuarios, almacenar los datos de estos en una base de datos relacional a la que se accede a través de JDBC. Para configurar Spring Security para la autenticación sobre un repositorio de usuarios respaldado por JDBC se usa el método *jdbcAuthentication()*.

A continuación, se muestra la configuración del repositorio de usuarios que se ha aplicado a la aplicación:

```
@Autowired
DataSource dataSource;

private static final String USERSBYUSERNAMEQUERY = "select usuario, clave, true
       from Usuarios where usuario = ?";
private static final String AUTHORITIESBYUSERNAMEQUERY = "select usuario, 'ROLE_USER'
       from Usuarios where usuario = ?";

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
       auth.jdbcAuthentication().
              dataSource(dataSource).
              usersByUsernameQuery(USERSBYUSERNAMEQUERY).
              authoritiesByUsernameQuery(AUTHORITIESBYUSERNAMEQUERY).
              passwordEncoder(new Md5PasswordEncoder());
}
```

Como se observa, se especifica:

- Un *DataSource*, mediante el método *dataSource()*, que pueda acceder a la base de datos relacional.
- Una primera consulta, mediante el método *usersByUsernameQuery()*, que recupera el nombre del usuario, su contraseña y si está o no habilitado, información que se usa para autenticar al usuario.
- Una segunda consulta, mediante el método *authoritiesByUsernameQuery()*, que busca las autoridades concedidas al usuario para su autorización.
- Un codificador de contraseñas, mediante el método *passwordEncoder()*. La contraseña se debe almacenar en la base de datos con el codificador aquí especificado. El método *passwordEncoder()* acepta cualquier implementación de la interfaz *PasswordEncoder* de Spring Security.

Una nota acerca de las consultas de usuario. Estas consultas no son obligatorias de especificar. En caso de no indicarlas, Spring Security asume ciertos detalles sobre el esquema de base de datos. Espera que existan determinadas tablas para guardar datos de usuarios. En concreto, el siguiente fragmento de código de Spring Security muestra las consultas SQL que se ejecutarán al buscar detalles de usuarios:

```
public static final String DEF_USERS_BY_USERNAME_QUERY = "select username,password,enabled
from users where username = ?";

public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY = "select username,authority
from authorities where username = ?";
```

6.2.1.2 Intercepción de solicitudes web

En una aplicación, no todas las solicitudes deben protegerse de la misma forma. Algunas necesitan autenticación y otras no. Algunas solo estarán disponibles para usuarios con determinadas autoridades y no disponibles para los que no tengan dichas autoridades.

Para definir la seguridad en cada solicitud se debe reemplazar el método *configure(HttpSecurity)*.

A continuación, se muestra la configuración de intercepción de solicitudes web que se ha aplicado a la

aplicación:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
	http.authorizeRequests().
		antMatchers("/login**").permitAll().
		anyRequest().authenticated().
	and().
		formLogin().
		loginPage("/login").
		defaultSuccessUrl("/home", true).
		failureUrl("/login?error").
		usernameParameter("usuario").
		passwordParameter("clave").
	and().
		logout().
		logoutSuccessUrl("/login?logout").
	and().
		csrf().disable();
}
```

El objeto *HttpSecurity* proporcionado a *configure()* se puede usar para configurar distintos aspectos de seguridad HTTP. En este caso se invoca *authorizeRequests()* y después se invocan métodos en el objeto que devuelve para indicar que queremos configurar detalles de seguridad en el nivel de las solicitudes.

La invocación `antMatchers("/login**").permitAll()` especifica que a la página de login se puede acceder sin condiciones.

La invocación `anyRequest().authenticated()` especifica que para el resto de solicitudes, el usuario debe estar autenticado.

En la siguiente tabla se describen todas las opciones disponibles:

Tabla 6–3 Métodos para la configuración de los accesos

Método	Funcionamiento
access(String)	Permite el acceso si la expresión SpEL proporcionada evalúa a true.
anonymous()	Permite el acceso a usuarios anónimos.
authenticated()	Permite el acceso a usuarios autenticados.
denyAll()	Impide el acceso sin condiciones.
fullyAuthenticated()	Permite el acceso si el usuario está totalmente autenticado (no se recuerda de antes).
hasAnyAuthority(String…)	Permite el acceso si el usuario tiene alguna de las autoridades indicadas.
hasAnyRole(String…)	Permite el acceso si el usuario tiene alguna de las funciones indicadas.
hasAuthority(String)	Permite el acceso si el usuario tiene la autoridad indicada.
hasIpAddress(String)	Permite el acceso si la solicitud proviene de la dirección IP indicada.
hasRole(String)	Permite el acceso si el usuario tiene la función indicada.
not()	Niega el efecto de cualquiera de los demás métodos de acceso.
permitAll()	Permite el acceso sin condiciones.
rememberMe()	Permite el acceso a todos los usuarios autenticados tras recordar un acceso anterior.

Y en la siguiente tabla se pueden ver todas las expresiones SpEL (*Spring Expression Language*) disponibles en Spring Security (que se pueden usar en el método *access()* y en otros sitios como se verá más adelante):

Tabla 6–4 Expresiones SpEL

Expresión de seguridad	Se evalúa como
authentication	El objeto de autenticación del usuario.
denyAll	Siempre como *false*.
hasAnyRole(lista de funciones)	*true* si al usuario se le han otorgado alguna de las funciones especificadas.
hasRole(función)	*true* si al usuario se le ha otorgado la función especificada.
hasIpAddress(dirección IP)	*true* si la solicitud proviene de la dirección IP indicada.
isAnonymous()	*true* si el usuario actual es anónimo.
isAuthenticated()	*true* si el usuario actual está autenticado.
isFullyAuthenticated()	*true* si el usuario actual está totalmente autenticado (no es un usuario recordado).
isRememberMe()	*true* si el usuario actual se autenticó de forma automática al ser recordado.
permitAll	Siempre como *true*.
principal	El objeto principal del usuario.

Tras especificar en el método *configure()* qué solicitudes requerían autorización, se personaliza el formulario de login en el que el usuario va a introducir sus credenciales:

```
                formLogin().
                loginPage("/login").
                defaultSuccessUrl("/home", true).
                failureUrl("/login?error").
                usernameParameter("usuario").
                passwordParameter("clave").
        and().
                logout().
                logoutSuccessUrl("/login?logout").
```

Con estas llamadas se ha indicado:

- *loginPage()*: Página de login.
- *defaultSuccessUrl()*: Página a la que vamos si el usuario y clave son correctos.
- *failureUrl()*: Página a la que vamos si el usuario y/o clave son incorrectos.
- *usernameParameter()*: Parámetro del formulario que hace referencia al usuario.
- *passwordParameter()*: Parámetro del formulario que hace referencia a la clave.
- *logoutSuccessUrl()*: Página a la que vamos si el usuario cierra sesión.

Seguridad del canal

Otra funcionalidad que trae Spring Security, aunque no se ha usado en la aplicación, es la de reforzar la seguridad del canal.

Por defecto, los datos se envían sin codificar por HTTP, de modo que cualquier atacante puede interceptar la solicitud y ver su información. Por ese motivo, la información delicada (como números de tarjeta de crédito) debe enviarse cifrada a través de HTTPS.

Además del método *authorizeRequests()*, el objeto *HttpSecurity* pasado a *configure()* dispone de un método *requiresChannel()* que permite declarar requisitos de canal para distintos patrones de URL.

Como la aplicación "Gestión de Sophistas Academia" se va a usar dentro de una *Intranet* no ha sido necesario reforzar la seguridad del canal.

Falsificación de petición en sitios cruzados

El ataque de falsificación de petición en sitios cruzados (CSRF) se produce cuando un sitio engaña a un usuario para que envíe una solicitud a otro servidor, posiblemente con un resultado negativo.

En Spring Security, la protección contra CSRF se habilita de forma predeterminada. Implementa dicha protección mediante un *token* de sincronización. Las solicitudes de cambio de estado (las que no son GET, HEAD, OPTIONS o TRACE) se interceptan y se comprueba si tienen un *token* CSRF. Si no lo incluyen o si no coincide con el del servidor, la solicitud falla con *CsrfException*.

Esto significa que los formularios de la aplicación Web deben enviar un *token* en un campo *_csrf* y que dicho token debe ser el mismo que se haya calculado y almacenado en el servidor para que coincida al enviar el formulario.

De nuevo, como la aplicación "Gestión de Sophistas Academia" se va a usar dentro de una Intranet, se ha optado por desactivar la protección contra CSRF con la invocación `csrf().disable()` dentro del método *configure()*.

6.2.2 Proteger la vista

Spring Security admite seguridad en la capa de la vista con una biblioteca de etiquetas JSP.

Tabla 6–5 Etiquetas JSP de Spring Security

Etiquetas JSP	Función
`<security:accesscontrollist>`	Representa condicionalmente su contenido si el usuario recibe autoridades por parte de una lista de control de acceso.
`<security:authentication>`	Representa detalles sobre la autenticación actual.
`<security:authorize>`	Representa condicionalmente su contenido si el usuario recibe determinadas autoridades o si una expresión SpEL evalúa a *true*.

Para poder utilizar la biblioteca de etiquetas JSP, tenemos que declararla en los archivos JSP donde la utilicemos:

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security" %>
```

6.2.2.1 Acceder a la información de autenticación

Una de las tareas que la biblioteca de etiquetas JSP de Spring Security hace es proporcionar acceso a la información de autenticación del usuario, por medio de la etiqueta JSP `<security:authentication>`.

Para representar el valor de una propiedad del objeto de autenticación del usuario:

```
<security:authentication property="nombre_propiedad" />
```

Para asignar este valor a una variable:

```
<security:authentication property="nombre_propiedad" var="nombre_variable" />
```

Las propiedades disponibles varían en función de la forma en que se autentique al usuario. A continuación, se recogen las propiedades comunes:

Tabla 6–6 Propiedades del objeto de autenticación del usuario

Propiedad de autenticación	Descripción
authorities	Una colección de objetos *GrantedAuthority* que representa los privilegios otorgados al usuario.
credentials	Las credenciales utilizadas para verificar el elemento principal (por lo general, la contraseña del usuario).
details	Información adicional sobre la autenticación (dirección IP, número de serie del certificado, ID de sesión, etc.).
principal	El principal del usuario.

En la aplicación "Gestión de Sophistas Academia", en el encabezado de la página, una vez un usuario ha iniciado sesión, se muestra un mensaje en el que se identifica al usuario por su nombre de usuario:

```
Hola, <security:authentication property="principal.username"/>
```

6.2.2.2 Representación condicional

Hay ocasiones en las que determinados fragmentos de la vista deben representarse, o no, en función de los privilegios de un usuario.

La etiqueta JSP de Spring Security `<security:authorize>` representa de forma condicional un fragmento de la vista en función de las autoridades con las que cuente el usuario.

Por ejemplo, el siguiente fragmento sólo sería visible a usuarios administradores (usuarios con el rol ROLE_ADMIN):

```
<security:authorize access="hasRole('ROLE_ADMIN')">
       Fragmento_accesible_administradores
</security>
```

Al atributo *access* se le proporciona una expresión SpEL cuyo resultado determina si el cuerpo de `<security:authorize>` se va a representar.

6.3 Protección de métodos

La seguridad de los métodos es un importante complemento a la seguridad del nivel Web de Spring Security, que se ha visto hasta ahora en el capítulo, respaldando las reglas de seguridad declaradas para proteger las solicitudes web.

El enfoque más utilizado de seguridad de métodos con Spring Security consiste en aplicar anotaciones de seguridad especiales a los métodos que se desea proteger. Spring Security ofrece tres tipos diferentes de anotaciones de seguridad:

- *@Secured* de Spring Security.
- *@RolesAllowed* de JSR-250.
- Anotaciones controladas por expresiones: *@PreAuthorize*, *@PostAuthorize*, *@PreFilter* y *@PostFilter*.

6.3.1 Protección de métodos con *@Secured*

Para habilitar la seguridad de métodos con esta anotación es necesario una clase de configuración de esta forma:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled=true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
```

```
}
```

Se observa que el atributo *secureEnabled* de la anotación *@EnableGlobalMethod* se establece en *true*, para crear un punto de corte de forma que los aspectos de Spring Security envuelvan los métodos de bean anotados con *@Secured*.

A continuación, se muestra un método anotado con *@Secured*:

```
@Secured({"ROLE_X","ROLE_Y"})
public void metodo() {
        // …

}
```

La anotación *@Secured* acepta una matriz de elementos *String* como argumento. Cada valor *String* es una autorización, necesaria para invocar el método. El usuario autenticado debe contar con, al menos, una de esas autoridades para obtener acceso al método.

Cuando un usuario no autenticado o uno que no cuente con los privilegios necesarios intenta invocar el método, el aspecto que contiene al método genera una de las excepciones de seguridad de Spring Security.

6.3.2 Protección de métodos con *@RolesAllowed*

La anotación *@RolesAllowed* es prácticamente equivalente a *@Secured* en todos los aspectos. La única diferencia importante es que *@RolesAllowed* es una de las anotaciones estándar de Java, definida en la JSR-250. Esta diferencia tiene un efecto más político que técnico.

Para habilitar la seguridad de métodos con esta anotación es necesario una clase de configuración de esta forma:

```
@Configuration
@EnableGlobalMethodSecurity(jsr250Enabled=true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {

}
```

Al establecer *jsr250Enabled* en *true*, se crea un punto de corte para que todos los métodos anotados con *@RolesAllowed* se envuelvan con aspectos de Spring Security.

A continuación, se muestra un método anotado con *@RolesAllowed*:

```
@RolesAllowed({"ROLE_X","ROLE_Y"})
public void metodo() {
        // …

}
```

La anotación *@RolesAllowed* acepta una matriz de elementos *String* como argumento. Cada valor *String* es una autorización, necesaria para invocar el método. El usuario autenticado debe contar con, al menos, una de esas autoridades para obtener acceso al método.

De nuevo, cuando un usuario no autenticado o uno que no cuente con los privilegios necesarios intenta invocar el método, el aspecto que contiene al método genera una de las excepciones de seguridad de Spring Security.

En algunos casos, las restricciones de seguridad dependen de algo más que los privilegios concretos de un usuario. Es entonces donde entran en juego las anotaciones del siguiente apartado.

6.3.3 Protección de métodos con *@PreAuthorize, @PostAuthorize, @PreFilter* y *@PostFilter*

Estas anotaciones utilizan SpEL para permitir restricciones de seguridad más interesantes para métodos. A continuación, se describen:

Tabla 6–7 Anotaciones para protección de métodos

Anotaciones	Descripción
@PreAuthorize	Restringe el acceso a un método antes de su invocación en función del resultado de evaluar una expresión.
@PostAuthorize	Permite la invocación de un método, aunque genera una excepción de seguridad si la expresión evalúa a *false*.
@PostFilter	Permite la invocación de un método, aunque filtra sus resultados en función de una expresión.
@PreFilter	Permite la invocación de un método, pero filtra la información de entrada antes de acceder al método.

Todas estas anotaciones aceptan una expresión SpEL para su parámetro *value*. La expresión puede ser cualquier expresión SpEL válida. Si la expresión evalúa a *true*, la regla de seguridad pasa; en caso contrario falla. Las implicaciones del resultado de la expresión difieren en función de la anotación empleada.

Para habilitar la seguridad de métodos con estas anotaciones es necesario una clase de configuración de esta forma:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled=true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {

}
```

7 ANÁLISIS DE REQUISITOS

The two most important requirements for major success are: first, being in the right place at the right time, and second, doing something about it.

Ray Kroc

El objetivo del análisis de requisitos es determinar las condiciones o capacidades que debe cumplir la aplicación que se quiere diseñar para satisfacer las necesidades de sus usuarios. Para lograr esto utilizamos la definición de requisitos. Un requisito se puede entender como una descripción informal de las necesidades y deseos que tiene el usuario final respecto a la aplicación.

Estos requisitos se han clasificado en los siguientes tipos y se van a definir para la aplicación "Gestión de Sophistas Academia" en las siguientes secciones:

- Requisitos de Información
- Requisitos Funcionales
- Requisitos de Negocio
- Requisitos de Seguridad
- Requisitos de Interfaz

7.1 Introducción

Actualmente en Sophistas Academia, las responsables llevan la gestión de ésta a base de "papeleo": las altas de los alumnos, de los profesores, sus horarios, las tarifas, etc. Esto provoca que las situaciones de caos se repitan cada vez de manera más continuada a medida que la academia crece.

Por un lado, el acceso a la información es complicado, cuando tienen la necesidad de buscar y recuperar unos determinados datos, se dirigen al archivo para hacer esta tarea manualmente. A medida que el número de alumnos empieza a incrementarse, la desorganización y dificultad en consultar información en los documentos empieza a ser visible.

Por otro lado, este aumento del número de alumnos y por consiguiente de clases, hace que cada vez sea más difícil cuadrar los horarios de dichos alumnos y los profesores sin ninguna herramienta disponible para ello.

En consecuencia, un software de gestión se hace cada vez más necesario, ya que permite:

- Dar acceso a los datos de forma controlada a los destinatarios que deban conocerlos
- Reducir trabajo burocrático
- Analizar los datos (es decir, convertir los datos en información relevante que permita saber qué sucede, por qué sucede y facilite la toma de decisiones adecuadas para mejorar la gestión)
- Asegurar coherencia de la información, evitando errores en los datos, ya que éstos son validados antes de ser almacenados en el sistema
- Facilitar la salvaguarda de la información; al estar todos los datos en un único lugar, resulta muy sencillo implantar un sistema de respaldo que realice copias de seguridad de la información de la academia

En resumen, todo esto supone a la larga un importante ahorro de tiempo y también económico para la academia y una mejora en la calidad de sus servicios.

La presente aplicación tiene como objetivo servir de software de gestión a Sophistas Academia. En el presente capítulo se detallan los diferentes requisitos de la misma.

7.2 Requisitos de Información

Esta sección describe de forma detallada todos los requisitos de almacenamiento de información. Estos son aquellos datos que la aplicación deberá almacenar de manera que alguna persona en un momento determinado pueda recuperarlos.

7.2.1 Requisitos de Información de Profesores

La aplicación deberá almacenar la información referida a los profesores. Un profesor se define por:

Tabla 7–1 Requisitos de almacenamiento de Profesores

Atributo	Tipo de dato	Validaciones y observaciones
Nombre	Cadena de hasta 50 caracteres	Obligatorio
Apellido 1	Cadena de hasta 50 caracteres	Obligatorio
Apellido 2	Cadena de hasta 50 caracteres	Opcional
NIF	Cadena de 9 caracteres	Obligatorio / Formato NIF / Debe ser único
Teléfono	Cadena de 9 caracteres numéricos	Obligatorio
Correo	Cadena de hasta 100 caracteres	Obligatorio / Formato de correo electrónico
Titulación	Cadena de hasta 100 caracteres	Opcional

7.2.2 Requisitos de Información de Alumnos

La aplicación deberá almacenar la información referida a los alumnos. Un alumno se define por:

Tabla 7–2 Requisitos de almacenamiento de Alumnos

Atributo	Tipo de dato	Validaciones y observaciones
Nombre	Cadena de hasta 50 caracteres	Obligatorio
Apellido 1	Cadena de hasta 50 caracteres	Obligatorio
Apellido 2	Cadena de hasta 50 caracteres	Opcional
NIF	Cadena de 9 caracteres	Opcional / Formato NIF / Debe ser único
Teléfono	Cadena de 9 caracteres numéricos	Ver apartado 2.2.2.
Correo	Cadena de hasta 100 caracteres	Ver apartado 2.2.2. / Formato de correo electrónico
Curso	Referencia a un curso	Obligatorio
Repetidor	Sí / No	Obligatorio
Fecha alta	Fecha	Obligatoria / Tomará el valor de la fecha en que el alumno se da de alta (no habrá que introducirla manualmente)

Fecha baja	Fecha	Inicialmente está vacía, tomará el valor de la fecha en que el alumno se da de baja (no habrá que introducirla manualmente)
Observaciones	Cadena de hasta 2000 caracteres	Opcional
Clases	Lista de referencias a clases	Opcional

7.2.2.1 Requisitos de Información de Responsables de Alumnos

En algunas ocasiones se deberá almacenar aparte de la información referida al alumno, información referida a un responsable de éste (padre, tutor, …). Un responsable se define por:

Tabla 7–3 Requisitos de almacenamiento de Responsables de Alumnos

Atributo	Tipo de dato	Validaciones y observaciones
Nombre	Cadena de hasta 50 caracteres	Obligatorio
Apellido 1	Cadena de hasta 50 caracteres	Obligatorio
Apellido 2	Cadena de hasta 50 caracteres	Opcional
NIF	Cadena de 9 caracteres	Obligatorio / Formato NIF
Teléfono	Cadena de 9 caracteres numéricos	Obligatorio
Correo	Cadena de hasta 100 caracteres	Obligatorio / Formato de correo electrónico

Estos atributos se validarán en el caso en que el usuario rellene alguno de ellos en el pertinente formulario de alta de alumno (o edición), puesto que con ello se considera que se debe almacenar un responsable para el alumno.

7.2.2.2 Observaciones

En el caso en que un alumno no tenga responsable, los atributos Teléfono y Correo del alumno serán obligatorios. Esto es, por cada alumno, se tiene que tener al menos un teléfono y un correo, ya sea de éste o de su responsable.

7.2.3 Requisitos de Información de Clases

La aplicación deberá almacenar la información referida a las clases. Una clase se define por:

Tabla 7–4 Requisitos de almacenamiento de Clases

Atributo	Tipo de dato	Validaciones y observaciones
Asignatura	Referencia a una asignatura	Obligatorio
Profesor	Referencia a un profesor	Obligatorio
Tarifa	Referencia a una tarifa	Obligatorio
Horario	Lista de referencias a horas semanales	Opcional

7.2.4 Requisitos de Información de Cursos

La aplicación deberá almacenar la información referida a los cursos. Un curso se define por:

Tabla 7–5 Requisitos de almacenamiento de Cursos

Atributo	Tipo de dato	Validaciones y observaciones
Nivel	Entero	Obligatorio
Etapa	Cadena de hasta 50 caracteres	Obligatorio / Valores: Primaria, Secundaria, …

No puede haber dos cursos con el mismo nivel y etapa.

7.2.5 Requisitos de Información de Asignaturas

La aplicación deberá almacenar la información referida a las asignaturas. Una asignatura se define por:

Tabla 7–6 Requisitos de almacenamiento de Asignaturas

Atributo	Tipo de dato	Validaciones y observaciones
Nombre	Cadena de hasta 50 caracteres	Obligatorio
Curso	Referencia a un curso	Obligatorio

No puede haber dos asignaturas del mismo curso con el mismo nombre.

7.2.6 Requisitos de Información de Tarifas

La aplicación deberá almacenar la información referida a las tarifas. Una tarifa se define por:

Tabla 7–7 Requisitos de almacenamiento de Tarifas

Atributo	Tipo de dato	Validaciones y observaciones
Nombre	Cadena de hasta 50 caracteres	Obligatorio / Debe ser único
Descripción	Cadena de hasta 256 caracteres	Obligatorio
Precio	Decimal	Obligatorio

7.3 Requisitos Funcionales

Esta sección describe de forma detallada todos aquellos mecanismos que la aplicación debe cumplir para que ayude al usuario final a lograr un objetivo.

7.3.1 Requisitos Funcionales sobre Profesores

7.3.1.1 Búsqueda de Profesores

La aplicación deberá soportar la funcionalidad de poder buscar una serie de profesores a través de un buscador. Dicho buscador se basará en los siguientes campos o propiedades:

- Nombre: El nombre del profesor debe contener la cadena introducida en este campo del buscador
- NIF: El NIF del profesor debe coincidir exactamente con la cadena introducida en este campo del buscador

En la búsqueda no se hace distinción de mayúsculas y minúsculas.

El buscador deberá mostrar un listado con todos aquellos registros que cumplieron las especificaciones de búsqueda. En dicho listado se mostrará, en primer lugar, una cabecera con los nombres de los campos principales:

- Nombre

- NIF
- Correo
- Teléfono

Si no se rellena ningún campo del buscador se mostrarán todos los profesores.

7.3.1.2 Alta de Profesores

La aplicación deberá ser capaz de soportar la funcionalidad de dar de alta un profesor. Para ello se permitirá insertar datos dentro de una serie de campos por pantalla:

- Nombre
- Apellido 1
- Apellido 2
- NIF
- Teléfono
- Correo
- Titulación

La información tendrá que pasar una serie de validaciones obligatorias antes de ser guardada. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.1.3 Detalle y edición de los datos personales de los Profesores

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver los datos personales de un profesor. Para ello se mostrará una pantalla con el detalle de los campos de un profesor, con el mismo aspecto que en la pantalla de alta de profesores.

Todos los campos serán editables, dando la opción de modificar la información almacenada del profesor. Antes de ser guardada, la nueva información tendrá que pasar una serie de validaciones. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.1.4 Consulta del horario de los Profesores

La aplicación deberá ser capaz de soportar la funcionalidad de poder consultar el horario de un profesor.

7.3.1.5 Eliminación de Profesores

La aplicación deberá ser capaz de soportar la funcionalidad de eliminar un profesor. Antes de ser eliminado la aplicación deberá pedir confirmación.

7.3.2 Requisitos Funcionales sobre Alumnos

7.3.2.1 Búsqueda de Alumnos

La aplicación deberá soportar la funcionalidad de poder buscar una serie de alumnos a través de un buscador. Dicho buscador se basará en los siguientes campos o propiedades:

- Nombre: El nombre del alumno debe contener la cadena introducida en este campo del buscador
- Curso: El alumno debe ser del curso indicado en este campo del buscador
- Mostrar sólo actuales: Si se indica que se muestren sólo los actuales, no aparecerán en el resultado de la búsqueda aquellos alumnos que se han dado de baja

En la búsqueda no se hace distinción de mayúsculas y minúsculas.

El buscador deberá mostrar un listado con todos aquellos registros que cumplieron las especificaciones de búsqueda. En dicho listado se mostrará, en primer lugar, una cabecera con los nombres de los campos principales:

- Nombre
- Curso
- Responsable
- Fecha Alta
- Fecha Baja

Si no se rellena ningún campo del buscador se mostrarán todos los alumnos.

7.3.2.2 Alta de nuevos Alumnos (y Responsable en su caso)

La aplicación deberá ser capaz de soportar la funcionalidad de dar de alta un alumno (y su responsable en caso de que sea necesario). Para ello se permitirá insertar datos dentro de una serie de campos por pantalla:

- Nombre
- Apellido 1
- Apellido 2
- NIF
- Teléfono
- Correo
- Curso
- Condición de repetidor
- Fecha alta: Vendrá rellena con el día actual
- Fecha baja: Vendrá vacía y sin posibilidad de rellenar
- Observaciones
- Responsable:
    - Nombre
    - Apellido 1
    - Apellido 2
    - NIF
    - Teléfono
    - Correo

La información tendrá que pasar una serie de validaciones obligatorias antes de ser guardada. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.2.3 Detalle y edición de los datos personales de los Alumnos (y Responsable en su caso)

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver los datos personales de un alumno, y de su responsable en el caso de que lo tuviera. Para ello se mostrará una pantalla con el detalle de los campos de un alumno, con el mismo aspecto que en la pantalla de alta de alumnos.

Todos los campos serán editables (excepto los campos para las fechas de alta y baja), dando la opción de modificar la información almacenada del alumno. Antes de ser guardada, la nueva información tendrá que pasar una serie de validaciones. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.2.4 Asignación de clases a Alumnos

La aplicación deberá ser capaz de soportar la funcionalidad de asignar clases a un alumno.

7.3.2.5 Consulta del horario de los Alumnos

La aplicación deberá ser capaz de soportar la funcionalidad de poder consultar el horario de un alumno.

7.3.2.6 Baja de Alumnos

La aplicación deberá ser capaz de soportar la funcionalidad de poder dar de baja un alumno. Antes de ser dado de baja la aplicación deberá pedir confirmación.

7.3.2.7 Alta de antiguos Alumnos dados de baja

La aplicación deberá ser capaz de soportar la funcionalidad de poder dar de alta un alumno que había sido dado de baja previamente. Antes de ser dado de alta la aplicación deberá pedir confirmación.

7.3.3 Requisitos Funcionales sobre Clases

7.3.3.1 Búsqueda de Clases

La aplicación deberá soportar la funcionalidad de poder buscar una serie de clases a través de un buscador. Dicho buscador se basará en los siguientes campos o propiedades:

- Curso: La clase debe ser del curso indicado en este campo del buscador
- Asignatura: La clase debe ser de la asignatura indicada en este campo del buscador
- Profesor: La clase debe ser impartida por el profesor indicado en este campo del buscador

El buscador deberá mostrar un listado con todos aquellos registros que cumplieron las especificaciones de búsqueda. En dicho listado se mostrará, en primer lugar, una cabecera con los nombres de los campos principales:

- Curso
- Asignatura
- Profesor
- Horario
- Tarifa

Si no se rellena ningún campo del buscador se mostrarán todas las clases.

7.3.3.2 Alta de Clases

La aplicación deberá ser capaz de soportar la funcionalidad de dar de alta una clase. Para ello se permitirá insertar datos dentro de una serie de campos por pantalla:

- Curso
- Asignatura
- Profesor
- Tarifa
- Horario

La información tendrá que pasar una serie de validaciones obligatorias antes de ser guardada. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.3.3 Detalle y edición de las Clases

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver el detalle de una clase. Para ello se mostrará una pantalla con el detalle de los campos de una clase, con el mismo aspecto que en la pantalla de alta de clases.

Todos los campos, excepto el curso y la asignatura, serán editables, dando la opción de modificar la información almacenada de la clase. Antes de ser guardada, la nueva información tendrá que pasar una serie de validaciones. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.3.4 Ver lista de alumnos

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver el listado de alumnos que están apuntados a una determinada clase.

7.3.3.5 Eliminación de Clases

La aplicación deberá ser capaz de soportar la funcionalidad de eliminar una clase. Antes de ser eliminada la aplicación deberá pedir confirmación.

7.3.4 Requisitos Funcionales sobre Cursos

7.3.4.1 Listado de Cursos

La aplicación deberá soportar la funcionalidad de poder listar todos los cursos. En dicho listado se mostrará, en primer lugar, una cabecera con los nombres de los campos:

- Nivel
- Etapa

7.3.4.2 Alta de Cursos

La aplicación deberá ser capaz de soportar la funcionalidad de dar de alta un curso. Para ello se permitirá insertar datos dentro de una serie de campos por pantalla:

- Nivel
- Etapa

La información tendrá que pasar una serie de validaciones obligatorias antes de ser guardada. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.4.3 Detalle y edición de los Cursos

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver el detalle de un curso. Para ello se mostrará una pantalla con el detalle de los campos de un curso, con el mismo aspecto que en la pantalla de alta de cursos.

Todos los campos serán editables, dando la opción de modificar la información almacenada del curso. Antes de ser guardada, la nueva información tendrá que pasar una serie de validaciones. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.4.4 Eliminación de Cursos

La aplicación deberá ser capaz de soportar la funcionalidad de eliminar un curso. Antes de ser eliminado la aplicación deberá pedir confirmación.

7.3.5 Requisitos Funcionales sobre Asignaturas

7.3.5.1 Listado de Asignaturas

La aplicación deberá soportar la funcionalidad de poder listar todas las asignaturas. En dicho listado se mostrará, en primer lugar, una cabecera con los nombres de los campos:

- Nombre
- Curso

7.3.5.2 Alta de Asignaturas

La aplicación deberá ser capaz de soportar la funcionalidad de dar de alta una asignatura. Para ello se permitirá

insertar datos dentro de una serie de campos por pantalla:

- Nombre
- Curso

La información tendrá que pasar una serie de validaciones obligatorias antes de ser guardada. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.5.3 Detalle y edición de las Asignaturas

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver el detalle de una asignatura. Para ello se mostrará una pantalla con el detalle de los campos de una asignatura, con el mismo aspecto que en la pantalla de alta de asignaturas.

Todos los campos serán editables, dando la opción de modificar la información almacenada de la asignatura. Antes de ser guardada, la nueva información tendrá que pasar una serie de validaciones. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.5.4 Eliminación de Asignaturas

La aplicación deberá ser capaz de soportar la funcionalidad de eliminar una asignatura. Antes de ser eliminada la aplicación deberá pedir confirmación.

7.3.6 Requisitos Funcionales sobre Tarifas

7.3.6.1 Listado de Tarifas

La aplicación deberá soportar la funcionalidad de poder listar todas las tarifas. En dicho listado se mostrará, en primer lugar, una cabecera con los nombres de los campos:

- Nombre
- Descripción
- Precio(€/hora)

7.3.6.2 Alta de Tarifas

La aplicación deberá ser capaz de soportar la funcionalidad de dar de alta una tarifa. Para ello se permitirá insertar datos dentro de una serie de campos por pantalla:

- Nombre
- Descripción
- Precio(€/hora)

La información tendrá que pasar una serie de validaciones obligatorias antes de ser guardada. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.6.3 Detalle y edición de las Tarifas

La aplicación deberá ser capaz de soportar la funcionalidad de poder ver el detalle de una tarifa. Para ello se mostrará una pantalla con el detalle de los campos de una tarifa, con el mismo aspecto que en la pantalla de alta de tarifas.

Todos los campos serán editables, dando la opción de modificar la información almacenada de la tarifa. Antes de ser guardada, la nueva información tendrá que pasar una serie de validaciones. Estas validaciones se indican en las secciones Requisitos de Información y Requisitos de Negocio del presente capítulo.

7.3.6.4 Eliminación de Tarifas

La aplicación deberá ser capaz de soportar la funcionalidad de eliminar una tarifa. Antes de ser eliminada la

aplicación deberá pedir confirmación.

7.3.7 Requisitos Funcionales sobre Estadísticas

7.3.7.1 Estadísticas sobre Etapas

La aplicación deberá ser capaz de soportar la funcionalidad de mostrar el porcentaje de alumnos matriculados en la academia por etapas.

7.3.7.2 Estadísticas sobre Altas y Bajas

La aplicación deberá ser capaz de soportar la funcionalidad de mostrar el número de altas y bajas de alumnos que ha habido en la academia en los últimos cuatro meses.

7.4 Requisitos de Negocio

Esta sección describe de forma detallada los requisitos que comprenden distintas reglas para el negocio.

7.4.1 Requisito de Negocio Profesores y compatibilidad horaria

Cuando se asigna unas horas a un profesor para una clase, ya sea al dar de alta la clase o al modificarla y cambiar el profesor o el horario, se debe comprobar que éste esté libre a estas horas, es decir, que no tenga ya asignada otra clase.

7.4.2 Requisito de Negocio Alumnos y compatibilidad horaria

Cuando se modifica el horario de una clase se debe comprobar que los alumnos de dicha clase estén libres en el nuevo horario, es decir, que no estén acudiendo a otras clases.

Cuando se asigna una clase a un alumno se debe comprobar que el alumno esté libre en las horas en las que se imparte dicha clase.

7.4.3 Requisito de Negocio Alumno y clases de su curso

Las clases que se van a poder asignar a un alumno son aquellas que pertenezcan a su curso.

7.4.4 Requisito de Negocio Alumno y cambio de curso

Cuando se modifica el curso de un alumno se le elimina de las clases donde éste estaba apuntado (para cumplir con el requisito 7.4.3.).

7.4.5 Requisito de Negocio Alumno y baja

Cuando se da de baja un alumno se le elimina de las clases donde éste estaba apuntado.

7.5 Requisitos de Seguridad

Esta sección describe de forma detallada los requisitos sobre seguridad que debe cumplir la aplicación.

7.5.1 Requisito de Seguridad Acceso a la aplicación

Para poder acceder a la aplicación el usuario deberá en primer lugar loguearse, introduciendo un nombre de usuario y contraseña correctos.

7.5.2 Requisito de Seguridad Expiración de sesión

La sesión de usuario debe caducar a los 30 minutos de inactividad.

7.6 Requisitos de Interfaz

Esta sección describe de forma detallada todos los requisitos "visuales" que debe aportar la aplicación.

7.6.1 Ventana de Identificación

La aplicación deberá ser capaz de mostrar la siguiente ventana al inicio de la misma:

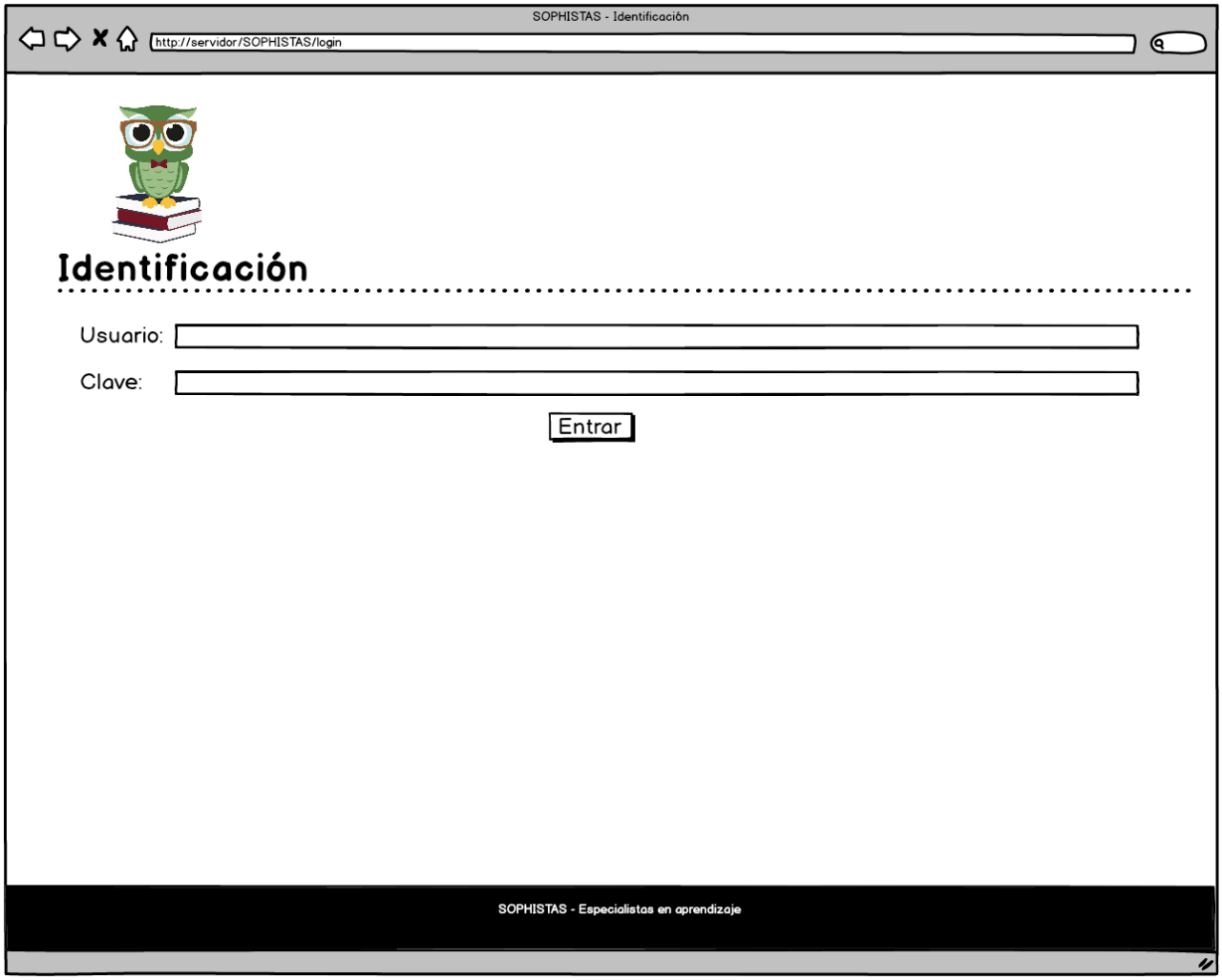


Figura 7-1. Ventana de Identificación

7.6.2 Ventana de Inicio

La aplicación deberá ser capaz de mostrar la siguiente ventana cuando el usuario se loguee:

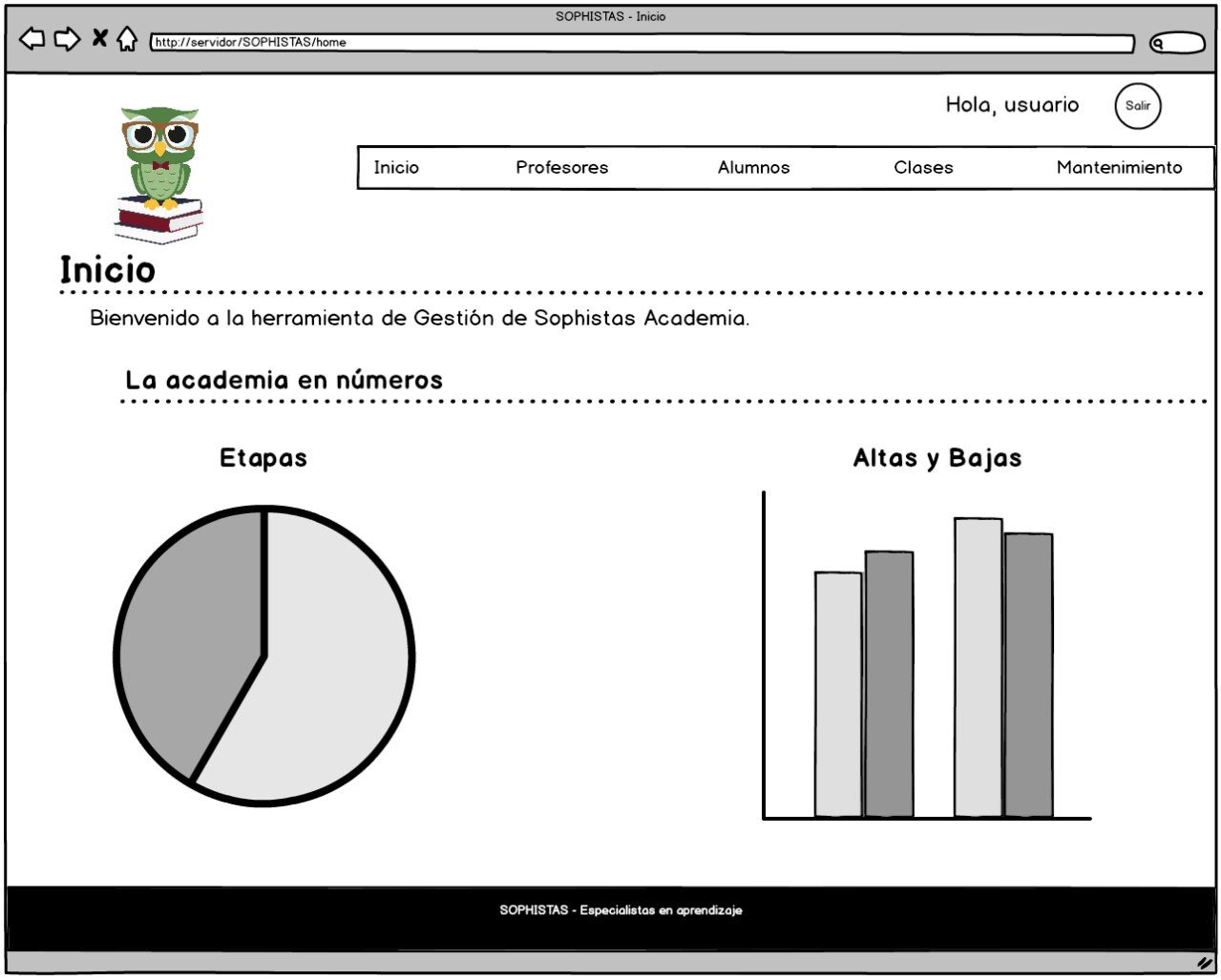


Figura 7-2. Ventana de Inicio

Como se aprecia, una vez logueado el usuario, le deberá aparecer un mensaje de saludo en la esquina superior derecha con su nombre.

7.6.3 Ventanas de Profesorado

La aplicación deberá ser capaz de mostrar la siguiente ventana con el buscador de profesores cuando el usuario entre en la sección "Profesorado":

Figura 7-3. Ventana de Búsqueda de profesores

Cuando el usuario haga clic sobre el botón "Nuevo profesor", le deberá aparecer la siguiente ventana emergente con el formulario de alta de profesor:

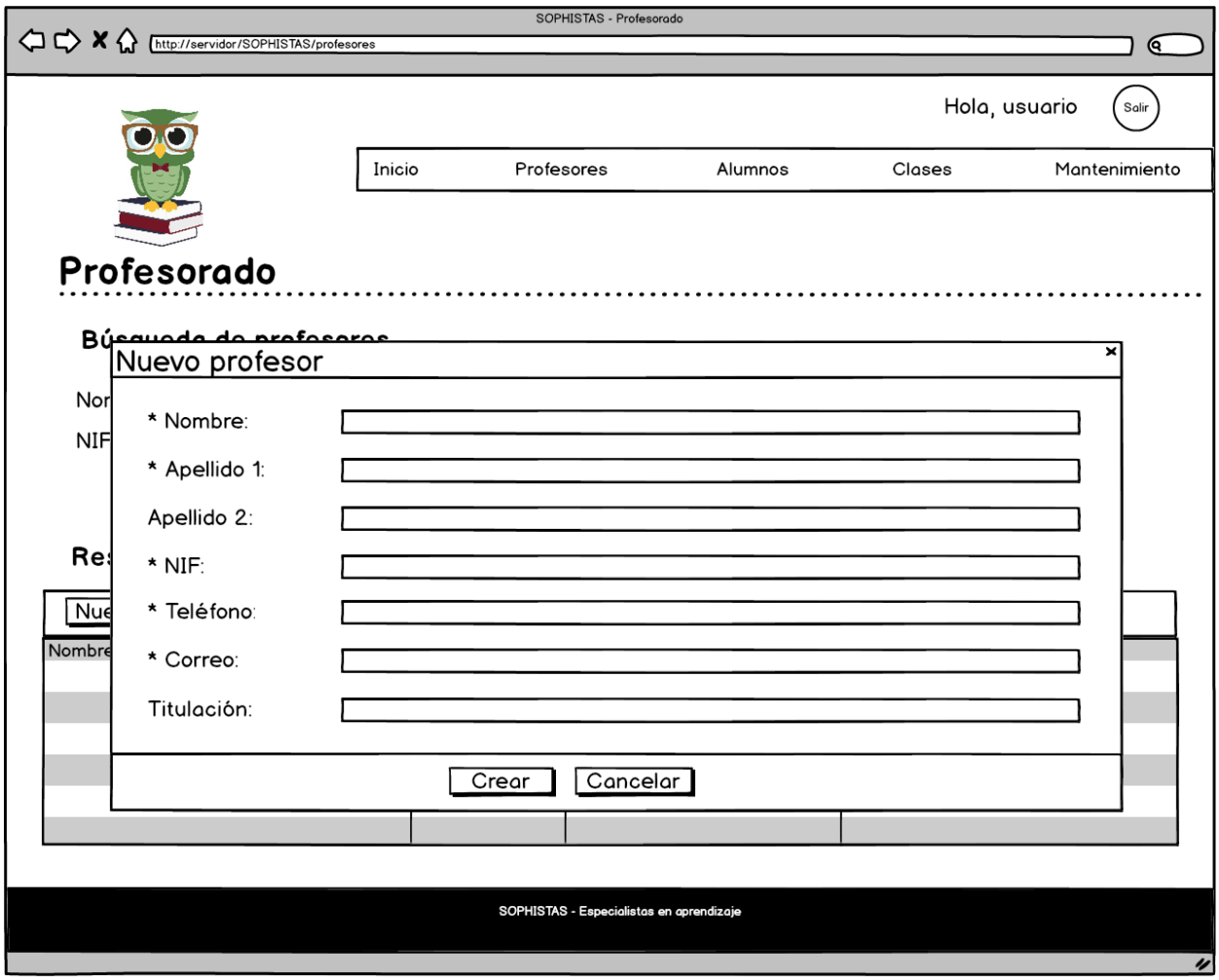


Figura 7-4. Ventana de Nuevo profesor

Cuando el usuario seleccione un profesor de la lista de profesores (resultado de la búsqueda) y haga clic en el botón "Datos personales", le deberá aparecer una ventana emergente como ésta, pero con los campos rellenos y el título "Datos personales profesor".

Cuando el usuario seleccione un profesor de la lista de profesores (resultado de la búsqueda) y haga clic en el botón “Clases”, le deberá aparecer la siguiente ventana emergente con las clases asignadas al profesor:

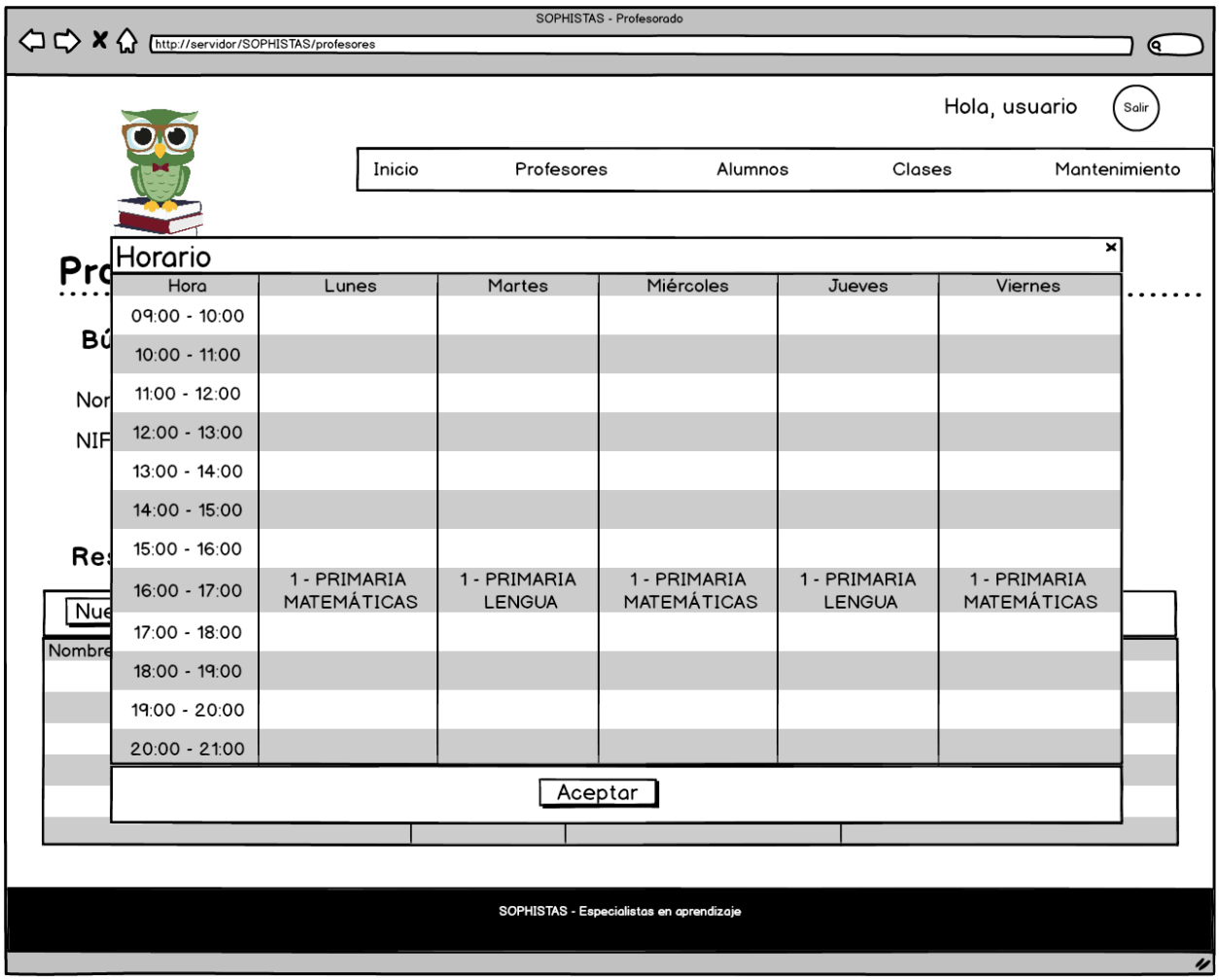


Figura 7-5. Ventana de Clases del profesor

Cuando el usuario seleccione un profesor de la lista de profesores (resultado de la búsqueda) y haga clic en el botón “Eliminar profesor”, le deberá aparecer una simple ventana emergente de confirmación con los botones “Sí” y “No”.

7.6.4 Ventanas de Alumnado

La aplicación deberá ser capaz de mostrar la siguiente ventana con el buscador de alumnos cuando el usuario entre en la sección "Alumnado":

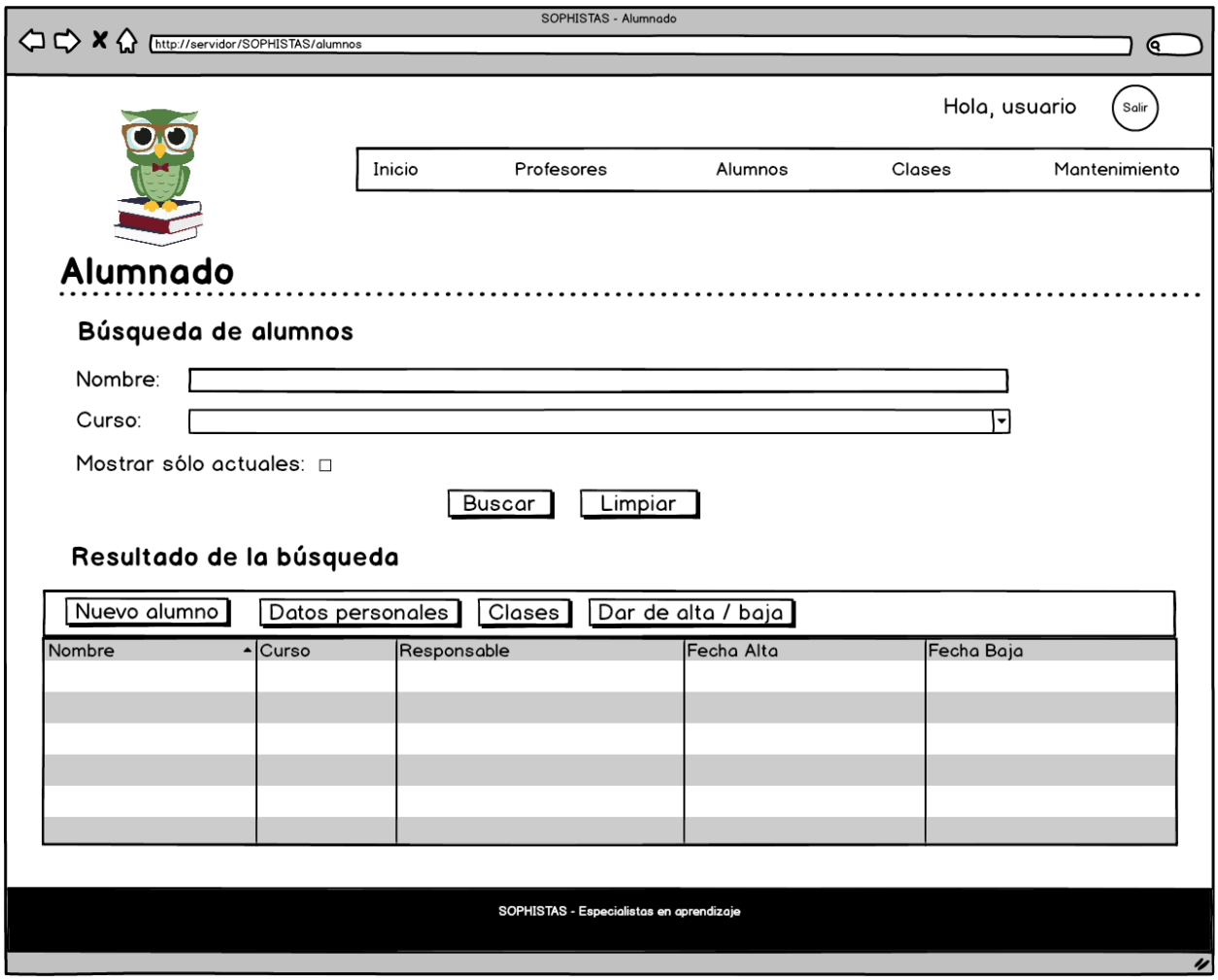


Figura 7-6. Ventana de Búsqueda de alumnos

Cuando el usuario haga clic sobre el botón "Nuevo alumno", le deberá aparecer la siguiente ventana emergente con el formulario de alta de alumno:

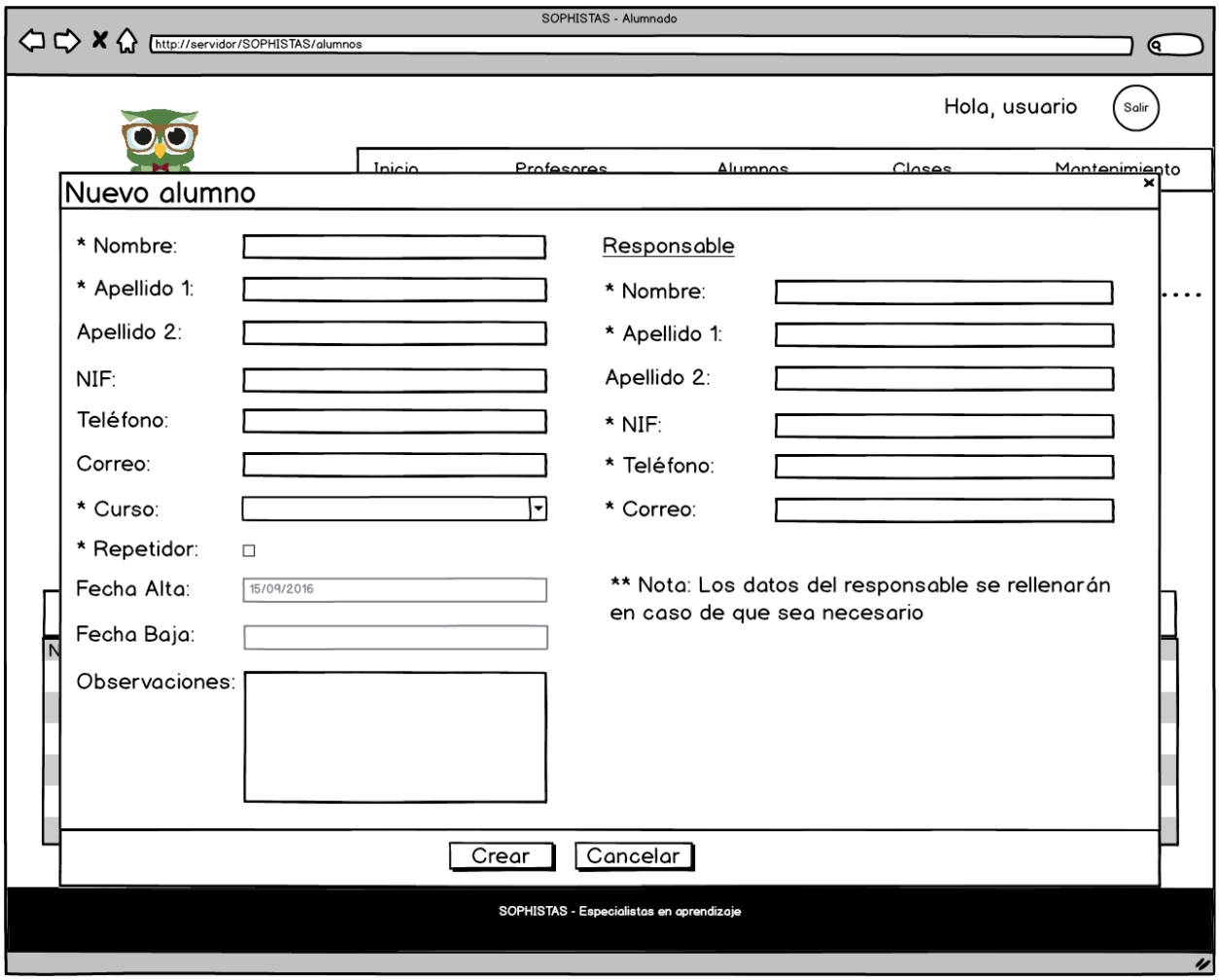


Figura 7-7. Ventana de Nuevo alumno

Cuando el usuario seleccione un alumno de la lista de alumnos (resultado de la búsqueda) y haga clic en el botón "Datos personales", le deberá aparecer una ventana emergente como ésta, pero con los campos rellenos y el título "Datos personales alumno".

Cuando el usuario seleccione un alumno de la lista de alumnos (resultado de la búsqueda) y haga clic en el botón "Clases", le deberá aparecer la siguiente ventana emergente con las clases disponibles para el alumno (las clases de su curso) y marcadas aquellas en la que está matriculado, además del horario del alumno en la parte inferior:

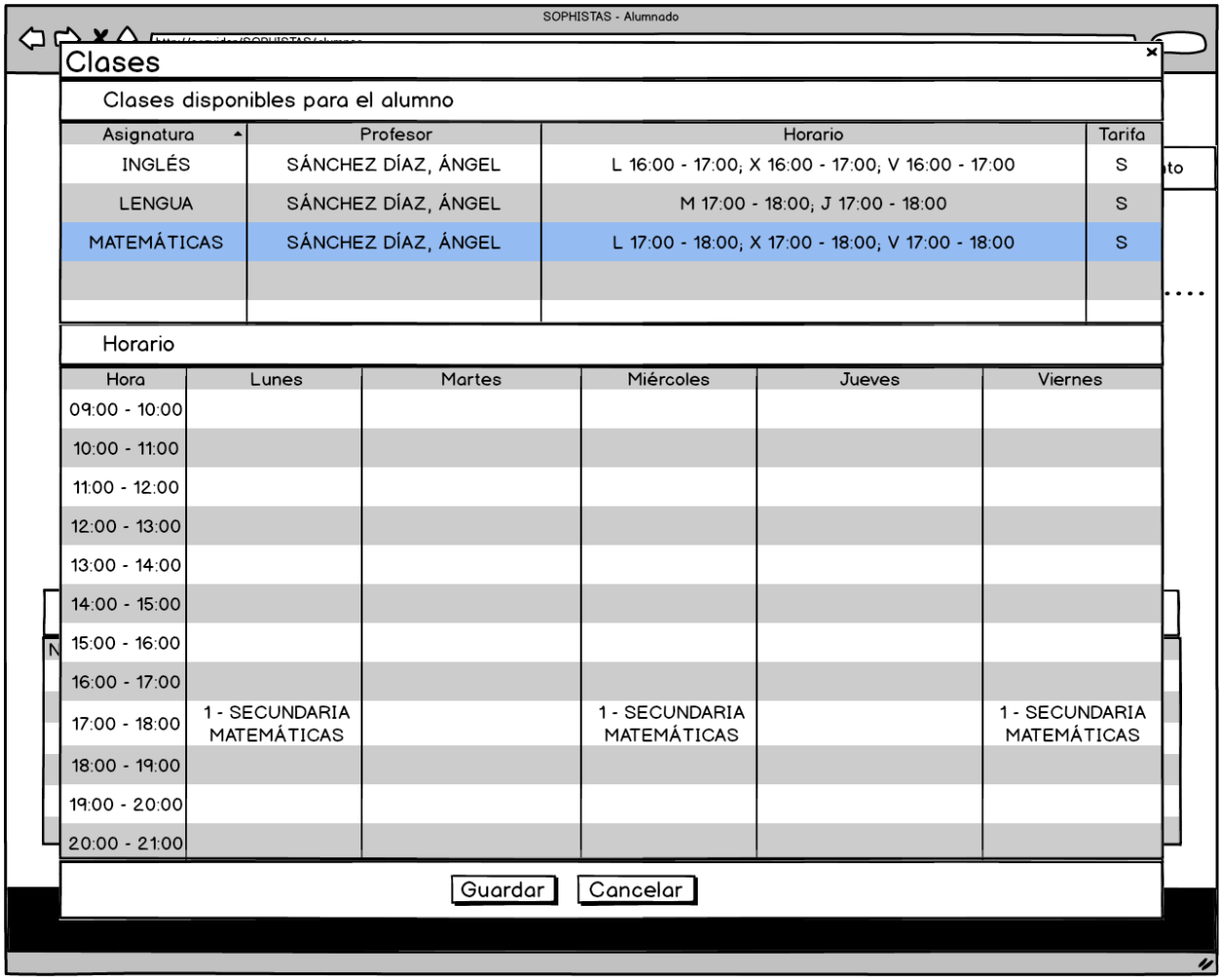


Figura 7-8. Ventana de Clases del alumno

Cuando el usuario seleccione un alumno de la lista de alumnos (resultado de la búsqueda) y haga clic en el botón "Dar de alta / baja", le deberá aparecer una simple ventana emergente de confirmación con los botones "Sí" y "No". Si el alumno está dado de alta, se pedirá confirmación de baja. Si, por el contrario, el alumno está dado de baja, se pedirá confirmación de alta.

7.6.5 Ventanas de Clases

La aplicación deberá ser capaz de mostrar la siguiente ventana con el buscador de clases cuando el usuario entre en la sección "Clases":

Figura 7-9. Ventana de Búsqueda de clases

Cuando el usuario haga clic sobre el botón “Nueva clase”, le deberá aparecer la siguiente ventana emergente con el formulario de alta de clase:

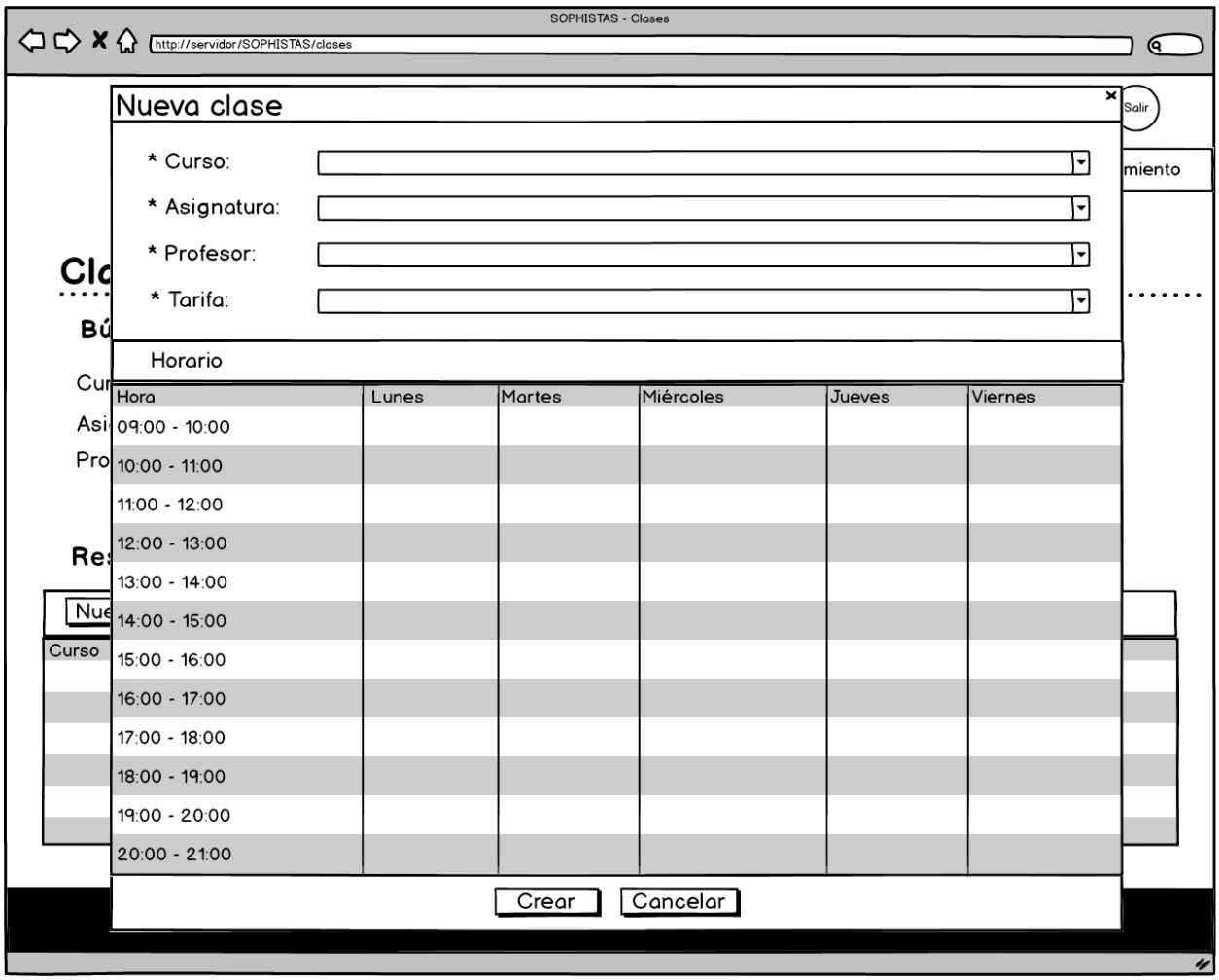


Figura 7-10. Ventana de Nueva clase

Cuando el usuario seleccione una clase de la lista de clases (resultado de la búsqueda) y haga clic en el botón “Editar clase”, le deberá aparecer una ventana emergente como ésta, pero con los campos rellenos y el título “Detalle clase”.

Cuando el usuario seleccione una clase de la lista de clases (resultado de la búsqueda) y haga clic en el botón "Alumnos", le deberá aparecer la siguiente ventana emergente con los alumnos matriculados en esa clase:

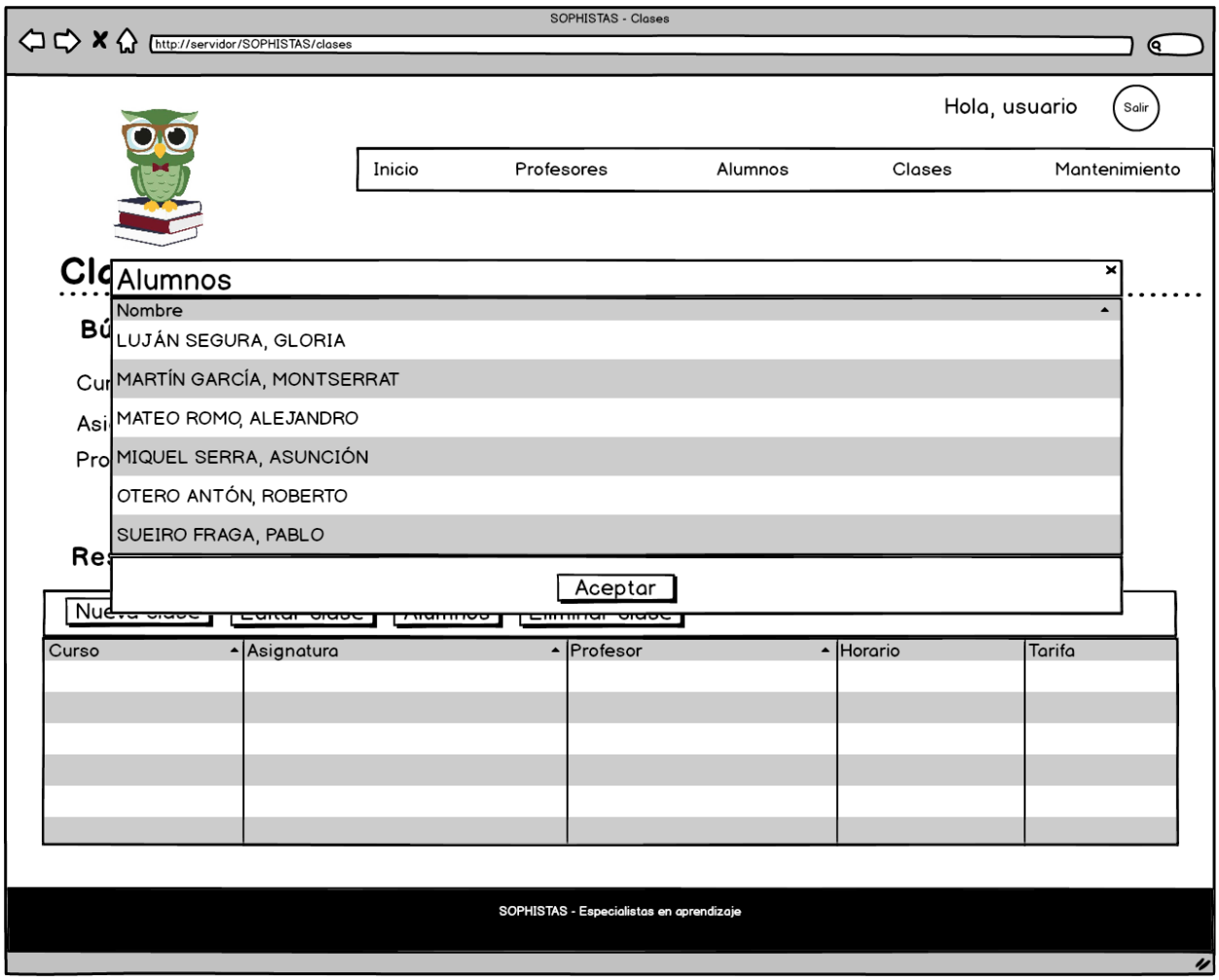


Figura 7-11. Ventana de Alumnos de la clase

Cuando el usuario seleccione una clase de la lista de clases (resultado de la búsqueda) y haga clic en el botón "Eliminar clase", le deberá aparecer una simple ventana emergente de confirmación con los botones "Sí" y "No".

7.6.6 Ventanas de Mantenimiento de Cursos

La aplicación deberá ser capaz de mostrar la siguiente ventana cuando el usuario entre en la sección "Mantenimiento de Cursos":

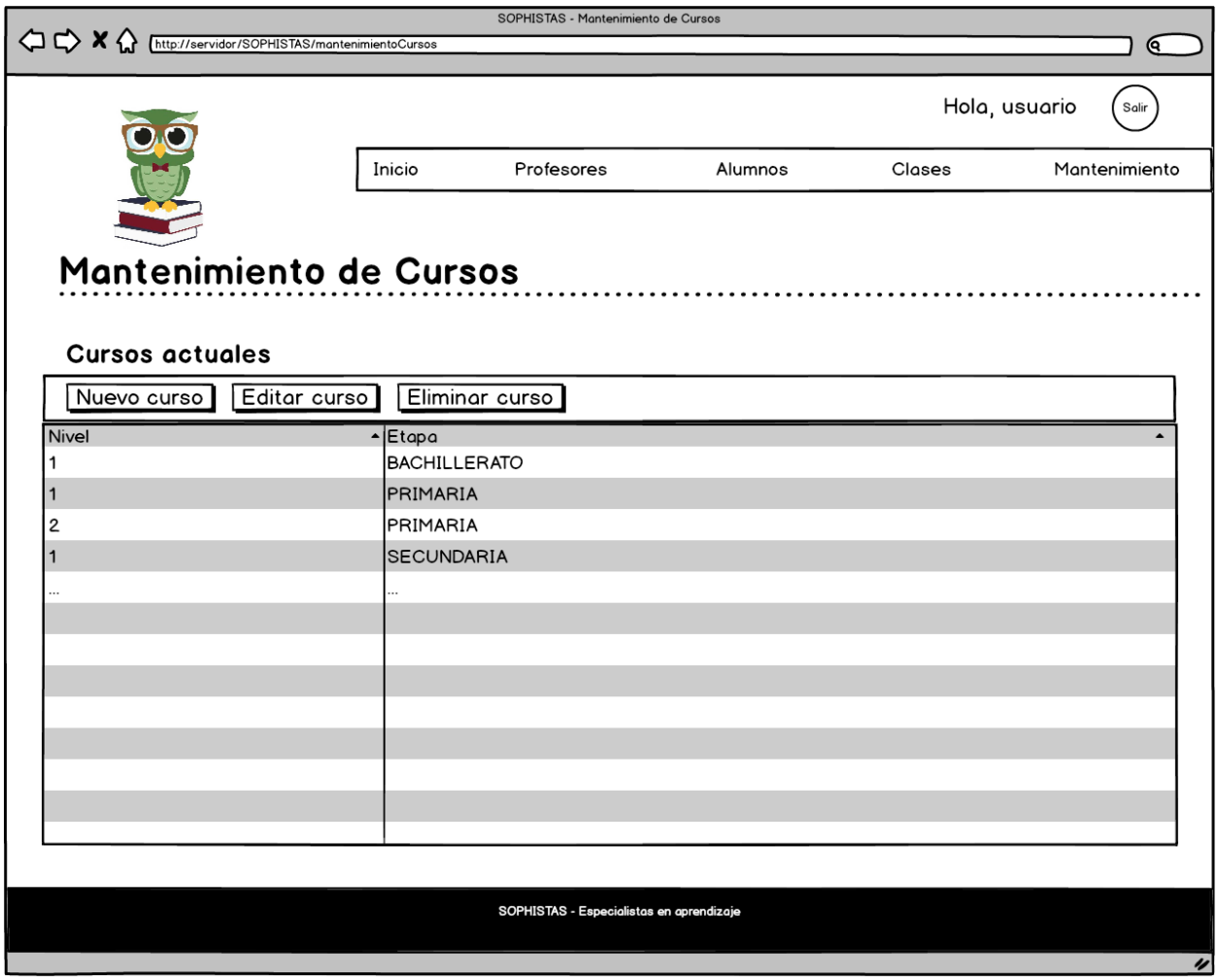


Figura 7-12. Ventana de Cursos actuales

Cuando el usuario haga clic sobre el botón “Nuevo curso”, le deberá aparecer la siguiente ventana emergente con el formulario de alta de curso:

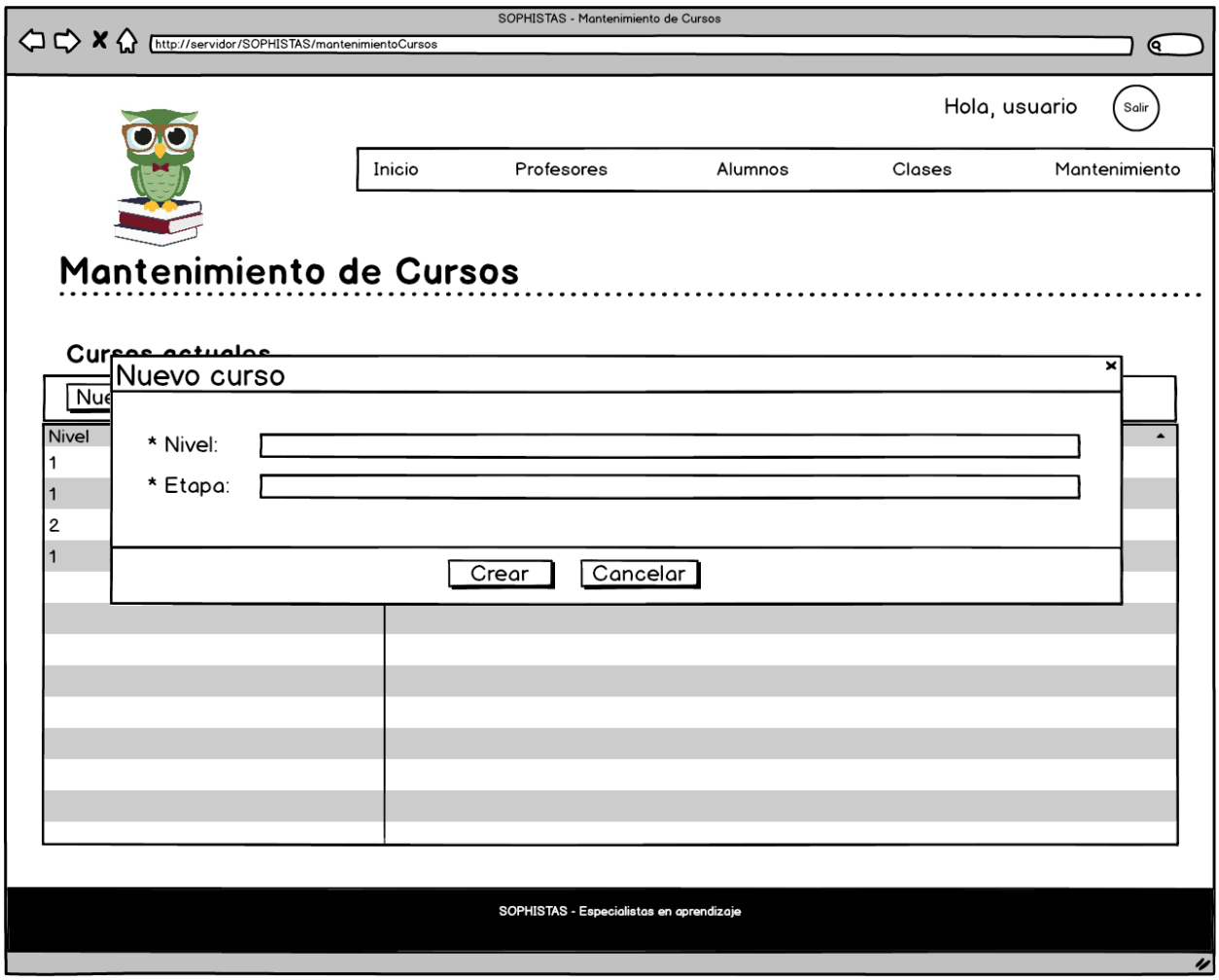


Figura 7-13. Ventana de Nuevo curso

Cuando el usuario seleccione un curso de la lista de cursos actuales y haga clic en el botón “Editar curso”, le deberá aparecer una ventana emergente como ésta, pero con los campos rellenos y el título “Detalle curso”.

Cuando el usuario seleccione un curso de la lista de cursos actuales y haga clic en el botón “Eliminar curso”, le deberá aparecer una simple ventana emergente de confirmación con los botones “Sí” y “No”.

7.6.7 Ventanas de Mantenimiento de Asignaturas

La aplicación deberá ser capaz de mostrar la siguiente ventana cuando el usuario entra en la sección "Mantenimiento de Asignaturas":

Figura 7-14. Ventana de Asignaturas actuales

Cuando el usuario haga clic sobre el botón "Nueva asignatura", le deberá aparecer la siguiente ventana emergente con el formulario de alta de asignatura:

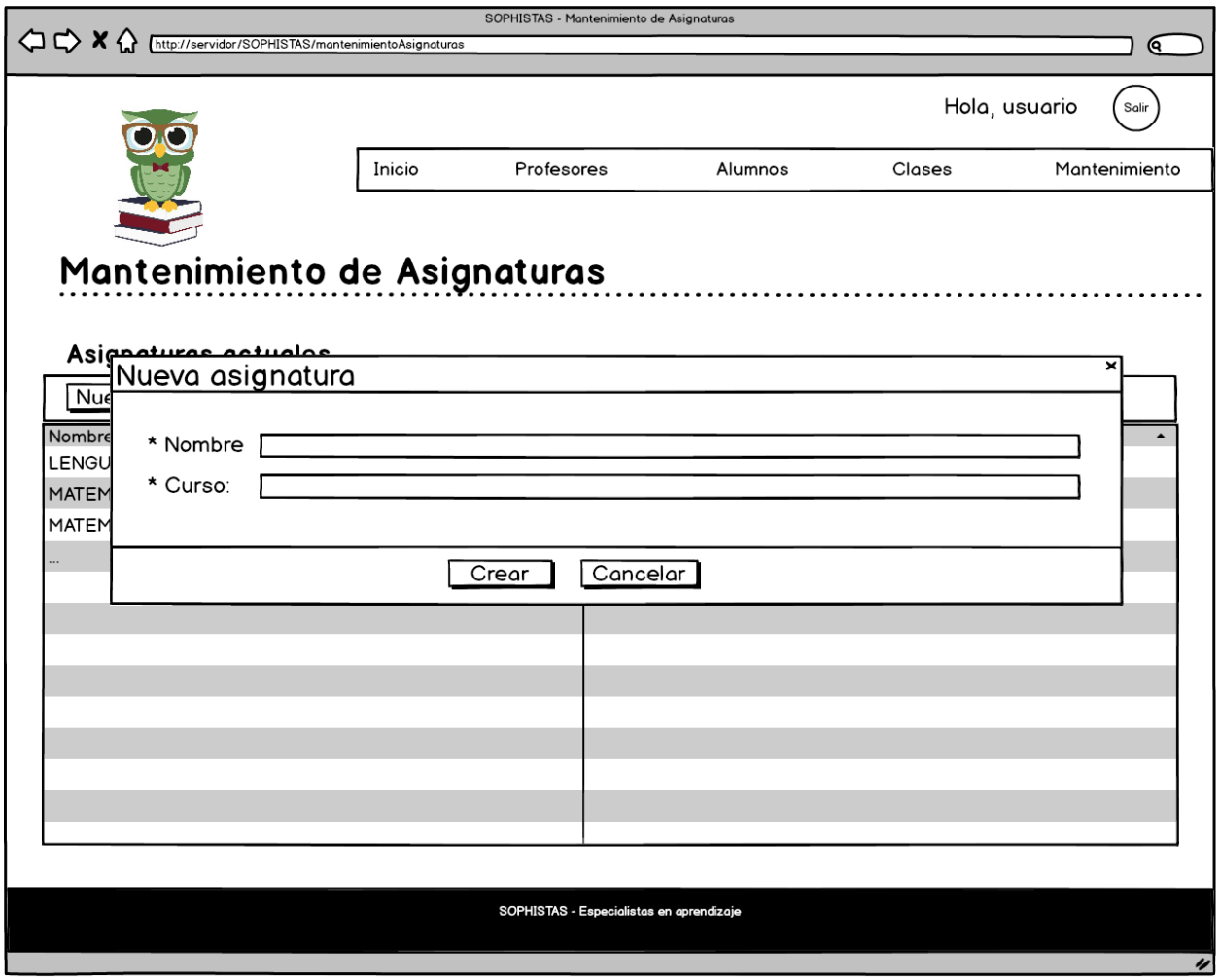


Figura 7-15. Ventana de Nueva asignatura

Cuando el usuario seleccione una asignatura de la lista de asignaturas actuales y haga clic en el botón "Editar asignatura", le deberá aparecer una ventana emergente como ésta, pero con los campos rellenos y el título "Detalle asignatura".

Cuando el usuario seleccione una asignatura de la lista de asignaturas actuales y haga clic en el botón "Eliminar asignatura", le deberá aparecer una simple ventana emergente de confirmación con los botones "Sí" y "No".

7.6.8 Ventanas de Mantenimiento de Tarifas

La aplicación deberá ser capaz de mostrar la siguiente ventana cuando el usuario entra en la sección "Mantenimiento de Tarifas":

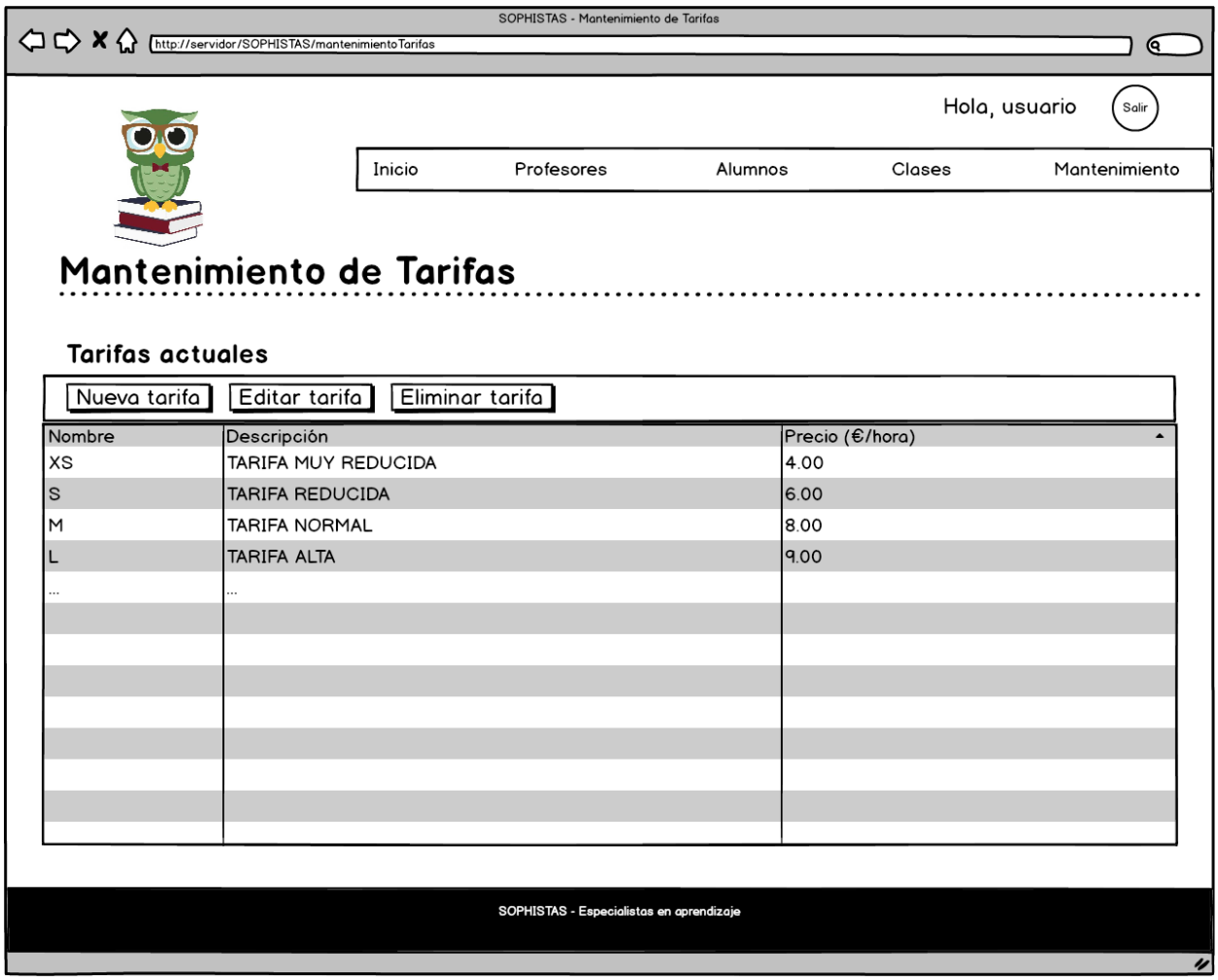


Figura 7-16. Ventana de Tarifas actuales

Cuando el usuario haga clic sobre el botón “Nueva tarifa”, le deberá aparecer la siguiente ventana emergente con el formulario de alta de tarifa:

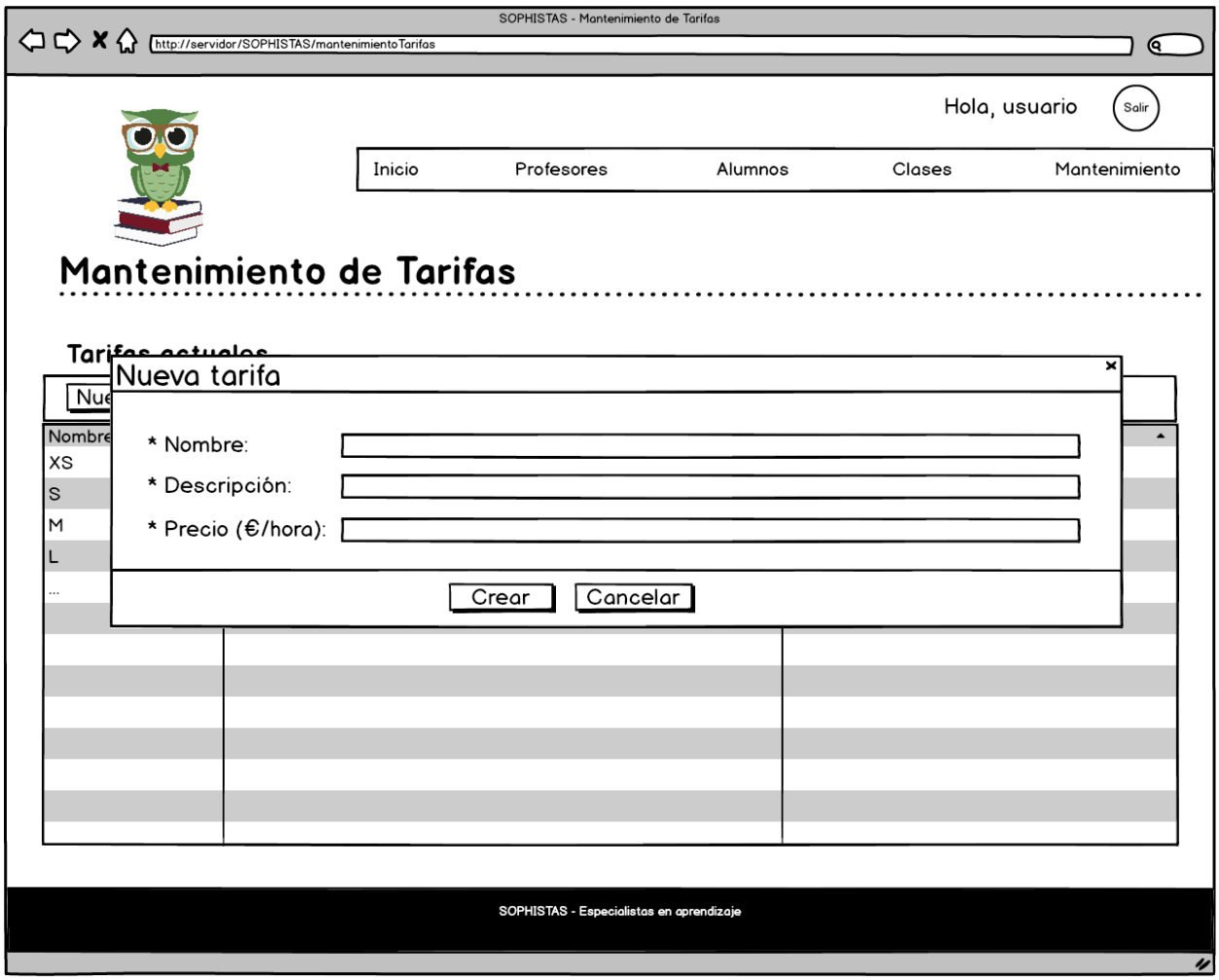


Figura 7-17. Ventana de Nueva tarifa

Cuando el usuario seleccione una tarifa de la lista de tarifas actuales y haga clic en el botón “Editar tarifa”, le deberá aparecer una ventana emergente como ésta, pero con los campos rellenos y el título “Detalle tarifa”.

Cuando el usuario seleccione una tarifa de la lista de tarifas actuales y haga clic en el botón “Eliminar tarifa”, le deberá aparecer una simple ventana emergente de confirmación con los botones “Sí” y “No”.

8 DISEÑO TÉCNICO

Programming without an overall architecture or design in mind is like exploring a cave with only a flashlight: You don't know where you've been, you don't know where you're going, and you don't know quite where you are.

Danny Thorpe

En este capítulo se va a proceder a la realización del diseño técnico de la aplicación "Gestión de Sophistas Academia" y para ello se ha tomado como base el análisis de requisitos confeccionado en el capítulo anterior.

En base a lo expuesto en el presente capítulo se construirá el software de la aplicación.

8.1 Arquitectura del sistema

La aplicación web se va a desarrollar en Java EE 6.

Para el desarrollo de la aplicación se va a usar el entorno Eclipse Java EE IDE for Web Developers (versión Mars). Las dependencias de librerías se resolverán con Apache Maven. Las pruebas unitarias se realizarán con JUnit 4 (integrado con Spring 4).

Para la base de datos se utilizará MySQL Server 5.7. Para el acceso a la base de datos se usará JPA 2 + Hibernate 5, y para los repositorios Spring Data JPA.

Para la inyección de dependencias y orientación a aspectos se hará uso de Spring 4 y se utilizará el modelo MVC a través de Spring MVC 4.

En la parte de la vista se hará uso de las tecnologías Apache Tiles, para la resolución de vistas; JSP, para las vistas; CSS, para el estilo; y Javascript, JQuery y Ext JS, para los scripts en el lado del cliente. La comunicación con Spring MVC será a través de AJAX + JSON en la medida de lo posible.

Para la seguridad de la web se usará Spring Security 4.

El servidor de aplicaciones será Tomcat 7.

En la siguiente figura se pueden ver las distintas tecnologías nombradas que se van a usar de manera esquemática:

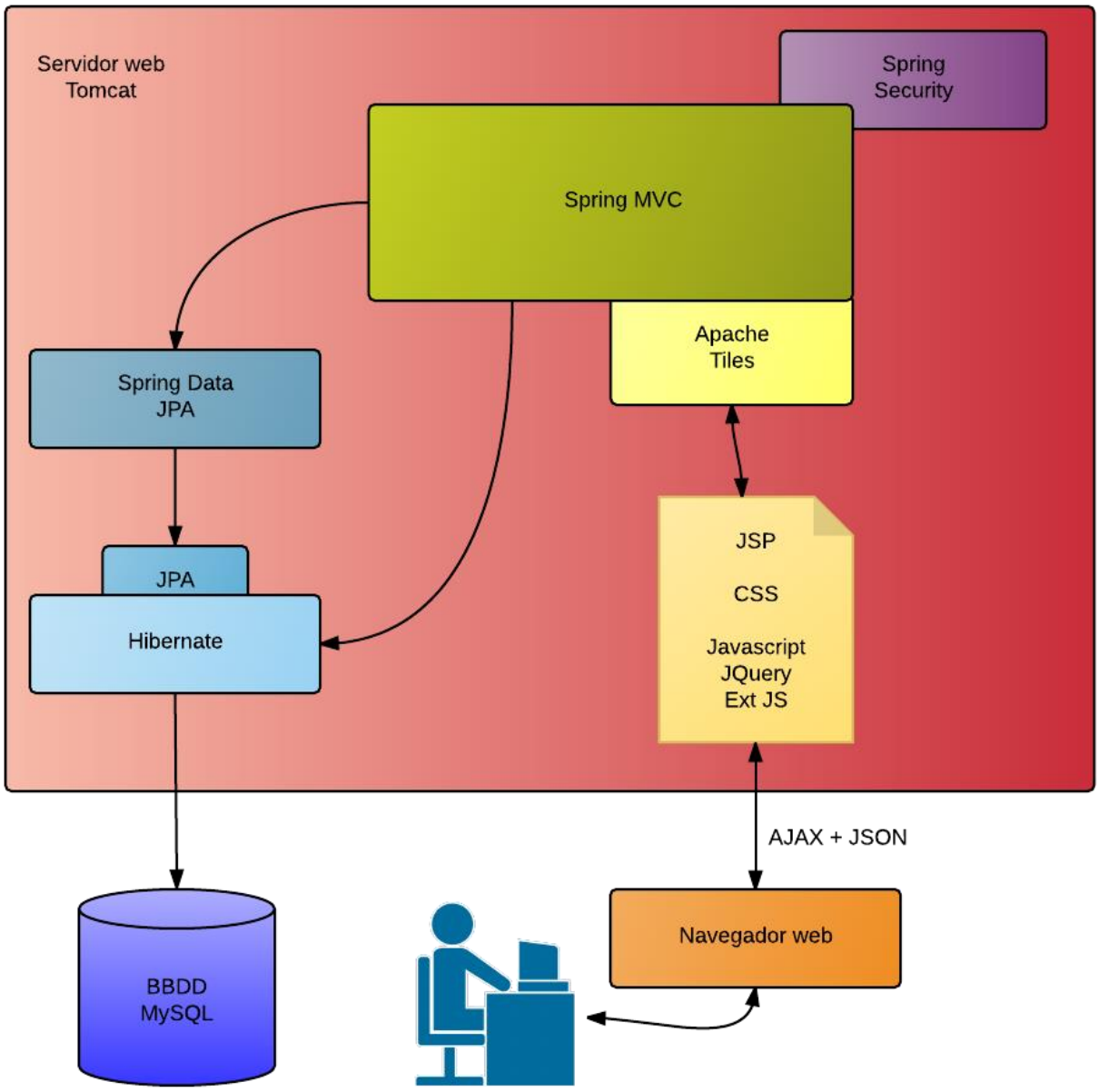


Figura 8-1. Arquitectura del sistema

8.2 Modelo de datos

El modelo de datos se implementará sobre una base de datos MySQL Server llamada SOPHISTAS. El diseño del modelo de datos con las tablas que componen la aplicación, los atributos de las tablas y las relaciones establecidas entre éstas, son descritas en el siguiente diagrama:

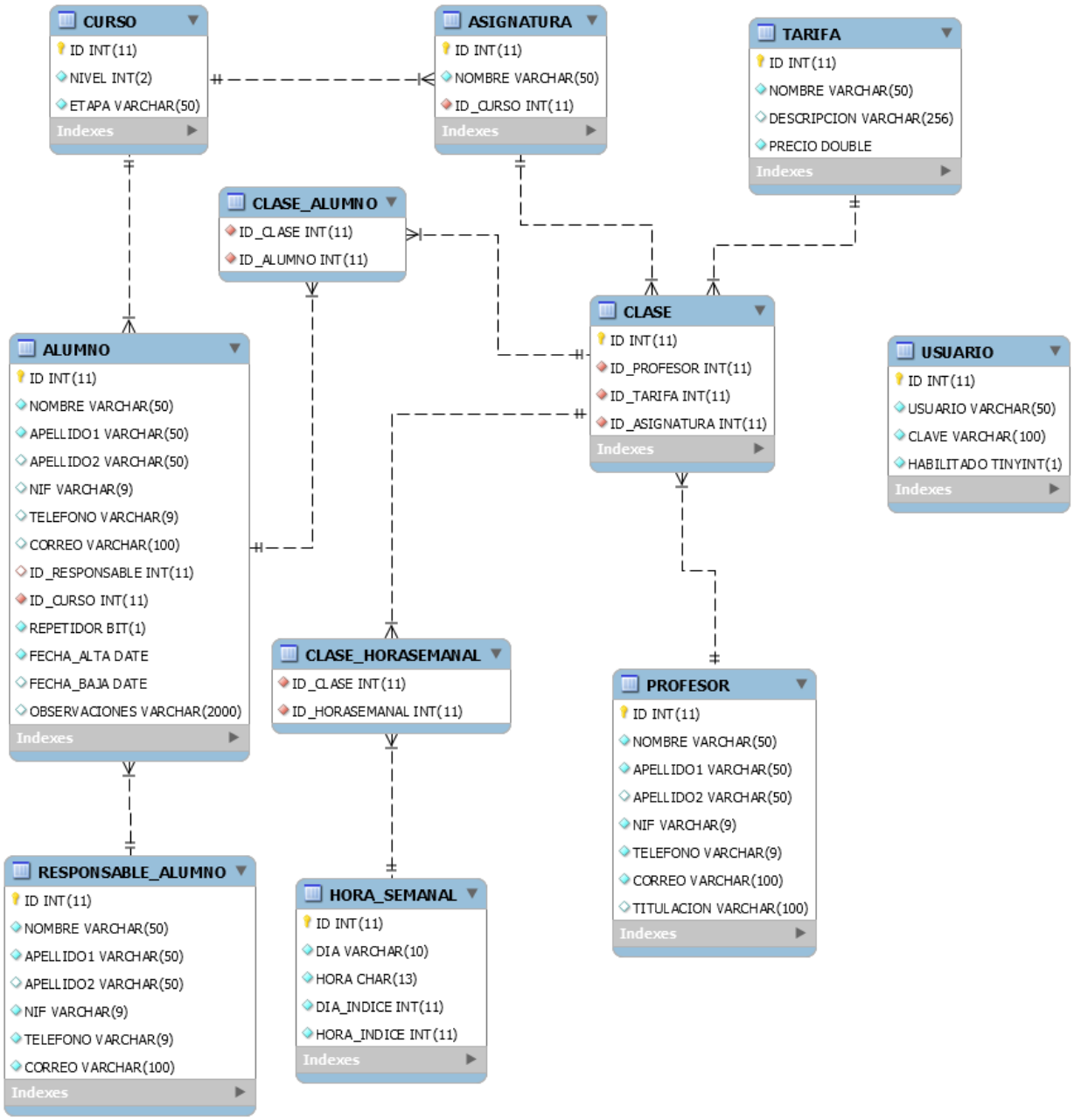


Figura 8-2. Modelo de datos

8.2.1 Descripción de las tablas

8.2.1.1 Tabla USUARIO

En esta tabla se van a almacenar los datos referentes a los usuarios que van a usar la aplicación. Esta información es la que se usará en el login. Sus campos van a ser:

Tabla 8–1 Tabla USUARIO

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada usuario	No	PK[15] AI[16]
USUARIO	VARCHAR(50)	Nombre del usuario	No	UQ[17]
CLAVE	VARCHAR(100)	Clave del usuario (se almacenará encriptada)	No	
HABILITADO	BIT(1)	Bandera para habilitar y deshabilitar al usuario	No	

8.2.1.2 Tabla CURSO

En esta tabla se van a almacenar los datos referentes a los cursos escolares de los que se dan clase en la academia. Sus campos van a ser:

Tabla 8–2 Tabla CURSO

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada curso	No	PK AI
NIVEL	INT(2)	Nivel dentro de la etapa del curso	No	
ETAPA	VARCHAR(50)	Etapa del curso (PRIMARIA, SECUNDARIA, …)	No	

8.2.1.3 Tabla ASIGNATURA

En esta tabla se van a almacenar los datos referentes a las asignaturas de los cursos de las que se dan clases en la academia. Sus campos van a ser:

Tabla 8–3 Tabla ASIGNATURA

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada asignatura	No	PK AI
NOMBRE	VARCHAR(50)	Nombre de la asignatura	No	
ID_CURSO	INT(11)	Referencia al curso de la asignatura	No	FK[18] a CURSO

8.2.1.4 Tabla TARIFA

En esta tabla se van a almacenar los datos referentes a las distintas tarifas que se aplican en la academia. Sus campos van a ser:

[15] *Primary Key*, es decir, clave primaria
[16] *Auto increment*, es decir, autoincremental
[17] *Unique*, es decir, único
[18] *Foreign Key*, es decir, clave foránea

Tabla 8–4 Tabla TARIFA

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada tarifa	No	PK AI
NOMBRE	VARCHAR(50)	Nombre de la tarifa	No	UQ
DESCRIPCION	VARCHAR(256)	Descripción de la tarifa	Sí	
PRECIO	DOUBLE	Precio de la clase con esta tarifa, en €/hora	No	

8.2.1.5 Tabla PROFESOR

En esta tabla se van a almacenar los datos referentes a los profesores de la academia. Sus campos van a ser:

Tabla 8–5 Tabla PROFESOR

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada profesor	No	PK AI
NOMBRE	VARCHAR(50)	Nombre del profesor	No	
APELLIDO1	VARCHAR(50)	Apellido 1 del profesor	No	
APELLIDO2	VARCHAR(50)	Apellido 2 del profesor	Sí	
NIF	CHAR(9)	NIF del profesor	No	UQ
TELEFONO	CHAR(9)	Teléfono del profesor	No	
CORREO	VARCHAR(100)	Correo del profesor	No	
TITULACION	VARCHAR(100)	Titulación del profesor	Sí	

8.2.1.6 Tabla CLASE

En esta tabla se van a almacenar los datos referentes a las clases que se imparten en la academia. Sus campos van a ser:

Tabla 8–6 Tabla CLASE

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada clase	No	PK AI
ID_PROFESOR	INT(11)	Referencia al profesor de la clase	No	FK a PROFESOR
ID_TARIFA	INT(11)	Referencia a la tarifa de la clase	No	FK a TARIFA
ID_ASIGNATURA	INT(11)	Referencia a la asignatura de la clase	No	FK a ASIGNATURA

8.2.1.7 Tabla ALUMNO

En esta tabla se van a almacenar los datos referentes a los alumnos de la academia. Sus campos van a ser:

Tabla 8–7 Tabla ALUMNO

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada alumno	No	PK AI
NOMBRE	VARCHAR(50)	Nombre del alumno	No	
APELLIDO1	VARCHAR(50)	Apellido 1 del alumno	No	
APELLIDO2	VARCHAR (50)	Apellido 2 del alumno	Sí	
NIF	CHAR(9)	NIF del alumno	Sí	
TELEFONO	CHAR(9)	Teléfono del alumno	Sí	
CORREO	VARCHAR(100)	Correo del alumno	Sí	
ID_RESPONSABLE	INT(11)	Referencia al responsable del alumno	Sí	FK a RESPONSABLE
ID_CURSO	INT(11)	Referencia al curso del alumno	No	FK a CURSO
REPETIDOR	BIT(1)	Condición de repetidor del alumno	No	
FECHA_ALTA	DATE	Fecha de alta del alumno en la academia	No	
FECHA_BAJA	DATE	Fecha de baja del alumno en la academia	Sí	
OBSERVACIONES	VARCHAR(2000)	Observaciones del alumno	Sí	

8.2.1.8 Tabla RESPONSABLE_ALUMNO

En esta tabla se van a almacenar los responsables de los alumnos (padre, tutor, ...) cuando sea necesario. Sus campos van a ser:

Tabla 8–8 Tabla RESPONSABLE_ALUMNO

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada responsable	No	PK AI
NOMBRE	VARCHAR(50)	Nombre del responsable	No	
APELLIDO1	VARCHAR(50)	Apellido 1 del responsable	No	
APELLIDO2	VARCHAR(50)	Apellido 2 del responsable	Sí	
NIF	CHAR(9)	NIF del responsable	No	UQ
TELEFONO	CHAR(9)	Teléfono del responsable	No	
CORREO	VARCHAR(100)	Correo del responsable	No	

8.2.1.9 Tabla HORA_SEMANAL

En esta tabla se van a almacenar las diferentes horas del día desde las 09:00 hasta las 21:00 de los días entre el lunes y el viernes. Sus campos son:

Tabla 8–9 Tabla HORA_SEMANAL

Atributo	Tipo	Descripción	Permite Null	Otros
ID	INT(11)	Identificador único para cada hora semanal	No	PK AI
DIA	VARCHAR(10)	Nombre del día de la semana	No	
HORA	CHAR(13)	Hora del día (ej.: 09:00 – 10:00)	No	
DIA_INDICE	INT(11)	Índice del día de la semana (se va a usar en los grid de los horarios)	No	
HORA_INDICE	INT(11)	Índice de la hora del día (se va a usar en los grid de los horarios)	No	

8.2.1.10 Tabla CLASE_ALUMNO

Esta tabla va a ser una tabla de unión entre la tabla CLASE y la tabla ALUMNO. Sus campos van a ser:

Tabla 8–10 Tabla CLASE_ALUMNO

Atributo	Tipo	Descripción	Permite Null	Otros
ID_CLASE	INT(11)	Referencia a una clase	No	FK a CLASE
ID_ALUMNO	INT(11)	Referencia a un alumno	No	FK a ALUMNO

8.2.1.11 Tabla CLASE_HORASEMANAL

Esta tabla va a ser una tabla de unión entre la tabla CLASE y la tabla HORA_SEMANAL. Sus campos van a ser:

Tabla 8–11 Tabla CLASE_HORASEMANAL

Atributo	Tipo	Descripción	Permite Null	Otros
ID_CLASE	INT(11)	Referencia a una clase	No	FK a CLASE
ID_HORASEMANAL	INT(11)	Referencia a una hora semanal	No	FK a HORA_SEMANAL

8.3 Modelo de capas

El diseño de la aplicación va a seguir el siguiente modelo de capas:

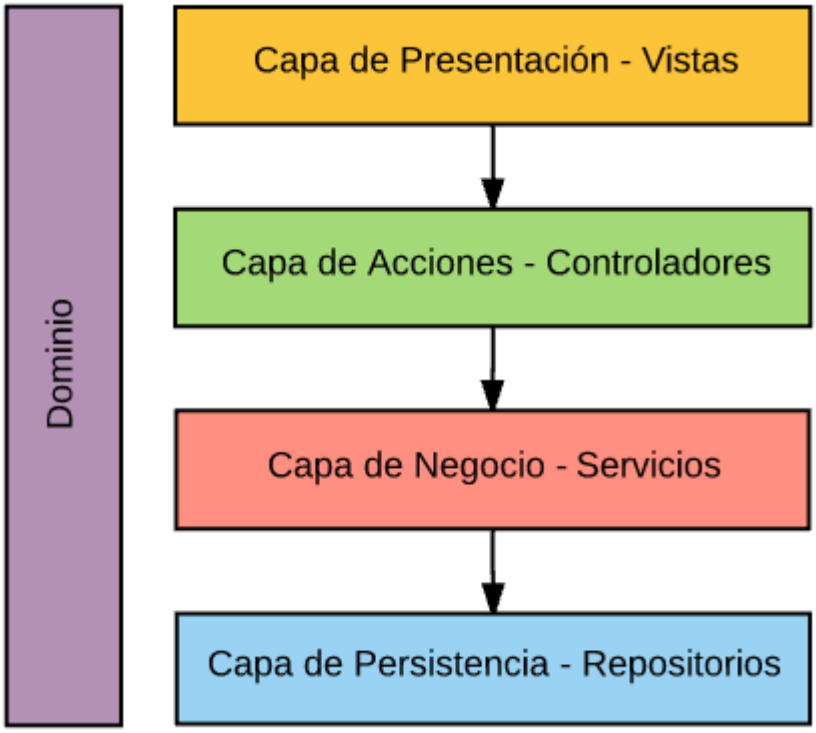


Figura 8-3. Modelo de capas

8.4 Estructura de paquetes

Las clases Java van a ir dentro de la carpeta src/main/java en los siguientes paquetes:

- Configuración: *com.sophistas.configuracion*
- Dominio (Entidades): *com.sophistas.dominio*
- Capa de Persistencia (Repositorios): *com.sophistas.persistencia*
- Capa de Acciones (Controladores): *com.sophistas.accion*
- Capa de Negocio (Servicios): *com.sophistas.negocio*
- Excepciones: *com.sophistas.excepciones*
- Utilidades: *com.sophistas.utilidades*

8.4.1 Configuración

En el paquete *com.sophistas.configuracion* van a ir las diferentes clases de configuración de la aplicación.

- ***SophistasWebAppInitializer***

  Clase para configurar el *DispatcherServlet*.
- ***SecurityWebInitializer***

  Clase para configurar el *DelegatingFilterProxy*.
- ***WebConfig***

  Clase para configurar el contexto *servlet* .
- ***RootConfig***

  Clase para configurar el contexto raíz, en concreto los componentes del nivel de datos.

- ***SecurityConfig***

  Clase para configurar el contexto raíz, en concreto los componentes relativos a la seguridad.

Se va a optar por la configuración en Java, en lugar de la configuración en XML, ya que es más potente, ofrece seguridad de tipos y permite la refactorización.

8.4.2 Dominio

En el paquete *com.sophistas.dominio* van a ir las entidades del sistema. A cada tabla del modelo de datos se le va a asignar una entidad (excepto a las tablas de unión y la tabla USUARIO, que no entra dentro de la lógica de negocio sino en el ámbito de la seguridad). Estas entidades se van a mapear con JPA, usando Hibernate como ORM subyacente y para los metadatos de mapeo se van a usar anotaciones Java (en lugar de usar XML).

A la hora de configurar la fábrica de administradores de entidades (*EntityManagerFactory*) en la clase *RootConfig* (*del paquete com.sophistas.configuracion*) se deberá indicar que es en este paquete donde se encuentran las entidades de dominio.

Las clases de este paquete van a seguir las siguientes características:

- Uno de los constructores de la clase va a ser el constructor por defecto, público y sin ningún tipo de argumentos, de manera que Hibernate pueda crear instancias mediante reflectividad.
- La clase no va a ser ni final ni interna, siempre de primer nivel.
- La clase va a implementar la interfaz *java.io.Serializable*.
- Por cada propiedad de la clase se va a tener un *get/set* asociado.
- Las propiedades van a ser de tipo nullables (*Integer*, *Long*, etc.).
- Para la configuración de mapeo de la clase con la tabla correspondiente en la base de datos se emplean anotaciones que se incorporan en la clase.
- El nombre de la clase coincidirá con el de la tabla, la primera letra en mayúsculas y el resto en minúsculas, y aplicando el mecanismo del *UpperCamelCase*[19] si el nombre es compuesto.
- La clase va a llevar la anotación *@Entity*.

8.4.2.1 Subpaquete *com.sophistas.dominio.estadisticas*

En este paquete van a ir dos clases simples (POJO) que se van a usar para recoger los resultados de las consultas de las estadísticas de los alumnos, hechas para la representación de las gráficas que irán en la pantalla de inicio.

8.4.3 Capa de Persistencia

En el paquete *com.sophistas.persistencia* van a ir las clases e interfaces usadas para el acceso a la base de datos, es decir, los repositorios.

Para los repositorios se va a usar Spring Data JPA. Para cada entidad JPA (clases del paquete *com.sophistas.dominio*) se va a crear un repositorio Spring Data, que no es más que una interfaz que especializa *JpaRepository*, la cual proporciona de serie las operaciones CRUD más habituales y que por tanto no se tienen que implementar.

La convención que se va a seguir para nombrar cada repositorio va a ser utilizar el nombre de la entidad con el sufijo "Repository". Por ejemplo, se tendrá el repositorio *TarifaRepository*.

Cuando sea necesario añadir métodos personalizados a un repositorio se va a usar la anotación *@Query* para proporcionar a Spring Data la consulta que ejecutar, utilizando para ello el lenguaje JPQL.

Si se requiere una funcionalidad para el repositorio que no se puede describir con la anotación *@Query*, se

[19] La primera letra de cada una de las palabras es mayúsculas.

trabajará con JPA a un nivel inferior, usando directamente el *EntityManager*. En estos casos se tendrá:

- La interfaz *EntidadRepository* que extenderá de *JpaRepository<Entidad, ClaseId>* y de *IEntidadCustomRepository*.
- La interfaz *IEntidadCustomRepository* que definirá estos métodos de acceso a la base de datos más complejos.
- La clase *EntidadRepositoryImpl* que implementará la interfaz *IEntidadCustomRepository*

.

8.4.4 Capa de Acciones

En el paquete *com.sophistas.accion* van a ir las clases que tomen el papel de controlador, esto es, las clases encargadas de interceptar las solicitudes HTTP del cliente y de traducir dichas solicitudes en operaciones específicas de negocio a ser realizadas.

Cada clase controlador va a:

- Utilizar el sufijo "Controller" en su nombre.
- Implementar una interfaz cuyo nombre será el mismo que el del controlador y con el prefijo "I", y que se encontrará en el subpaquete *com.sophistas.accion.interfaces*.
- Llevar la anotación *@Controller*.
- En la medida de lo posible los métodos no llevarán a cabo tareas de procesamiento, sino que se delegará la responsabilidad de la lógica de negocio a los objetos de servicio (paquete *com.sophistas.negocio*).

  Por tanto, su labor será:

  - Validar los datos que se le pasan. Si son parámetros de formularios se harán las correspondientes validaciones "simples" y "dinámicas".
  - Hacer uso del correspondiente objeto de servicio para toda la lógica de negocio.
  - Devolver el correspondiente resultado que será: el nombre lógico de una vista con su correspondiente modelo; o un objeto Java con la información necesaria en el cliente, y que llegará a éste en formato JSON.

En la clase *WebConfig* (*del paquete com.sophistas.configuracion*) se deberá indicar que este paquete debe ser analizado por el analizador de componentes de Spring.

8.4.4.1 Subpaquete *com.sophistas.dto*

En este paquete van las clases DTO (*Data Transfer Object*), que se usan para facilitar el paso de información desde y hacia las vistas. Estas clases van a tener las siguientes características:

- Van a ser clases POJO que implementen la interfaz *Serializable*.
- Van a utilizar el sufijo "DTO" en su nombre.

8.4.5 Capa de Negocio

El paquete *com.sophistas.negocio* contiene las clases encargadas de consumir las funciones que ofrece la capa de persistencia (los repositorios) y proporcionar la funcionalidad necesaria a la capa de acción (los controladores).

Para cada entidad va a haber una clase de negocio que va a:

- Utilizar el sufijo "Manager" en su nombre.
- Implementar una interfaz cuyo nombre será el mismo que el de la clase y con el prefijo "I", y que se encontrará en el subpaquete *com.sophistas.negocio.interfaces*.

- Llevar la anotación *@Service*.

En la clase *WebConfig* (*del paquete com.sophistas.configuracion*) se deberá indicar que este paquete debe ser analizado por el analizador de componentes de Spring.

8.4.5.1 Subpaquete *com.sophistas.negocio.validacion*

En este paquete van a ir las clases que se van a usar a la hora de validar los parámetros de los formularios que llegan a los controladores.

8.4.6 Excepciones

El paquete *com.sophistas.excepciones* va a incluir las diferentes excepciones personalizadas usadas en la aplicación.

Cada clase de este paquete va a:

- Utilizar el sufijo "Exception".
- Extender la clase *Exception*, que es la clase base de las excepciones en Java.

8.4.7 Utilidades

El paquete *com.sophistas.utilidades* va a contener la clase *Utiles*. Esta clase no va a requerir instanciación, y va a tener una serie de métodos estáticos (*static*) que realicen funciones que se vayan a reutilizar a lo largo de la aplicación.

8.5 Capa de presentación

La capa de presentación la constituyen las vistas que se van a integrar con el resto de capas.

Los ficheros correspondientes a la capa de presentación van a ir dentro de la carpeta src/main/webapp.

8.5.1 Páginas JSP

En la aplicación se va a usar JSP para la tecnología de vista.

Para el diseño se va a hacer uso de Apache Tiles como motor de diseño para reducir la duplicación de elementos comunes. En la carpeta WEB-INF/layout van a ir:

- El fichero tiles.xml que va a contener las especificaciones de mosaico.
- El fichero page.jsp, que va a ser la plantilla que va a usar el mosaico base.
- Los ficheros menu.jsp y seccioninferior.jsp, que van a contener el menú de la aplicación y la sección inferior de las páginas, ambos comunes para todas las pantallas.
- El fichero scripts.jsp que va a tener los enlaces a los distintos ficheros CSS y las referencias a los distintos ficheros Javascript usados desde todas las páginas de la aplicación. (page.jsp incluirá a scripts.jsp)

El resto de ficheros JSP van a ir en la carpeta raíz y contendrán la sección central de las diferentes páginas de la aplicación.

El uso de Apache Tiles se configurará en la clase *WebConfig* (del paquete *com.sophistas.configuracion*).

8.5.2 Resto de elementos de la capa de presentación

La carpeta src/main/webapp también va a contener estas carpetas:

- fuentes: En esta carpeta va a ir la fuente Font Awesome, la cual se va a usar para los diferentes iconos de la aplicación.

- imagenes: En esta carpeta van a ir las diferentes imágenes que se van a incrustrar en las vistas de la aplicación web.
- estilos: En esta carpeta van a ir los ficheros con código CSS usados para aplicar un estilo propio a la aplicación.
- ext: Esta carpeta va a contener la biblioteca Ext JS, la cual se va a usar con licencia *Open source* para conseguir una web dinámica e interactiva con el usuario.
- jquery: Esta carpeta va a contener la biblioteca jQuery, la cual se va a usar ya que simplifica en gran medida el código Javascript.
- js: En esta carpeta van a ir los diferentes ficheros JS. Cada JSP tendrá su propio fichero JS con el mismo nombre, variando únicamente la extensión. Los métodos comunes se guardarán en un fichero utils.js que sí podrá ser llamado por varias páginas diferentes.

8.6 Seguridad

Para dotar de seguridad a la aplicación web se va a hacer uso de Spring Security.

La aplicación web sólo va a poder ser usada por aquellos usuarios que tengan las credenciales necesarias. Como se vió en el apartado 8.2.1.1, en la tabla USUARIO van a estar almacenados los datos referentes a dichos usuarios: el nombre de usuario y la clave (que se almacenará encriptada).

Se definirá una página de *login* (login.jsp) donde el usuario tendrá que introducir dichas credenciales. Todas las solicitudes que se hagan, excepto las solicitudes a la página de *login*, requerirán de autenticación. En caso de hacer una solicitud sin tener la autenticación, provocará que al usuario se le redirija a la página de *login*.

La configuración referente a la seguridad irá en las clases *SecurityWebInitializer* **y** *SecurityConfig* del paquete *com.sophistas.seguridad* como se indicó en el apartado 8.4.1.

9 MANUAL DE USUARIO

One of my biggest rules is, if it needs an instruction manual it's probably designed wrong.

Mark Tacchi

Steve Jobs, genio de la informática y la electrónica, fue de los primeros en plantear que un buen producto probablemente no necesita de un manual de usuario, a día de hoy, pocos lo discuten ya. La aplicación "Gestión de Sophistas Academia" se ha enfocado y desarrollado con la idea siempre de tener como resultado una aplicación web intuitiva, sencilla y completa.

El presente manual ha sido elaborado con la intención de ayudar si es necesario al usuario de la aplicación "Gestión de Sophistas Academia".

9.1 Acceso a la aplicación

El acceso a la aplicación se hará a través de la dirección:

http://servidor/SOPHISTAS/

La primera tarea del usuario será identificarse en el sistema. Para ello, deberá rellenar correctamente los campos "Usuario" y "Clave" del siguiente formulario:

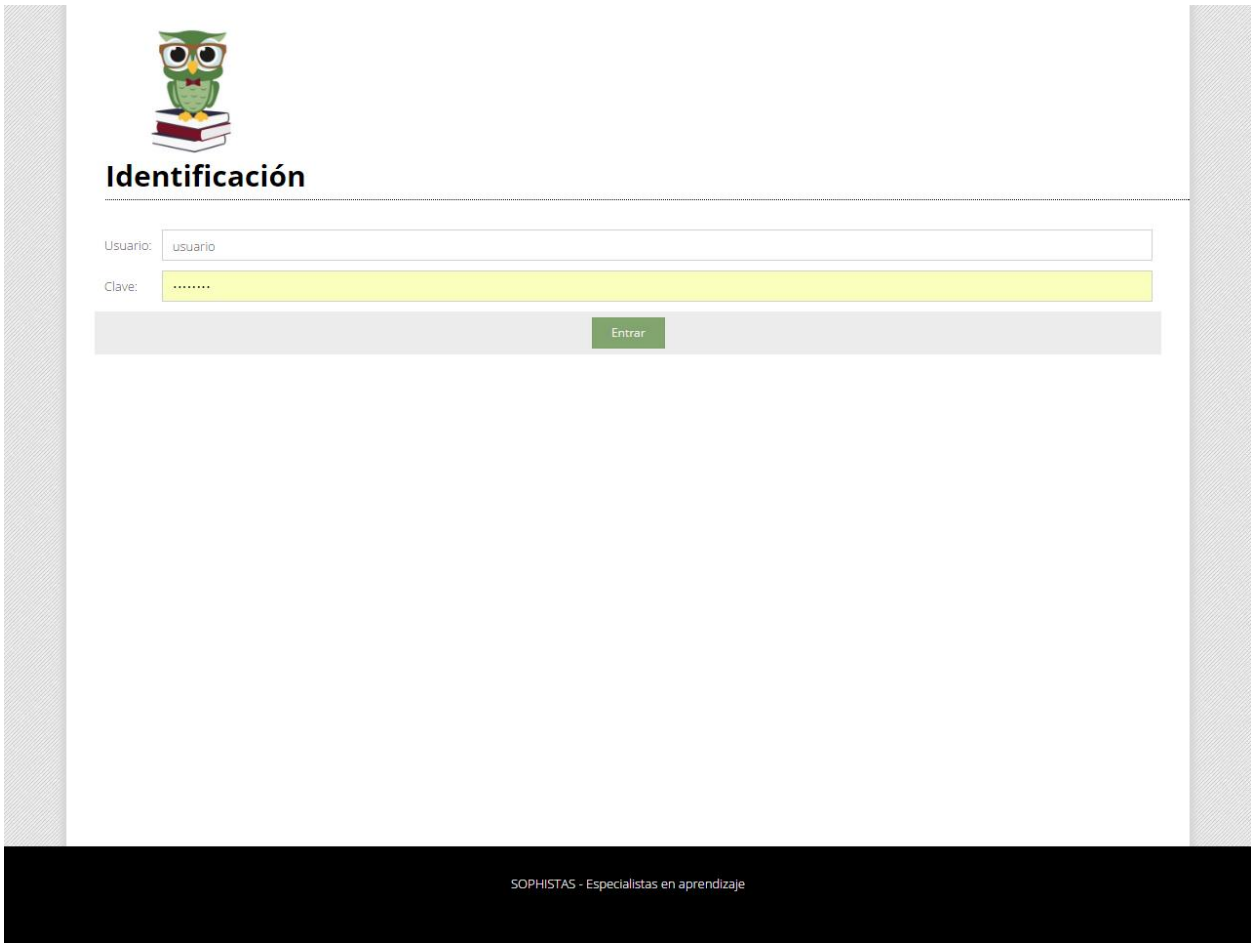


Figura 9-1. Formulario de Identificación

Si el usuario rellenara de forma errónea alguno de los campos, la aplicación mostrará el siguiente error:

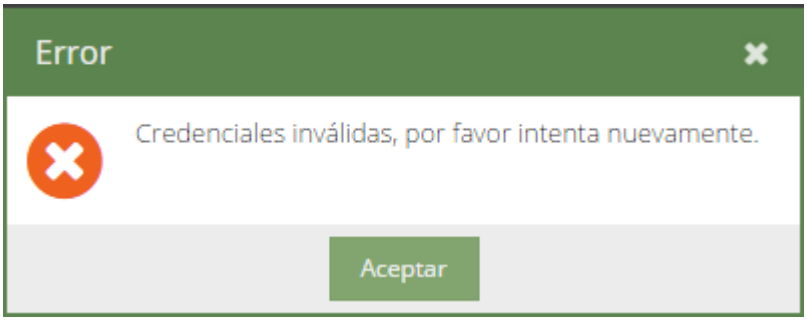


Figura 9-2. Error de Identificación

El usuario deberá volver a introducir los campos “Usuario” y “Clave”.

Cuando se autentique correctamente, se le mostrará al usuario la siguiente pantalla de Inicio.

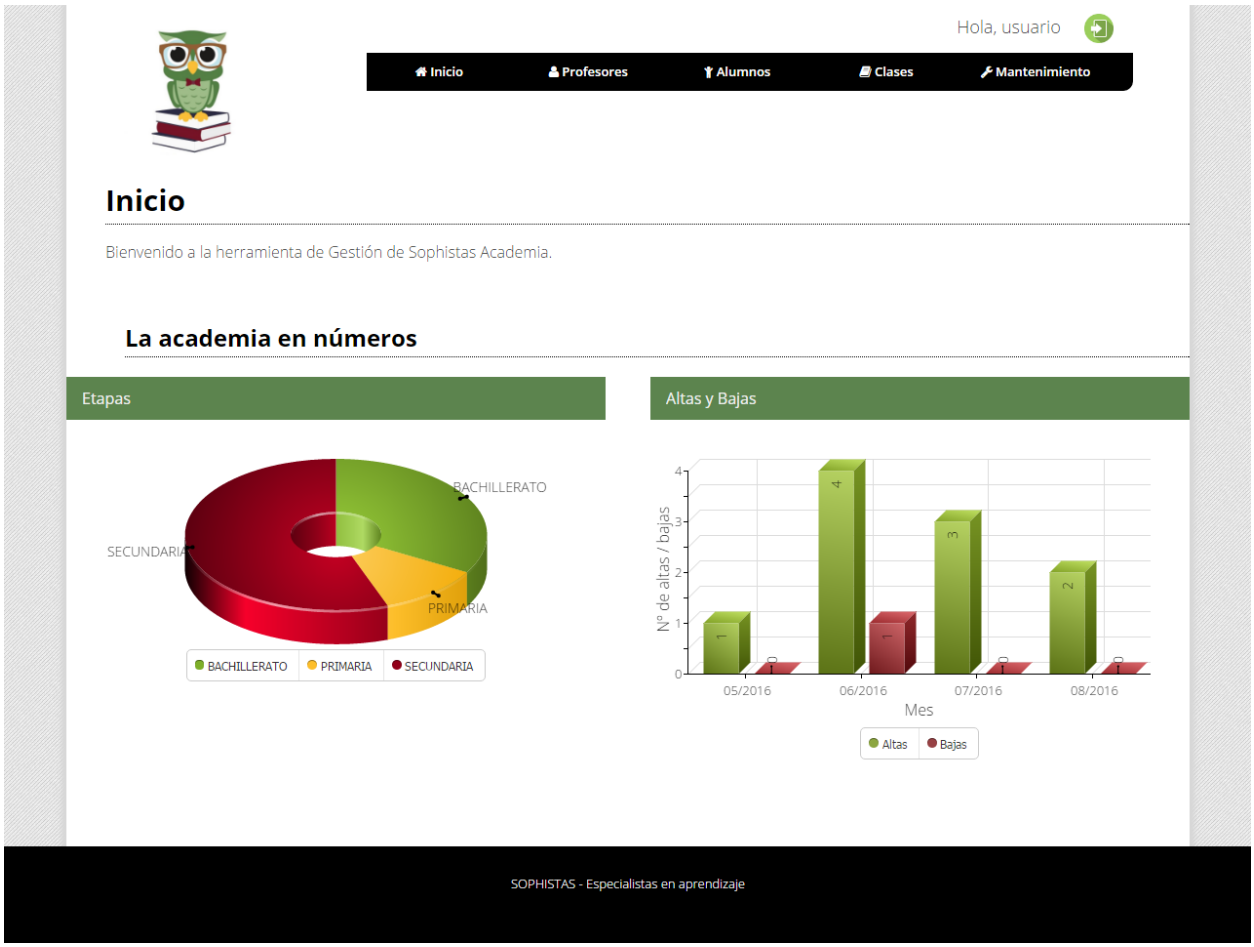


Figura 9-3. Pantalla de Inicio

En esta pantalla la aplicación le da la bienvenida al usuario, además de mostrarle dos gráficos con estadísticas de la academia:

- "Etapas": Gráfico que muestra el porcentaje de alumnos matriculados en la academia por etapas (Primaria, Secundaria, …)
- "Altas y Bajas": Gráfico que muestra el número de altas y bajas de alumnos que ha habido en la academia en los últimos cuatro meses.

9.2 Menú

Una vez que el usuario ha iniciado sesión en la aplicación (ha introducido correctamente sus credenciales), mientras navega por las distintas pantallas de ésta, siempre va a tener accesible el menú en la parte superior derecha.

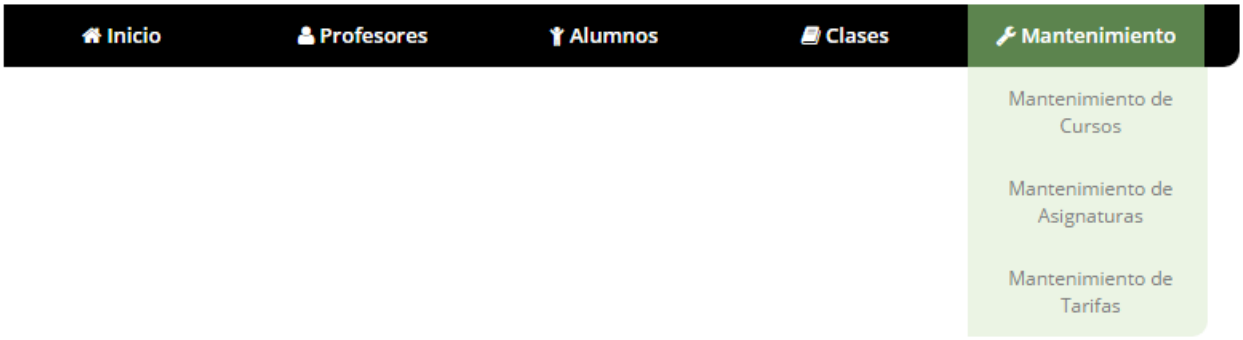


Figura 9-4. Menú

Desde el menú, el usuario podrá acceder a las diferentes secciones de la aplicación:

- "Inicio": Pulsando la opción "Inicio" el usuario accederá a la pantalla de Inicio descrita en el apartado 9.1.
- "Profesorado": Pulsando la opción "Profesores", el usuario accederá a la sección "Profesorado" que se describirá en el apartado 9.3.
- "Alumnado": Pulsando la opción "Alumnos", el usuario accederá a la sección "Alumnado" que se describirá en el apartado 9.4.
- "Clases": Pulsando la opción "Clases", el usuario accederá a la sección "Clases" que se describirá en el apartado 9.5.
- "Mantenimiento de Cursos": Pulsando la opción "Mantenimiento de Cursos", el usuario accederá a la sección "Mantenimiento de Cursos" que se describirá en el apartado 9.6.
- "Mantenimiento de Asignaturas": Pulsando la opción "Mantenimiento de Asignaturas", el usuario accederá a la sección "Mantenimiento de Asignaturas" que se describirá en el apartado 9.7.
- "Mantenimiento de Tarifas": Pulsando la opción "Mantenimiento de Tarifas", el usuario accederá a la sección "Mantenimiento de Tarifas" que se describirá en el apartado 9.8.

9.3 Profesorado

Para acceder a la sección "Profesorado", el usuario deberá pulsar la opción "Profesores" del menú. Desde esta sección el usuario va a poder:

- Hacer una búsqueda de profesores
- Dar de alta un nuevo profesor
- Ver los datos personales de un profesor y editarlos
- Consultar el horario de un profesor
- Eliminar un profesor

9.3.1 Búsqueda de Profesores

Para hacer una búsqueda de profesores, el usuario deberá acceder a la sección "Profesorado". El usuario podrá ver el siguiente formulario de búsqueda:

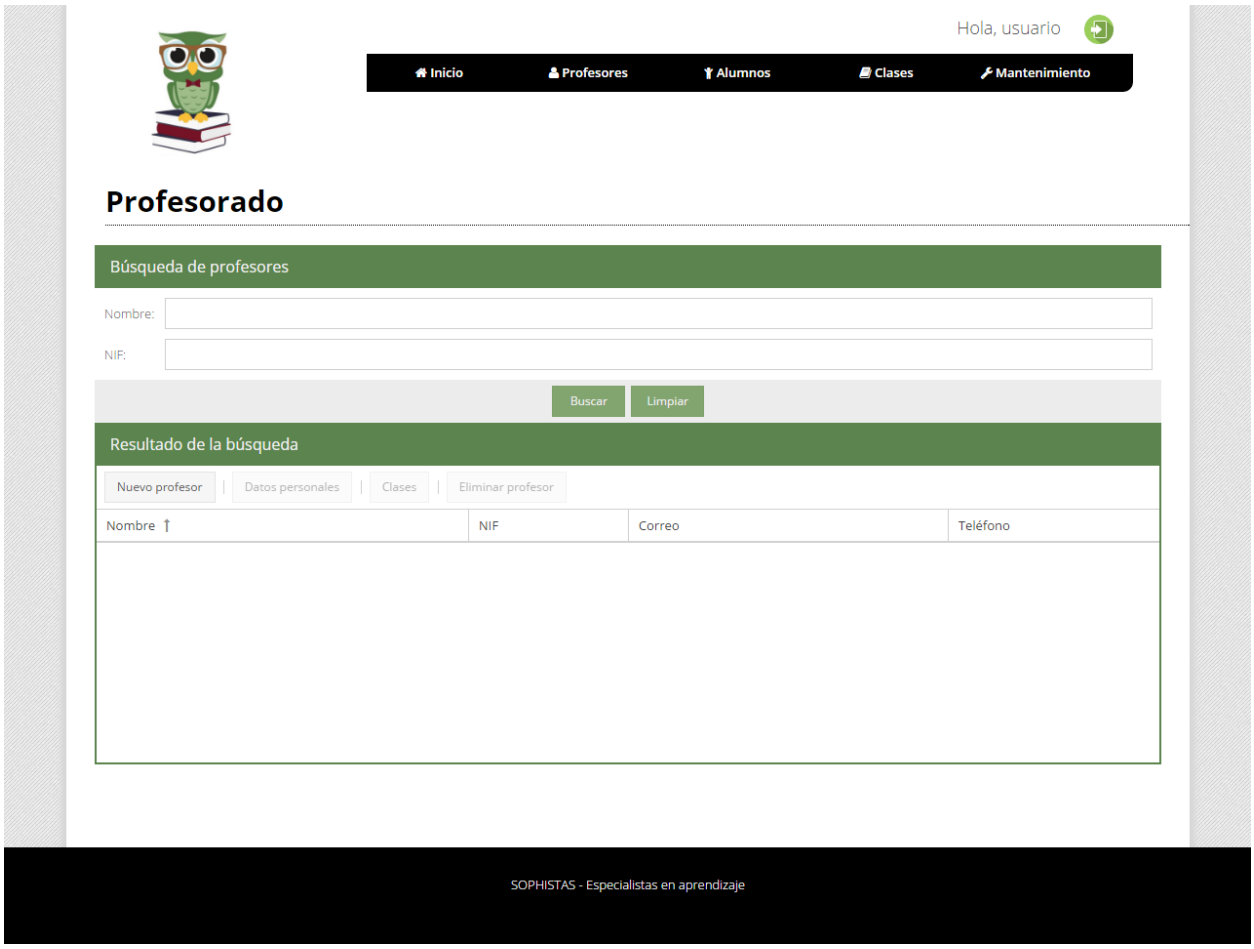


Figura 9-5. Búsqueda de profesores

El usuario podrá buscar por:

- Nombre
- NIF

El usuario deberá introducir la información por la que desea buscar y pulsar el botón “Buscar”. El buscador mostrará un listado con todos aquellos profesores que cumplieron las especificaciones de búsqueda.

Notas:

- En la búsqueda no se hace distinción de mayúsculas y minúsculas.
- Si no se rellena ningún campo del buscador se mostrarán todos los profesores.

9.3.2 Alta de Profesores

Para dar de alta un nuevo profesor, el usuario deberá acceder a la sección “Profesorado” y una vez allí pulsar el botón “Nuevo profesor”. El usuario podrá ver el siguiente formulario de alta que deberá rellenar:

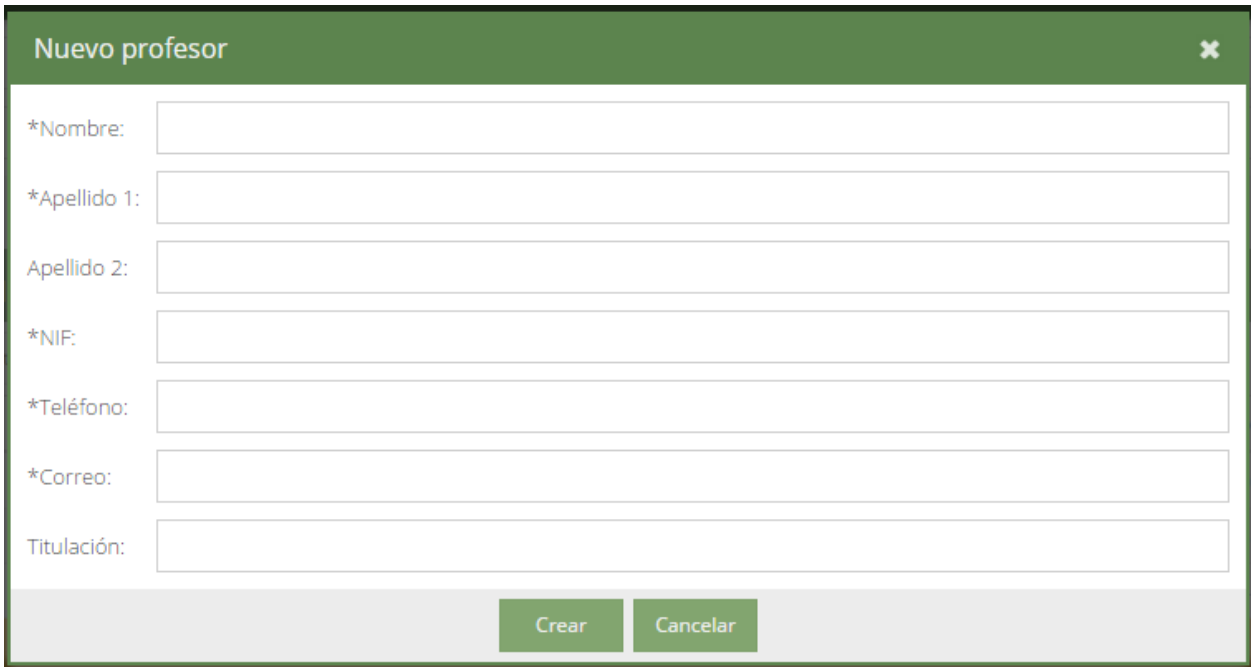


Figura 9-6. Formulario de Nuevo profesor

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- El NIF debe tener formato correcto
- El teléfono debe ser un número de 9 cifras
- El correo debe tener formato correcto
- No puede haber otro profesor con el mismo NIF

Una vez rellenado los campos, el usuario pulsará el botón "Crear".

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

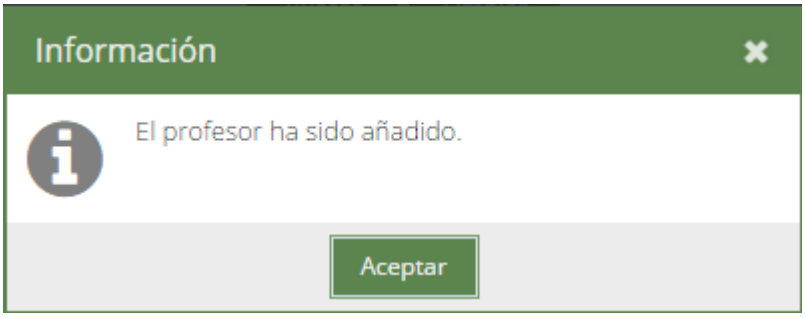


Figura 9-7. El profesor ha sido añadido

9.3.3 Detalle y edición de los datos personales de los Profesores

Para ver los datos personales de un profesor, el usuario deberá acceder a la sección "Profesorado", buscarlo como se explica en el apartado 9.3.1, seleccionarlo y pulsar el botón "Datos personales".

El usuario podrá ver el siguiente formulario con los datos personales del profesor, los cuales podrá modificar:

Figura 9-8. Formulario de Datos personales profesor

Al igual que cuando se da de alta un nuevo profesor, los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- El NIF debe tener formato correcto
- El teléfono debe ser un número de 9 cifras
- El correo debe tener formato correcto
- No puede haber otro profesor con el mismo NIF

Una vez rellenado los campos, el usuario pulsará el botón "Modificar". Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

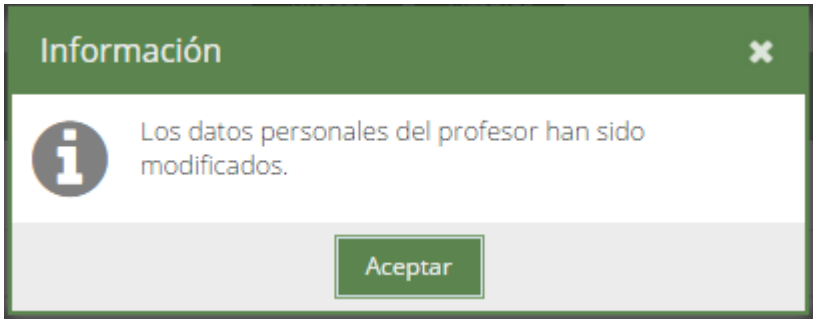


Figura 9-9. Los datos personales del profesor han sido modificados

Si el usuario sólo quiere ver los datos personales del profesor sin modificarlos, deberá cerrar la ventana emergente del formulario para seguir usando la aplicación.

9.3.4 Consulta del horario de los Profesores

Para consultar el horario de un profesor, el usuario deberá acceder a la sección "Profesorado", buscarlo como se explica en el apartado 9.3.1, seleccionarlo y pulsar el botón "Clases".

El usuario podrá ver una ventana como ésta con el horario del profesor:

Figura 9-10. Horario del profesor

9.3.5 Eliminación de Profesores

Para eliminar un profesor, el usuario deberá acceder a la sección "Profesorado", buscarlo como se explica en el apartado 9.3.1, seleccionarlo y pulsar el botón "Eliminar profesor".

Antes de ser eliminado, el usuario deberá confirmar:

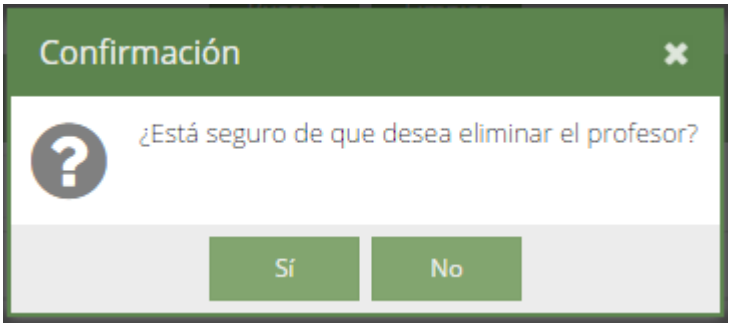


Figura 9-11. Confirmación para eliminar profesor

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

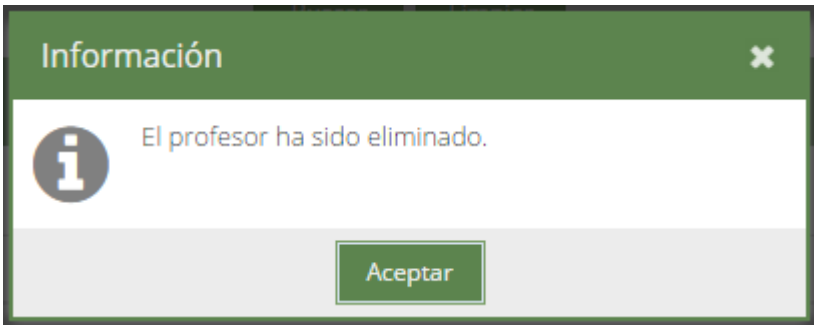


Figura 9-12. El profesor ha sido eliminado

9.4 Alumnado

Para acceder a la sección "Alumnado", el usuario deberá pulsar la opción "Alumnos" del menú. Desde esta sección el usuario va a poder:

- Hacer una búsqueda de alumnos
- Dar de alta un nuevo alumno (y responsable en su caso)
- Ver los datos personales de un alumno y editarlos (y responsable en su caso)
- Asignar clases a un alumno y consultar su horario
- Dar de baja un alumno
- Dar de alta un antiguo alumno

9.4.1 Búsqueda de Alumnos

Para hacer una búsqueda de alumnos, el usuario deberá acceder a la sección "Alumnado". El usuario podrá ver el siguiente formulario de búsqueda:

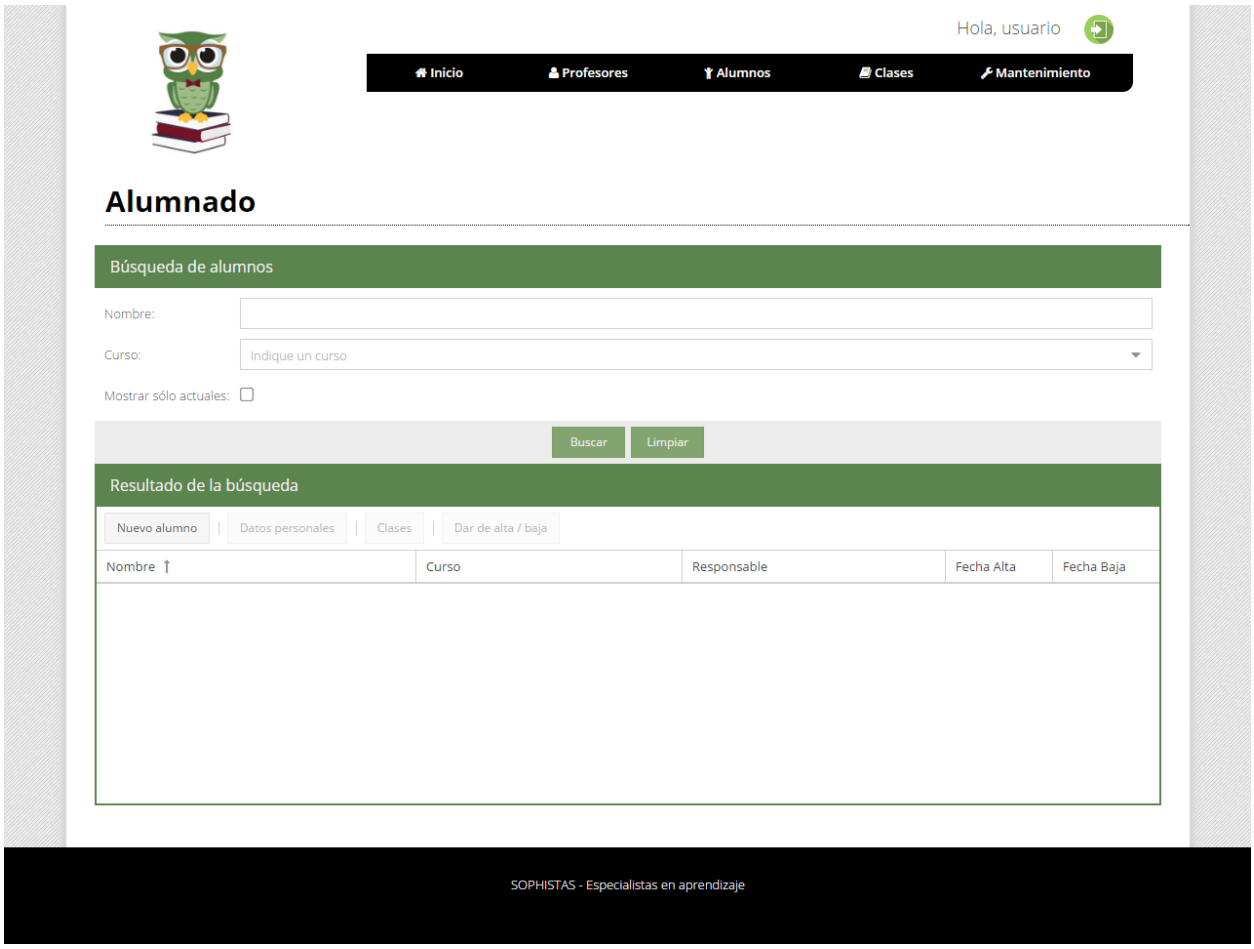


Figura 9-13. Búsqueda de alumnos

El usuario podrá buscar por:

- Nombre
- Curso
- Mostrar sólo actuales

El usuario deberá introducir la información por la que desea buscar y pulsar el botón “Buscar”. El buscador mostrará un listado con todos aquellos alumnos que cumplieron las especificaciones de búsqueda.

Notas:

- En la búsqueda no se hace distinción de mayúsculas y minúsculas.
- Si no se rellena ningún campo del buscador se mostrarán todos los alumnos.

9.4.2 Alta de nuevos Alumnos (y Responsable en su caso)

Para dar de alta un nuevo alumno (y su responsable en caso de que sea necesario), el usuario deberá acceder a la sección “Alumnado” y una vez allí pulsar el botón “Nuevo alumno”. El usuario podrá ver el siguiente formulario de alta que deberá rellenar:

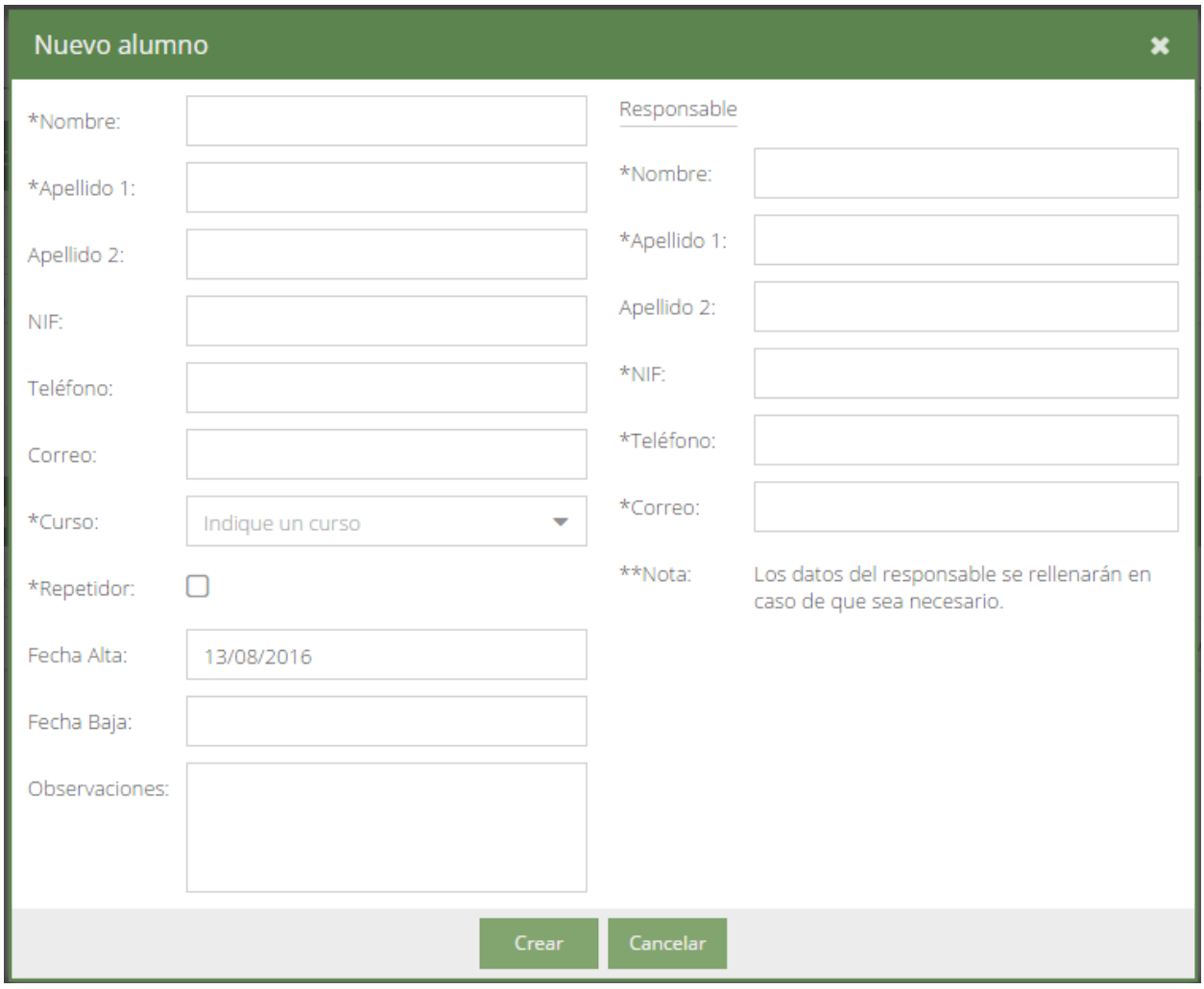


Figura 9-14. Formulario de Nuevo alumno

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- Los NIF deben tener formato correcto
- Los teléfonos deben ser números de 9 cifras
- Los correos deben tener formato correcto
- No puede haber otro alumno con el mismo NIF
- Los campos del Responsable se validarán en el caso en que el usuario rellene alguno de ellos, puesto que con ello se considera que se debe almacenar un responsable para el alumno.

La fecha de alta será la fecha actual y no se podrá modificar. La fecha de baja estará vacía y no será editable.

Una vez rellenado los campos, el usuario pulsará el botón "Crear".

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

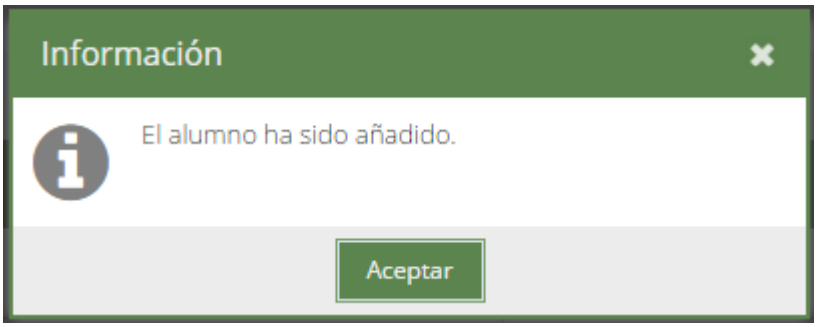


Figura 9-15. El alumno ha sido añadido

9.4.3 Detalle y edición de los datos personales de los Alumnos (y Responsable en su caso)

Para ver los datos personales de un alumno, y de su responsable en el caso de que lo tuviera, el usuario deberá acceder a la sección "Alumnado", buscarlo como se explica en el apartado 9.4.1, seleccionarlo y pulsar el botón "Datos personales".

El usuario podrá ver el siguiente formulario con los datos personales del alumno, los cuales podrá modificar:

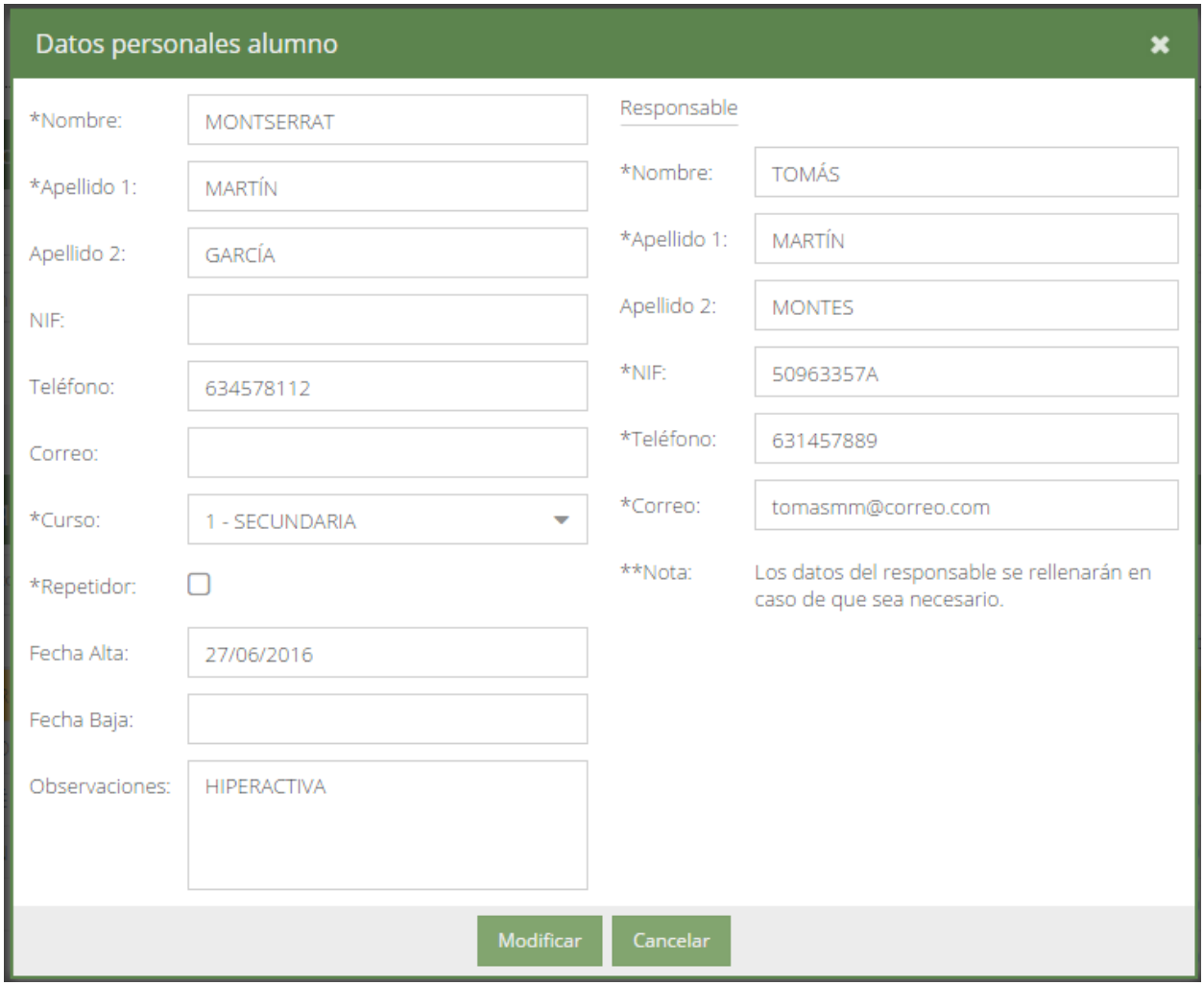


Figura 9-16. Formulario de Datos personales alumno

Al igual que cuando se da de alta un nuevo alumno, los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- Los NIF deben tener formato correcto
- Los teléfonos deben ser números de 9 cifras
- Los correos deben tener formato correcto
- No puede haber otro alumno con el mismo NIF
- Los campos del Responsable se validarán en el caso en que el usuario rellene alguno de ellos, puesto que con ello se considera que se debe almacenar un responsable para el alumno.

Los campos para las fechas de alta y baja no son editables.

Una vez rellenado los campos, el usuario pulsará el botón "Modificar". Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

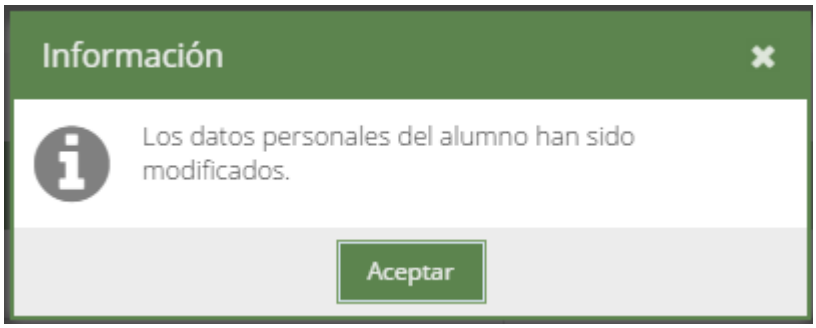


Figura 9-17. Los datos personales del alumno han sido modificados

Si el usuario sólo quiere ver los datos personales del alumno sin modificarlos, deberá cerrar la ventana emergente del formulario para seguir usando la aplicación.

9.4.4 Asignación de clases y consulta del horario de los Alumnos

Para asignar las clases o consultar el horario de un alumno, el usuario deberá acceder a la sección "Alumnado", buscarlo como se explica en el apartado 9.4.1, seleccionarlo y pulsar el botón "Clases".

El usuario podrá ver una ventana como ésta:

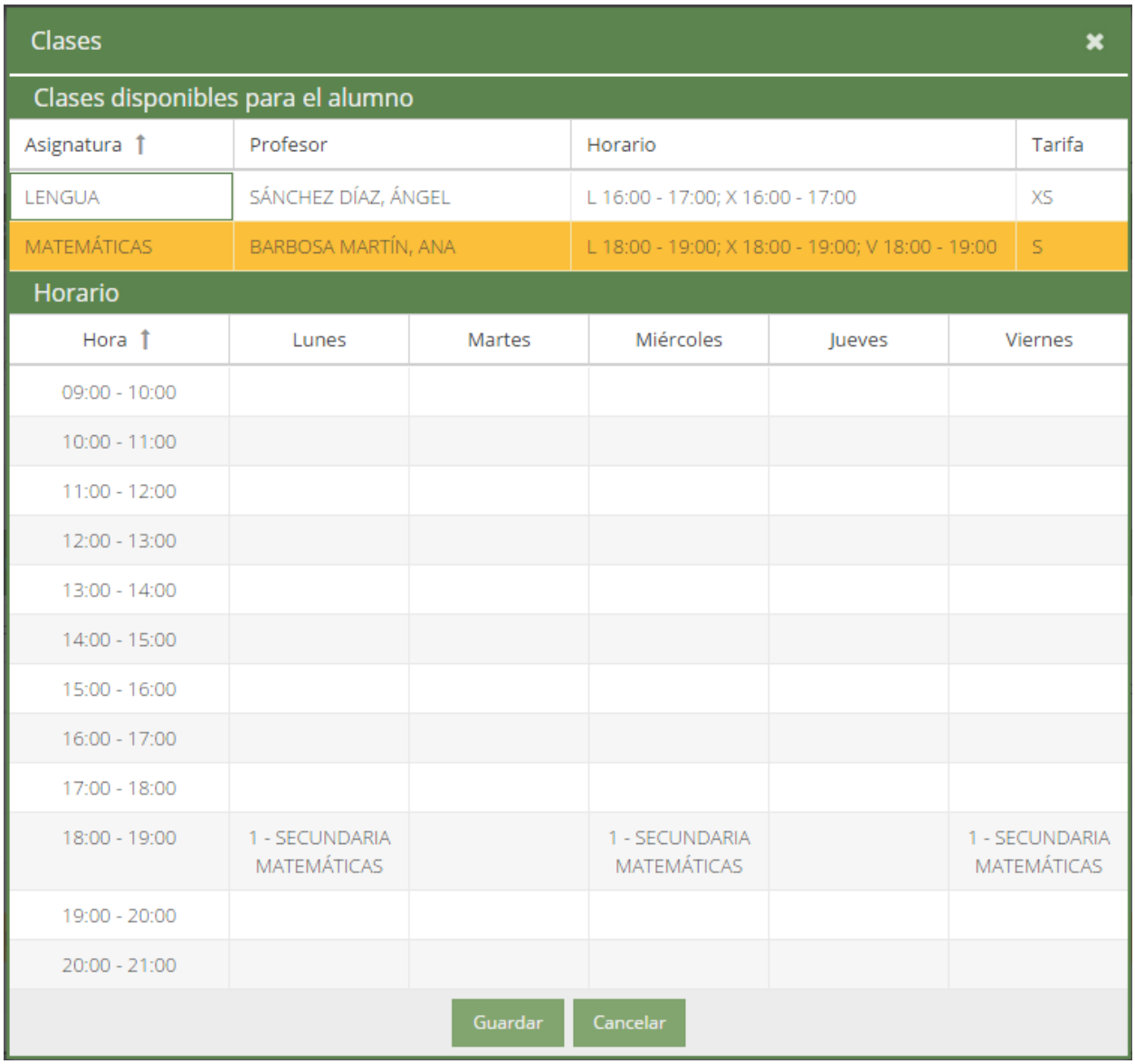


Figura 9-18. Asignación de clases y horario del alumno

En la parte parte superior aparecen las distintas clases disponibles para el alumno, esto es, las clases de su curso. Estarán marcadas aquellas en las que el alumno está matriculado. Si se quiere asignar al alumno una nueva clase o, por el contrario, borrar al alumno de una clase, se seleccionará aquí la clase en cuestión. En la parte inferior se puede ir viendo el horario del alumno.

Una vez seleccionadas las clases el usuario deberá pulsar el botón “Guardar”. Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

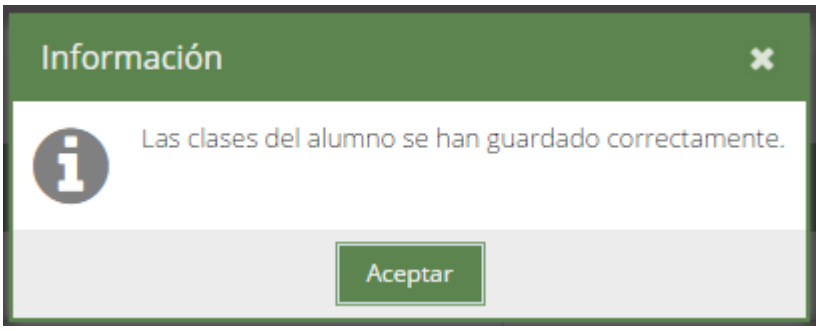


Figura 9-19. Las clases del alumno se han guardado correctamente

Si el usuario sólo quiere ver el horario del alumno sin modificar las clases, deberá cerrar la ventana emergente del formulario para seguir usando la aplicación.

9.4.5 Baja de alumnos

Para dar de baja un alumno, el usuario deberá acceder a la sección "Alumnado", buscarlo como se explica en el apartado 9.4.1, seleccionarlo y pulsar el botón "Dar de alta / baja".

Antes de ser dado de baja, el usuario deberá confirmar:

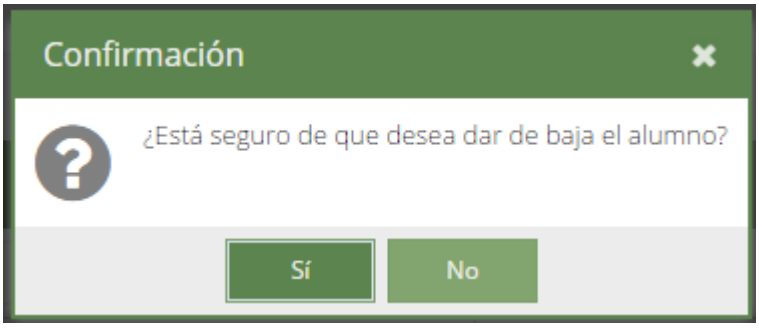


Figura 9-20. Confirmación para dar de baja alumno

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

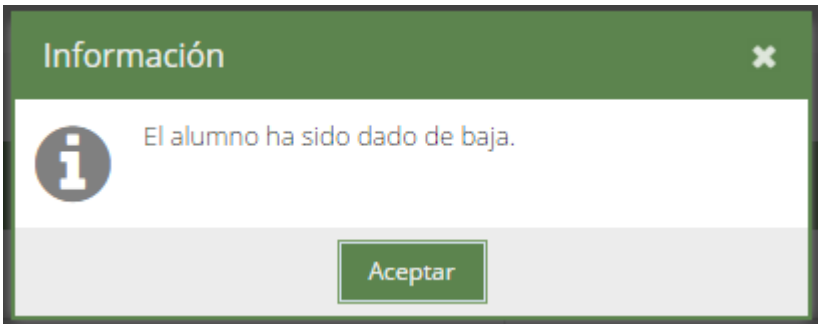


Figura 9-21. El alumno ha sido dado de baja

9.4.6 Alta de antiguos Alumnos dados de baja

Para dar de alta un antiguo alumno que había sido dado de baja previamente, el usuario deberá acceder a la sección "Alumnado", buscarlo como se explica en el apartado 9.4.1, seleccionarlo y pulsar el botón "Dar de alta / baja".

Antes de ser dado de alta, el usuario deberá confirmar:

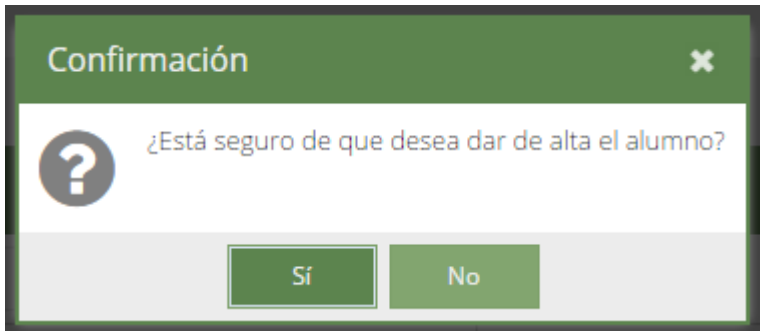


Figura 9-22. Confirmación para dar de alta un antiguo alumno

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

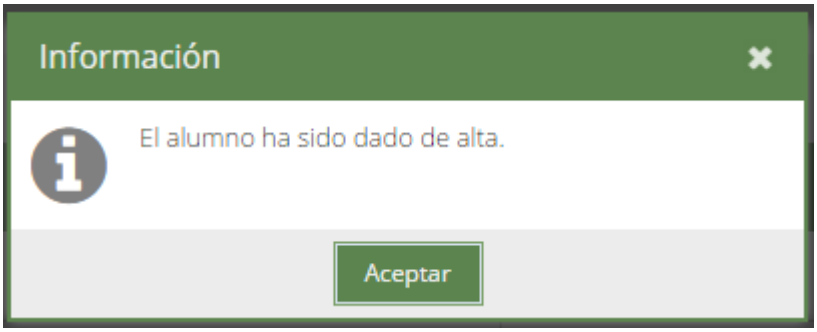


Figura 9-23. El alumno ha sido dado de alta

9.5 Clases

Para acceder a la sección "Clases", el usuario deberá pulsar la opción "Clases" del menú. Desde esta sección el usuario va a poder:

- Hacer una búsqueda de clases
- Dar de alta una nueva clase
- Ver el detalle de una clase y editarla
- Ver la lista de alumnos de una clase
- Eliminar una clase

9.5.1 Búsqueda de Clases

Para hacer una búsqueda de clases, el usuario deberá acceder a la sección "Clases". El usuario podrá ver el siguiente formulario de búsqueda:

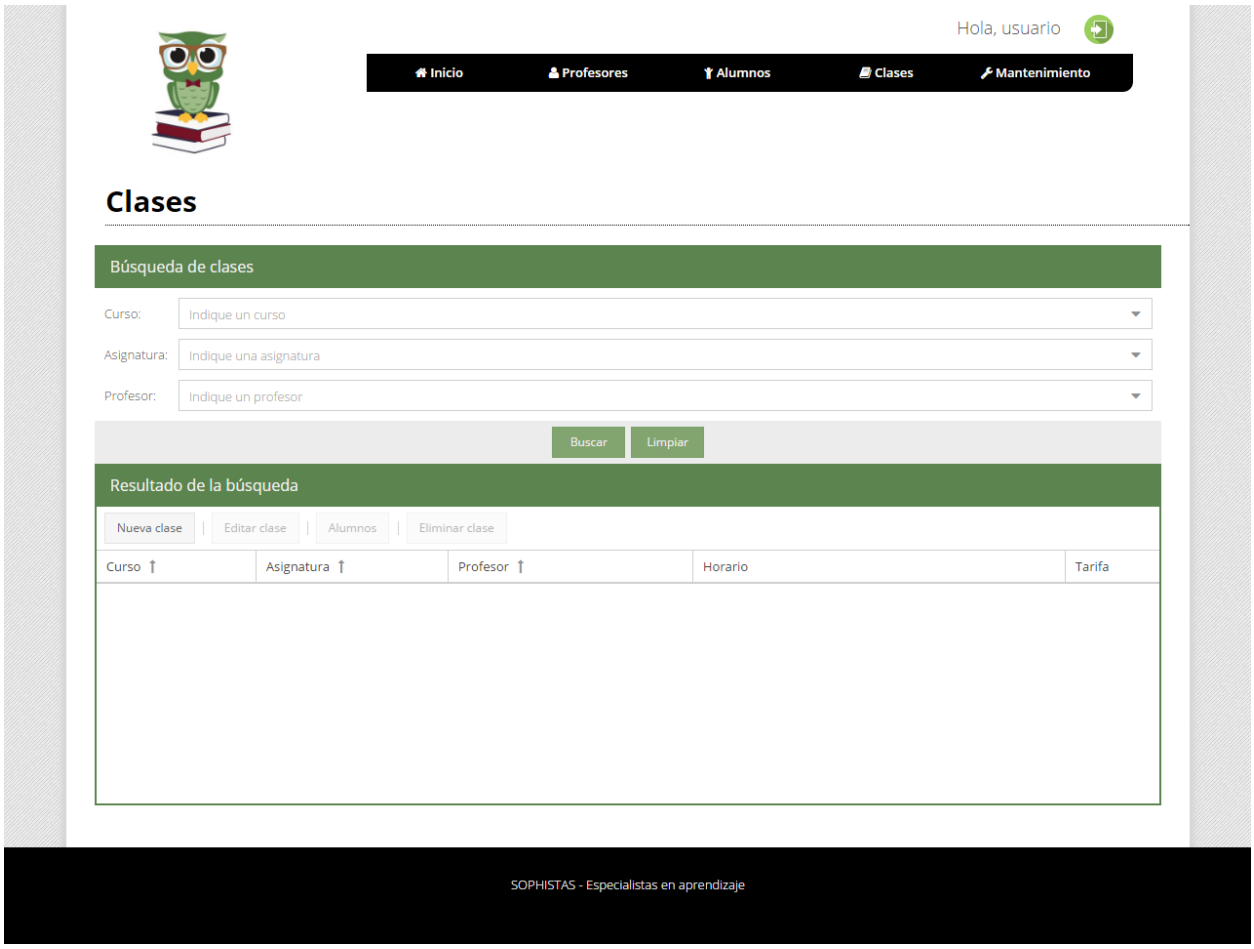


Figura 9-24. Búsqueda de clases

El usuario podrá buscar por:

- Curso
- Asignatura
- Profesor

El usuario deberá introducir la información por la que desea buscar y pulsar el botón "Buscar". El buscador mostrará un listado con todas aquellas clases que cumplieron las especificaciones de búsqueda.

Nota:

- Si no se rellena ningún campo del buscador se mostrarán todas las clases.

9.5.2 Alta de Clases

Para dar de alta una nueva clase, el usuario deberá acceder a la sección "Clases" y una vez allí pulsar el botón "Nueva clase". El usuario podrá ver el siguiente formulario de alta que deberá rellenar:

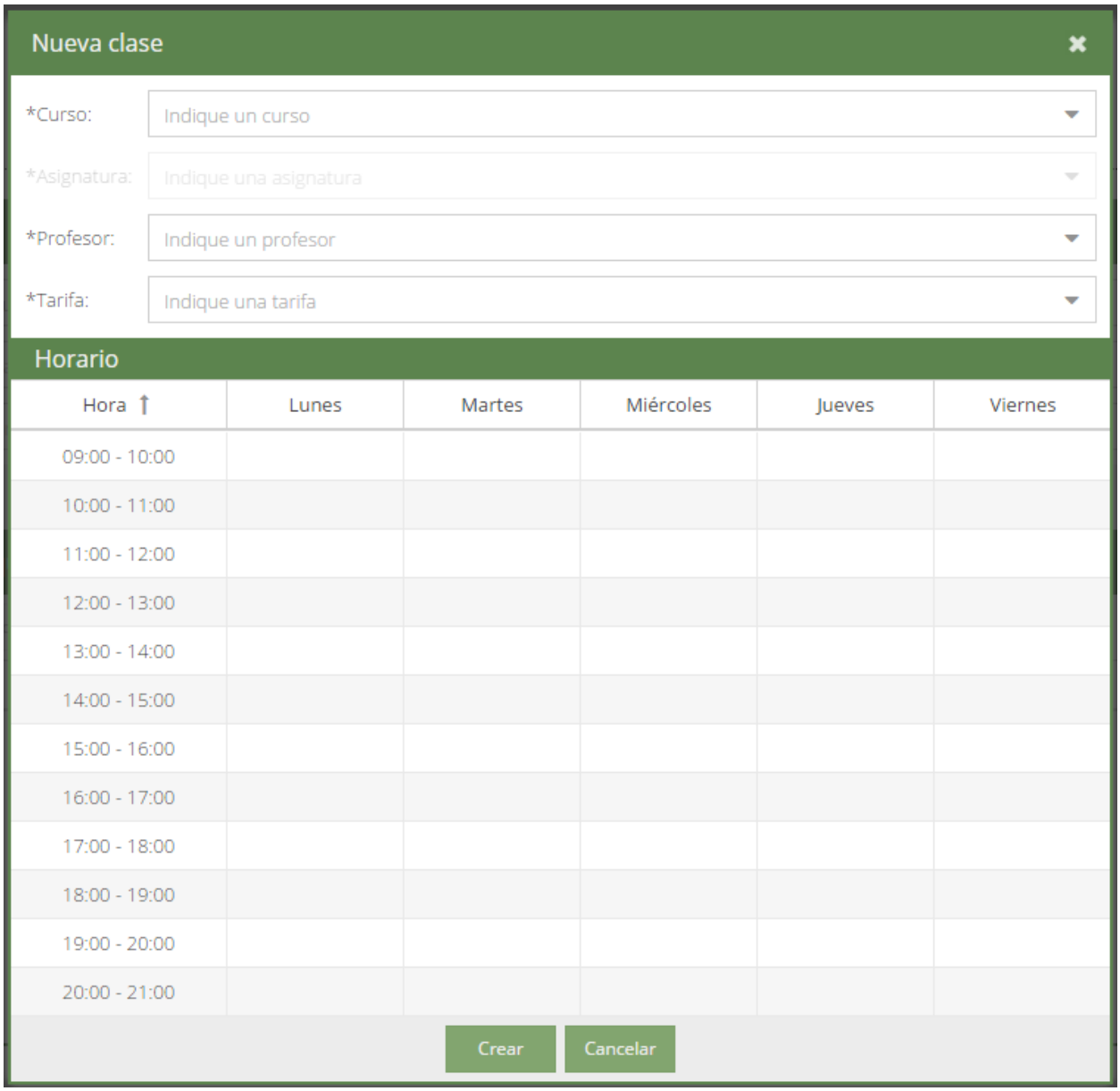


Figura 9-25. Formulario de Nueva clase

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- El horario seleccionado debe ser compatible con el horario del profesor

Una vez rellenado los campos, el usuario pulsará el botón "Crear".

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

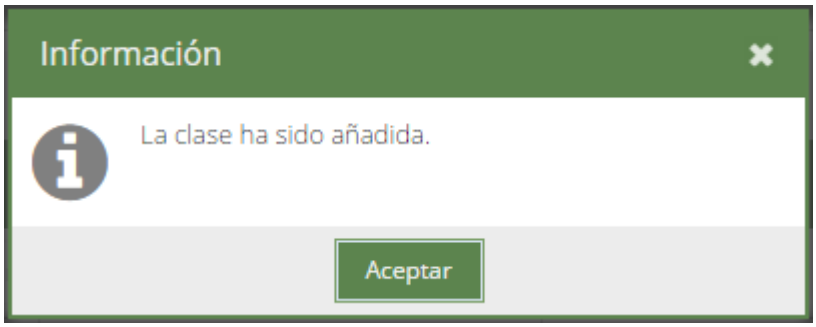


Figura 9-26. La clase ha sido añadida

9.5.3 Detalle y edición de las Clases

Para ver el detalle de una clase, el usuario deberá acceder a la sección "Clases", buscarla como se explica en el apartado 9.5.1, seleccionarla y pulsar el botón "Editar clase".

El usuario podrá ver el siguiente formulario con el detalle de la clase, la cual podrá modificar:

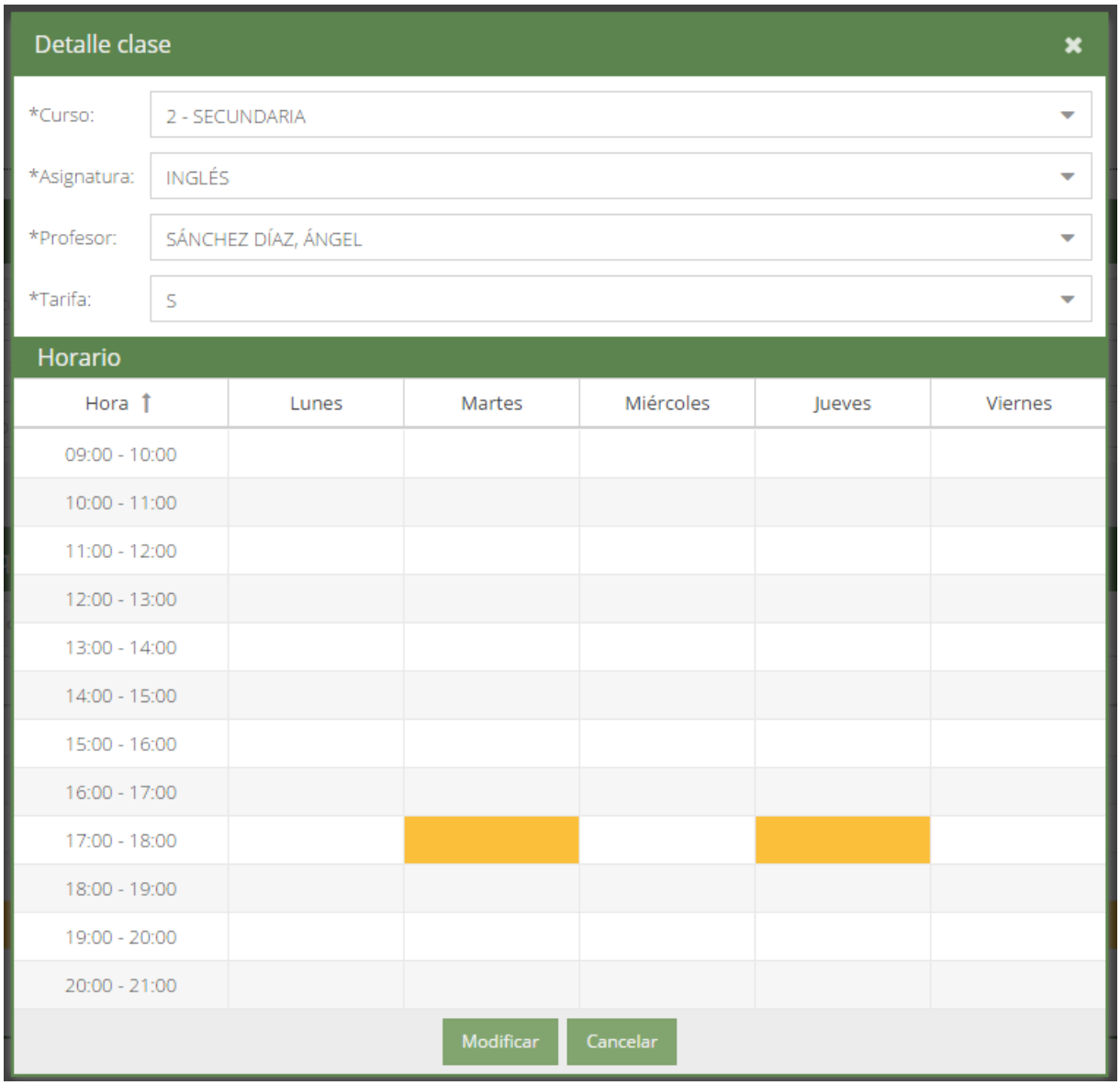


Figura 9-27. Formulario de Detalle clase

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- El horario seleccionado debe ser compatible con el horario del profesor
- El horario seleccionado debe ser compatible con el horario de los alumnos matriculados en esa clase

Los campos para el curso y la asignatura no son editables.

Una vez rellenado los campos, el usuario pulsará el botón "Modificar". Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

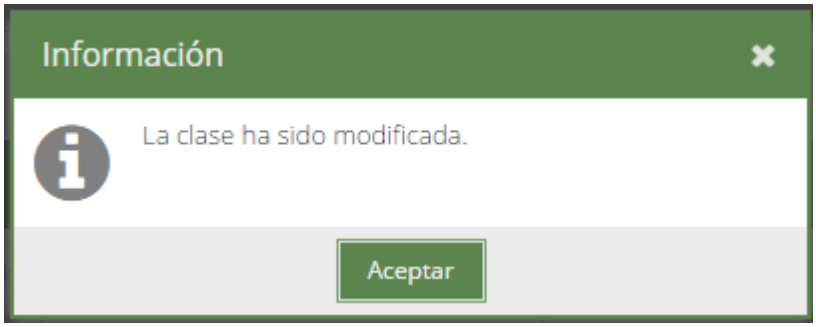


Figura 9-28. La clase ha sido modificada

Si el usuario sólo quiere ver el detalle de la clase sin modificarla, deberá cerrar la ventana emergente del formulario para seguir usando la aplicación.

9.5.4 Ver lista de alumnos

Para ver el listado de alumnos matriculados en una clase, el usuario deberá acceder a la sección "Clases", buscarla como se explica en el apartado 9.5.1, seleccionarla y pulsar el botón "Alumnos".

El usuario podrá ver una ventana como ésta con la lista de alumnos:

Figura 9-29. Alumnos de la clase

9.5.5 Eliminación de Clases

Para eliminar una clase, el usuario deberá acceder a la sección "Clases", buscarla como se explica en el apartado 9.5.1, seleccionarla y pulsar el botón "Eliminar clase".

Antes de ser eliminada, el usuario deberá confirmar:

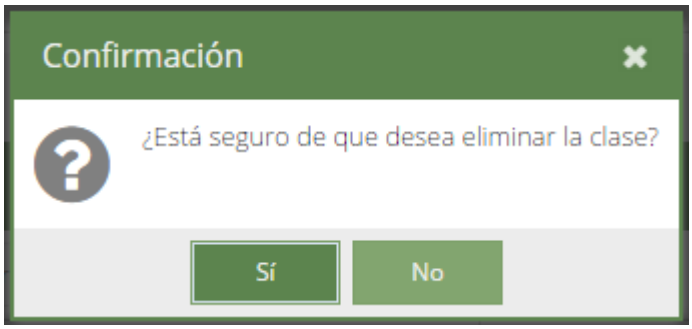


Figura 9-30. Confirmación para eliminar clase

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

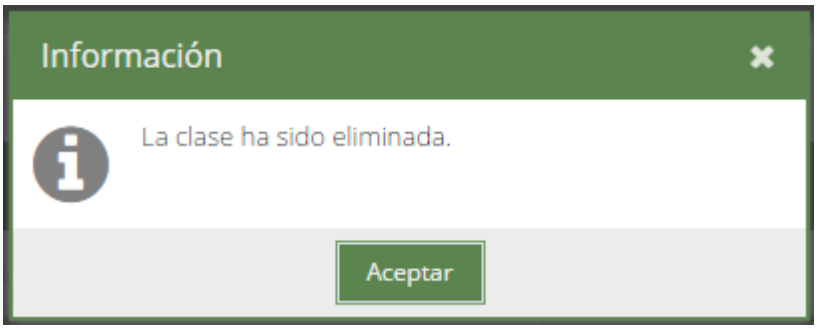


Figura 9-31. La clase ha sido eliminada

9.6 Mantenimiento de Cursos

Para acceder a la sección "Mantenimiento de Cursos", el usuario deberá pulsar la opción "Mantenimiento de Cursos" del menú. Desde esta sección el usuario va a poder:

- Ver un listado de los cursos
- Dar de alta un nuevo curso
- Ver el detalle de un curso y editarlo
- Eliminar un curso

9.6.1 Listado de Cursos

Para ver el listado de los cursos, el usuario deberá acceder a la sección "Mantenimiento de Cursos". El usuario podrá ver una pantalla como ésta:

Figura 9-32. Listado de cursos

9.6.2 Alta de Cursos

Para dar de alta un nuevo curso, el usuario deberá acceder a la sección "Mantenimiento de Cursos" y una vez allí pulsar el botón "Nuevo curso". El usuario podrá ver el siguiente formulario de alta que deberá rellenar:

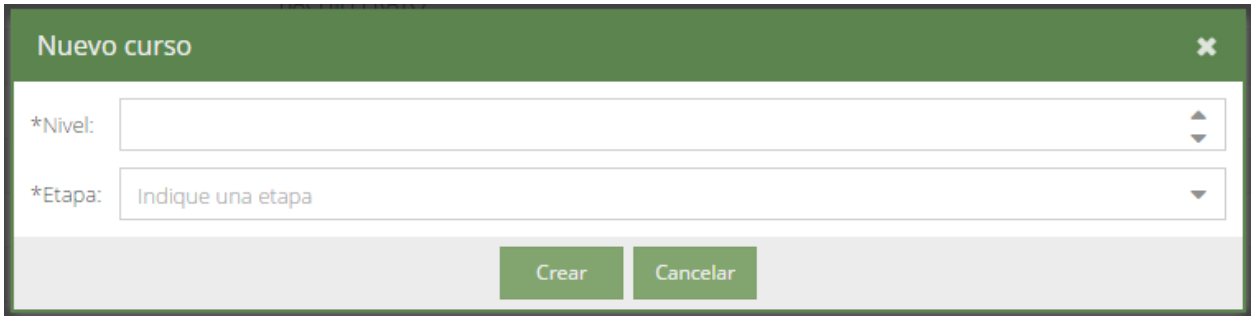


Figura 9-33. Formulario de Nuevo curso

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- No puede haber otro curso con el nivel y la etapa introducidos

Una vez rellenado los campos, el usuario pulsará el botón "Crear".

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

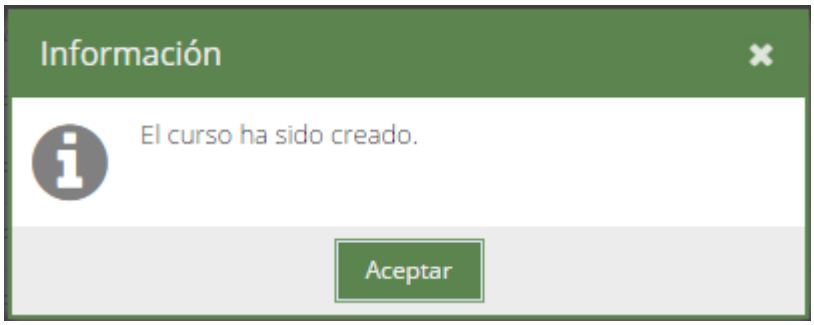


Figura 9-34. El curso ha sido creado

9.6.3 Detalle y edición de los Cursos

Para ver el detalle de un curso, el usuario deberá acceder a la sección "Mantenimiento de Cursos", seleccionarlo y pulsar el botón "Editar curso".

El usuario podrá ver el siguiente formulario con el detalle del curso, el cual podrá modificar:

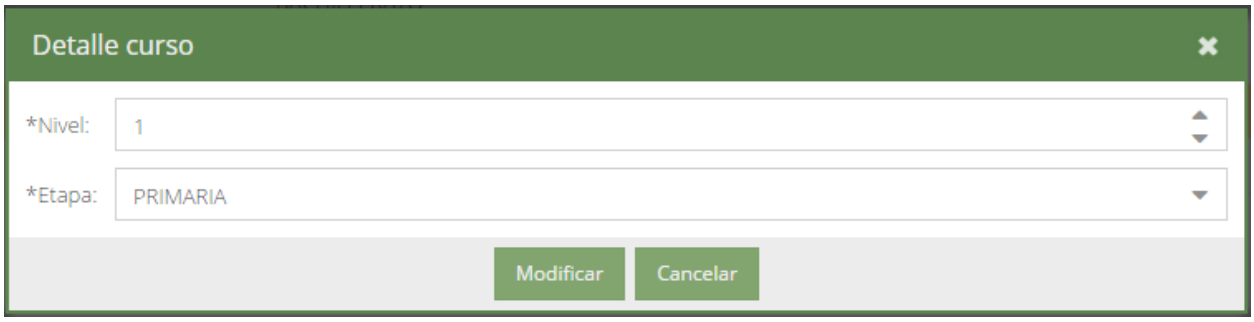


Figura 9-35. Formulario de Detalle curso

Al igual que cuando se da de alta un nuevo curso, los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- No puede haber otro curso con el nivel y la etapa introducidos

Una vez rellenado los campos, el usuario pulsará el botón "Modificar". Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

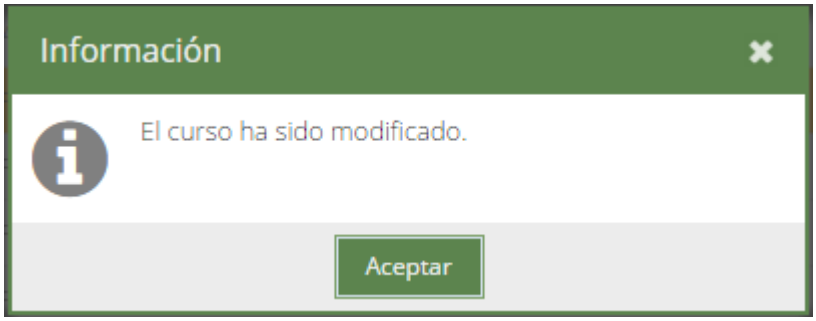


Figura 9-36. El curso ha sido modificado

Si el usuario sólo quiere ver el detalle del curso sin modificarlo, deberá cerrar la ventana emergente del formulario para seguir usando la aplicación.

9.6.4 Eliminación de Cursos

Para eliminar un curso, el usuario deberá acceder a la sección "Mantenimiento de Cursos", seleccionarlo y pulsar el botón "Eliminar curso".

Antes de ser eliminado, el usuario deberá confirmar:

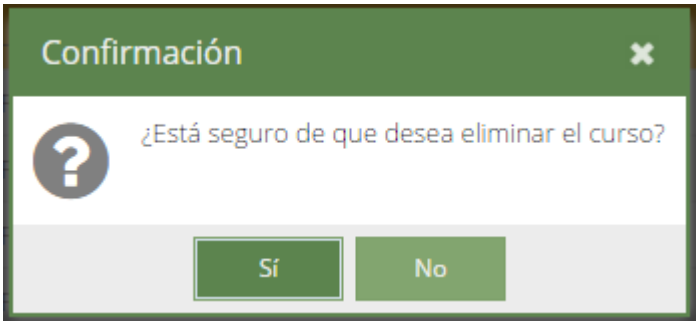


Figura 9-37. Confirmación para eliminar curso

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

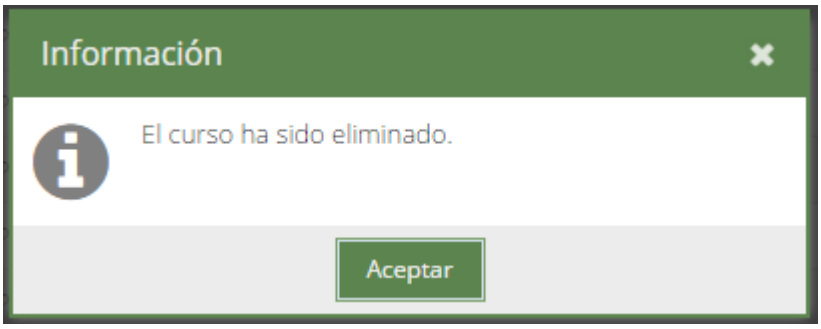


Figura 9-38. El curso ha sido eliminado

9.7 Mantenimiento de Asignaturas

Para acceder a la sección "Mantenimiento de Asignaturas", el usuario deberá pulsar la opción "Mantenimiento de Asignaturas" del menú. Desde esta sección el usuario va a poder:

- Ver un listado de las asignaturas
- Dar de alta una nueva asignatura
- Ver el detalle de una asignatura y editarla
- Eliminar una asignatura

9.7.1 Listado de Asignaturas

Para ver el listado de las asignaturas, el usuario deberá acceder a la sección "Mantenimiento de Asignaturas". El usuario podrá ver una pantalla como ésta:

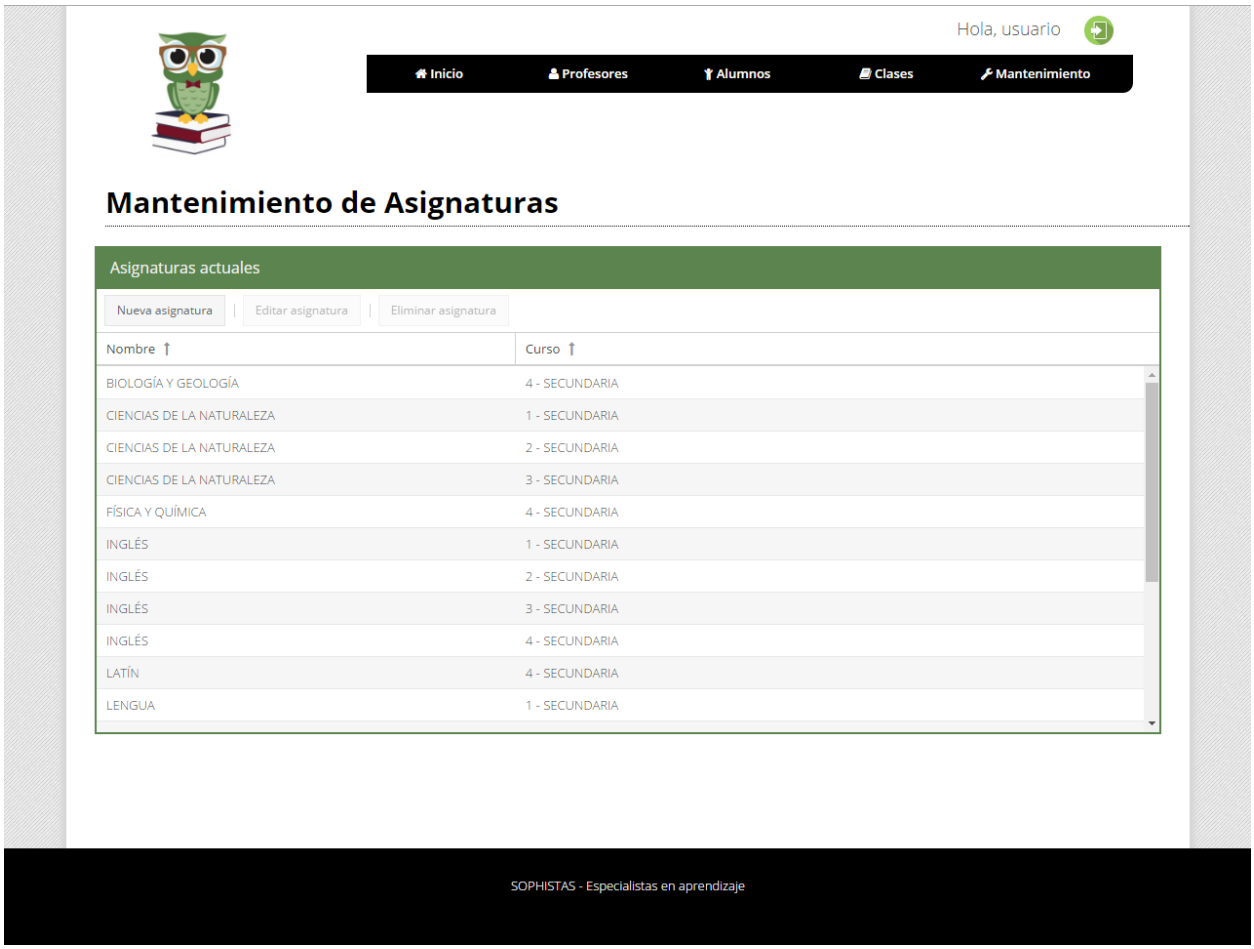


Figura 9-39. Listado de asignaturas

9.7.2 Alta de Asignaturas

Para dar de alta una nueva asignatura, el usuario deberá acceder a la sección "Mantenimiento de Asignaturas" y una vez allí pulsar el botón "Nueva asignatura". El usuario podrá ver el siguiente formulario de alta que deberá rellenar:

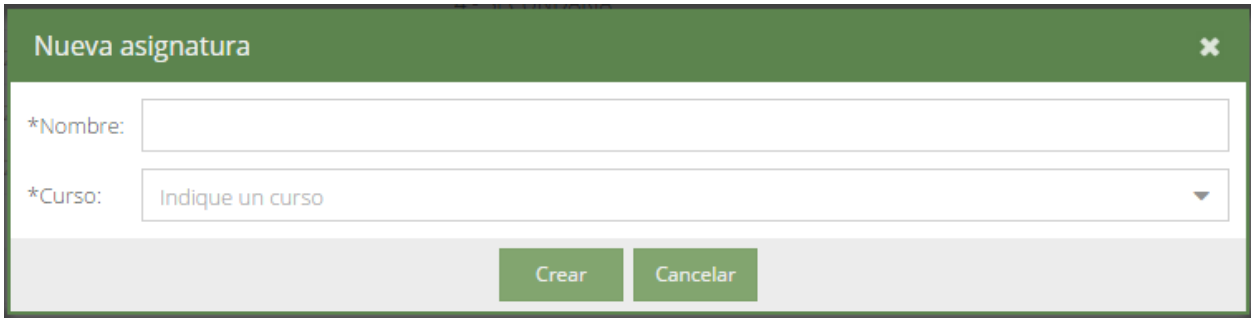


Figura 9-40. Formulario de Nueva asignatura

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- No puede haber otra asignatura con el nombre y el curso introducidos

Una vez rellenado los campos, el usuario pulsará el botón "Crear".

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

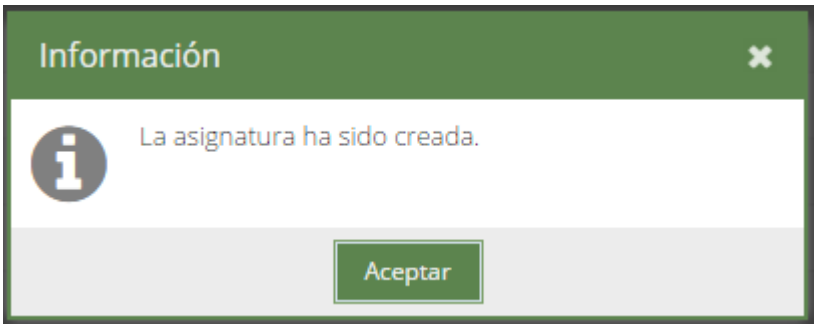


Figura 9-41. La asignatura ha sido creada

9.7.3 Detalle y edición de las Asignaturas

Para ver el detalle de una asignatura, el usuario deberá acceder a la sección "Mantenimiento de Asignaturas", seleccionarla y pulsar el botón "Editar asignatura".

El usuario podrá ver el siguiente formulario con el detalle de la asignatura, la cual podrá modificar:

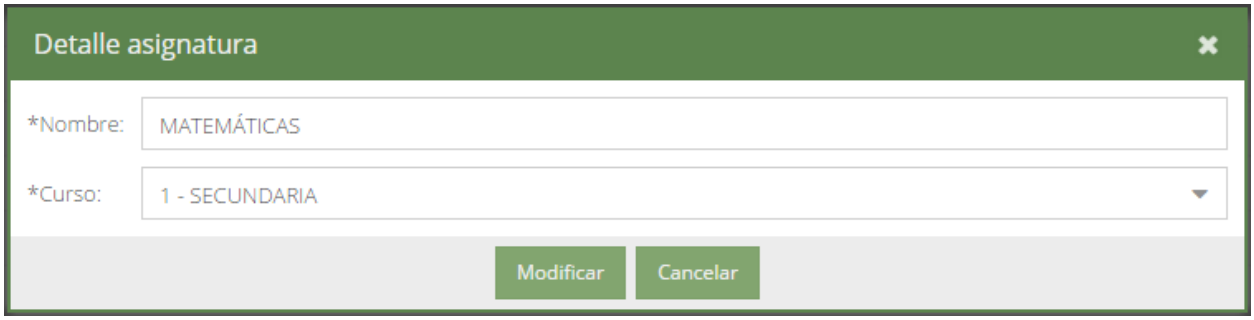


Figura 9-42. Formulario de Detalle asignatura

Al igual que cuando se da de alta una nueva asignatura, los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- No puede haber otra asignatura con el nombre y el curso introducidos

Una vez rellenado los campos, el usuario pulsará el botón "Modificar". Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

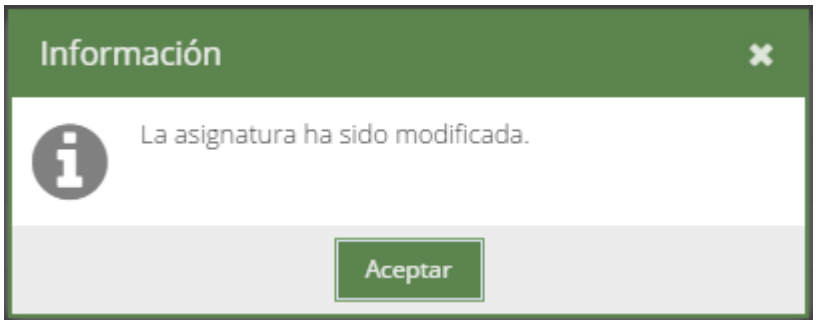


Figura 9-43. La asignatura ha sido modificada

Si el usuario sólo quiere ver el detalle de la asignatura sin modificarla, deberá cerrar la ventana emergente del

formulario para seguir usando la aplicación.

9.7.4 Eliminación de Asignaturas

Para eliminar una asignatura, el usuario deberá acceder a la sección "Mantenimiento de Asignaturas", seleccionarla y pulsar el botón "Eliminar asignatura".

Antes de ser eliminada, el usuario deberá confirmar:

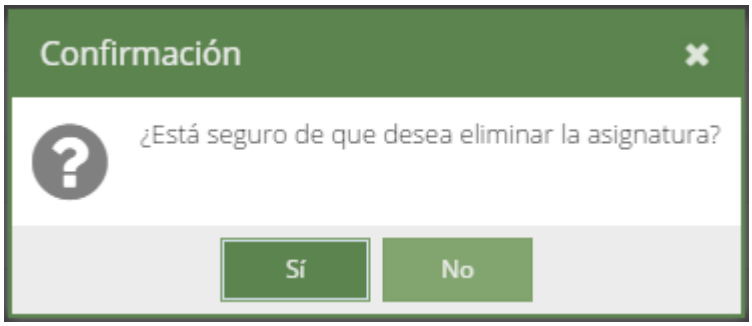


Figura 9-44. Confirmación para eliminar asignatura

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

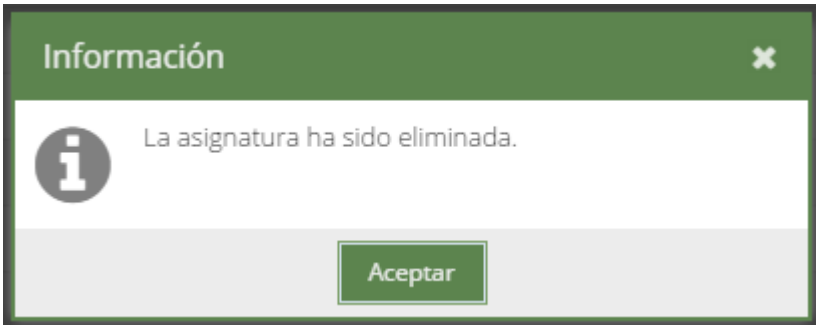


Figura 9-45. La asignatura ha sido eliminada

9.8 Mantenimiento de Tarifas

Para acceder a la sección "Mantenimiento de Tarifas", el usuario deberá pulsar la opción "Mantenimiento de Tarifas" del menú. Desde esta sección el usuario va a poder:

- Ver un listado de las tarifas
- Dar de alta una nueva tarifa
- Ver el detalle de una tarifa y editarla
- Eliminar una tarifa

9.8.1 Listado de Tarifas

Para ver el listado de las tarifas, el usuario deberá acceder a la sección "Mantenimiento de Tarifas". El usuario podrá ver una pantalla como ésta:

Figura 9-46. Listado de tarifas

9.8.2 Alta de Tarifas

Para dar de alta una nueva tarifa, el usuario deberá acceder a la sección "Mantenimiento de Tarifas" y una vez allí pulsar el botón "Nueva tarifa". El usuario podrá ver el siguiente formulario de alta que deberá rellenar:

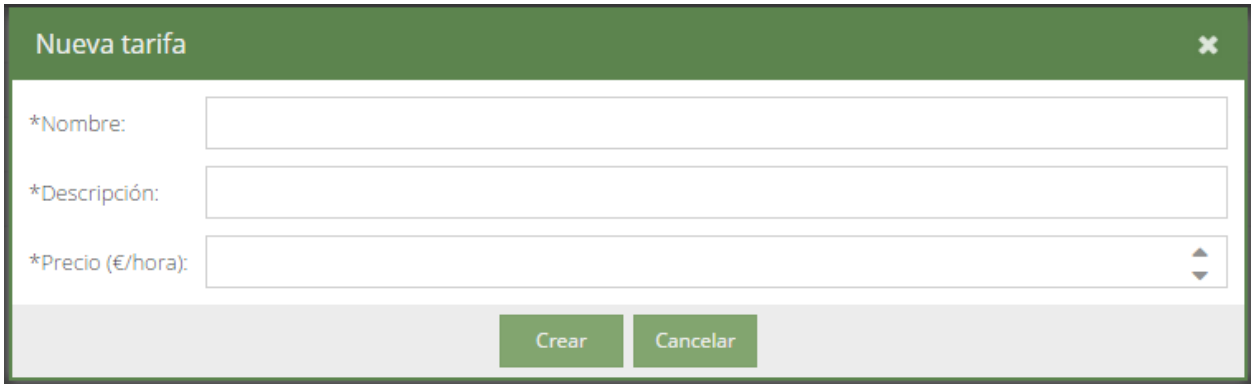


Figura 9-47. Formulario de Nueva tarifa

Los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un '*' son obligatorios
- No puede haber otra tarifa con el nombre introducido

Una vez rellenado los campos, el usuario pulsará el botón “Crear”.

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

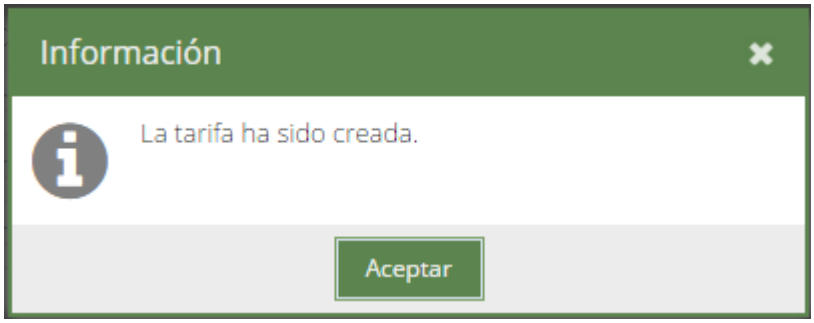


Figura 9-48. La tarifa ha sido creada

9.8.3 Detalle y edición de las Tarifas

Para ver el detalle de una tarifa, el usuario deberá acceder a la sección “Mantenimiento de Tarifas”, seleccionarla y pulsar el botón “Editar tarifa”.

El usuario podrá ver el siguiente formulario con el detalle de la tarifa, la cual podrá modificar:

Figura 9-49. Formulario de Detalle tarifa

Al igual que cuando se da de alta una nueva tarifa, los datos introducidos deben cumplir los siguientes puntos:

- Los campos marcados con un ‘*’ son obligatorios
- No puede haber otra tarifa con el nombre introducido

Una vez rellenado los campos, el usuario pulsará el botón “Modificar”. Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

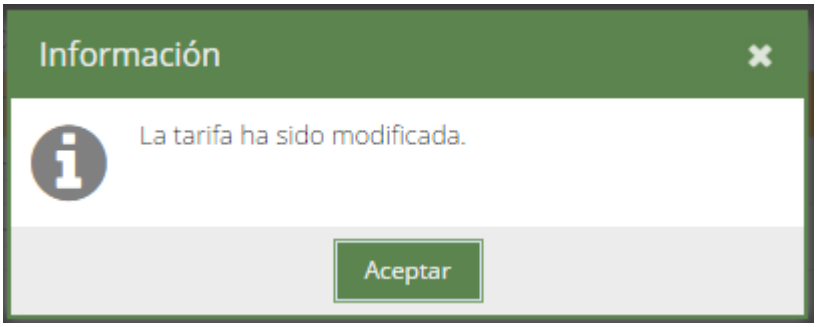


Figura 9-50. La tarifa ha sido modificada

Si el usuario sólo quiere ver el detalle de la tarifa sin modificarla, deberá cerrar la ventana emergente del formulario para seguir usando la aplicación.

9.8.4 Eliminación de Tarifas

Para eliminar una tarifa, el usuario deberá acceder a la sección "Mantenimiento de Tarifas", seleccionarla y pulsar el botón "Eliminar tarifa".

Antes de ser eliminada, el usuario deberá confirmar:

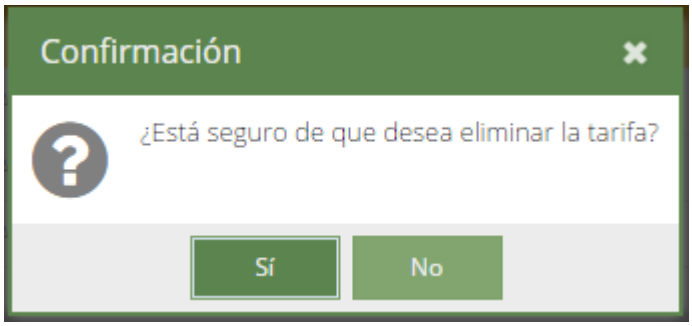


Figura 9-51. Confirmación para eliminar tarifa

Si todo ha ido bien, se le mostrará el siguiente mensaje al usuario:

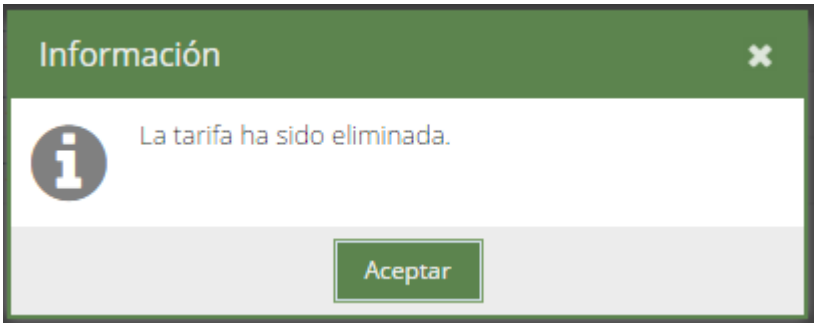


Figura 9-52. La tarifa ha sido eliminada

9.9 Validaciones de los formularios

Cada vez que se rellena un formularios y se pulsa el botón correspondiente ("Crear", "Modificar", …), la aplicación valida que los datos introducidos por el usuario sean correctos, es decir, sigan el formato correcto, sean coherentes, … Si el usuario introduce algún campo de manera errónea la aplicación le mostrará mensajes como estos:

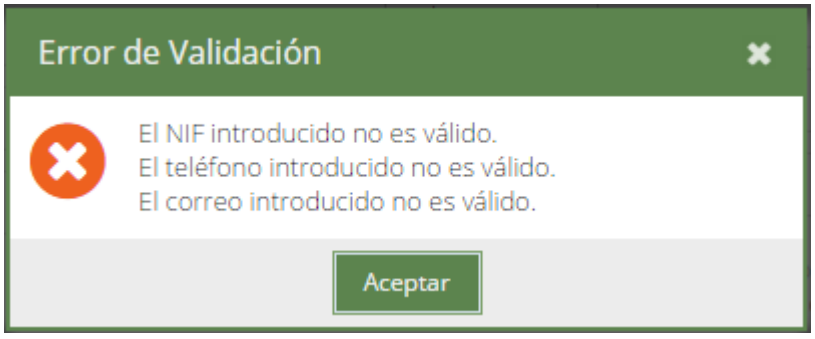


Figura 9-53. Error de Validación

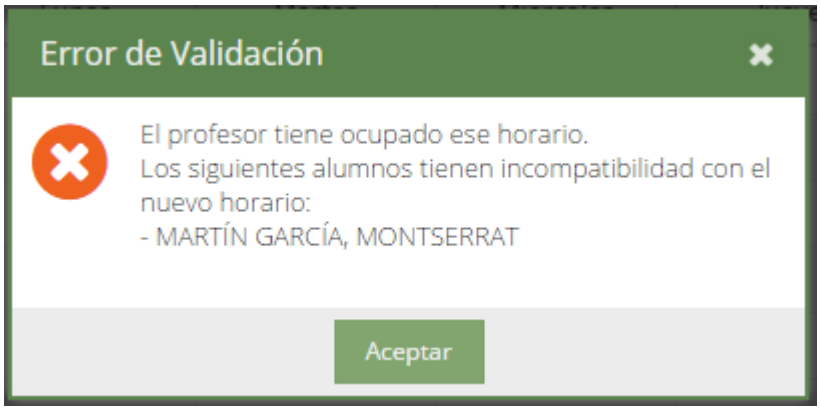


Figura 9-54. Error de Validación 2

9.10 Salir de la aplicación

Para salir de la aplicación el usuario deberá pulsar el botón situado en la esquina superior derecha:

Figura 9-55. Botón Salir

Se cerrará la sesión del usuario y a éste le aparecerá el siguiente mensaje:

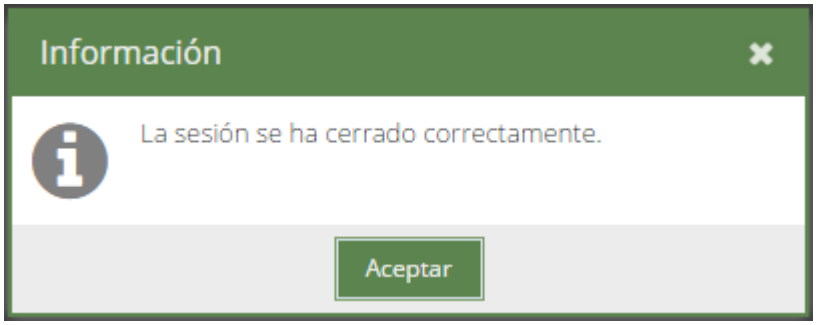


Figura 9-56. Fin de sesión

10 PLAN DE PRUEBAS INTEGRADAS

No hay nadie menos afortunado que el hombre a quien la adversidad olvida, pues no tiene oportunidad de ponerse a prueba.

Lucio Anneo Séneca

Una vez que la aplicación web se ha desarrollado, es necesario hacerla pasar por una serie de pruebas antes de entrar a la fase de producción (comienzo del uso de la aplicación). Mediante dichas pruebas, se medirá su reacción integral frente a diversas acciones que realizarán los usuarios desde sus páginas.

Las pruebas deben obedecer en su totalidad a un plan generado ANTES de empezar con la programación y DESPUÉS de haber realizado el análisis de requisitos de la aplicación, ya que este plan de pruebas debe hacerse en función de cada uno de los requisitos identificados en el análisis.

Las pruebas deben ser:

- Completas: deben cubrir la mayor cantidad de código y las posibilidades de ejecución que la aplicación puede ejecutar.
- Repetibles y reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra.
- Profesionales: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

El plan de pruebas integradas, en caso de realizarse las pruebas en su totalidad con éxito, es la forma de certificar que la aplicación abarca el conjunto de funcionalidades que el análisis de requisistos identificó.

10.1 Gestión de acceso

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Login con credenciales incorrectas	1. Acceder a la aplicación 2. Introducir unas credenciales incorrectas 3. Botón “Entrar”	Debe aparecer el mensaje “Credenciales inválidas, por favor intenta nuevamente.”	OK	
2	Login con credenciales correctas	1. Acceder a la aplicación 2. Introducir unas credenciales correctas 3. Botón “Entrar”	Debe dirigirse a la pantalla de inicio de la aplicación	OK	
3	Logout	1. Acceder a la aplicación 2. Introducir unas credenciales correctas 3. Botón “Entrar” 4. Botón “Salir”	Debe aparecer el mensaje “La sesión se ha cerrado correctamente.” y dirigirse a la pantalla de login	OK	
4	Comprobar expiración de sesión	1. Acceder a la aplicación 2. Introducir unas credenciales correctas 3. Botón “Entrar” 4. Estar más de 30 minutos sin hacer nada 5. Entrar en alguna sección de la aplicación	Debe redirigirse a la pantalla de login ya que la sesión debe haber expirado	OK	

10.2 Estadísticas en página de inicio

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Estadísticas “Etapas”	1. Acceder a la aplicación 2. Loguearse en la aplicación	En el diagrama “Etapas” de la página de inicio deben aparecer los porcentajes correctos de alumnos en las diferentes etapas	OK	
2	Estadísticas “Altas y Bajas”	1. Acceder a la aplicación 2. Loguearse en la aplicación	En el diagrama “Altas y Bajas” de la página de inicio deben aparecer los números correctos de altas y bajas de alumnos en los últimos 4 meses	OK	

10.3 Profesores

10.3.1 Búsqueda de profesores

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Búsqueda sin parámetros	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Buscar”	Debe aparecer un listado con todos los profesores almacenados en la aplicación	OK	
2	Búsqueda por nombre	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Rellenar el campo “Nombre” 5. Botón “Buscar”	Debe aparecer un listado de los profesores cuyo nombre contenga la cadena introducida	OK	
3	Búsqueda por NIF	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Rellenar el campo “NIF” 5. Botón “Buscar”	Debe aparecer el profesor cuyo NIF sea el introducido	OK	

10.3.2 Nuevo profesor

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nuevo profesor con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar todos los campos del formulario correctamente 6. Botón “Crear”	Debe almacenarse el nuevo profesor en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“El profesor ha sido añadido.”)	OK	

2	Nuevo profesor sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario sin introducir el campo “Nombre” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK
3	Nuevo profesor sin rellenar campo “Apellido 1”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario sin introducir el campo “Apellido 1” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1'.”	OK
4	Nuevo profesor sin rellenar campo “NIF”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario sin introducir el campo “NIF” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'NIF'.”	OK
5	Nuevo profesor sin rellenar campo “Teléfono”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario sin introducir el campo “Teléfono” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Teléfono'.”	OK
6	Nuevo profesor sin rellenar campo “Correo”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario sin introducir el campo “Correo” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Correo'.”	OK

7	Nuevo profesor con campo “NIF” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario con valor inválido en el campo “NIF” 6. Botón “Crear”	Debe aparecer el mensaje “El NIF introducido no es válido.”	OK
8	Nuevo profesor con campo “Teléfono” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario con valor inválido en el campo “Teléfono” 6. Botón “Crear”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK
9	Nuevo profesor con campo “Correo” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario con valor inválido en el campo “Correo” 6. Botón “Crear”	Debe aparecer el mensaje “El correo introducido no es válido.”	OK
10	Nuevo profesor con NIF repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Botón “Nuevo profesor” 5. Rellenar el formulario con valor repetido en el campo “NIF” 6. Botón “Crear”	Debe aparecer el mensaje “Ya hay almacenado un profesor con ese NIF.”	OK

10.3.3 Datos personales

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver datos personales de un profesor	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales”	Deben mostrarse los datos personales del profesor	OK	
2	Modificar datos personales de un profesor con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar todos los campos del formulario correctamente con los nuevos datos 8. Botón “Modificar”	Deben almacenarse los nuevos datos del profesor en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“Los datos personales del profesor han sido modificados.”)	OK	
3	Modificar datos personales de un profesor sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Nombre” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK	

4	Modificar datos personales de un profesor sin rellenar campo “Apellido 1”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Apellido 1” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1'.”	OK
5	Modificar datos personales de un profesor sin rellenar campo “NIF”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “NIF” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'NIF'.”	OK
6	Modificar datos personales de un profesor sin rellenar campo “Teléfono”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Teléfono” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Teléfono'.”	OK
7	Modificar datos personales de un profesor sin rellenar campo “Correo”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Correo” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Correo'.”	OK

8	Modificar datos personales de un profesor con campo “NIF” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “NIF” 8. Botón “Modificar”	Debe aparecer el mensaje “El NIF introducido no es válido.”	OK
9	Modificar datos personales de un profesor con campo “Teléfono” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Teléfono” 8. Botón “Modificar”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK
10	Modificar datos personales de un profesor con campo “Correo” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Correo” 8. Botón “Modificar”	Debe aparecer el mensaje “El correo introducido no es válido.”	OK
11	Modificar datos personales de un profesor con NIF repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Profesorado” 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón “Datos personales” 7. Rellenar el formulario con valor repetido en el campo “NIF” 8. Botón “Modificar”	Debe aparecer el mensaje “Ya hay almacenado un profesor con ese NIF.”	OK

10.3.4 Clases

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver clases de un profesor	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Profesorado" 4. Buscar el profesor 5. Seleccionar el profesor 6. Botón "Clases"	Debe mostrarse el horario con las clases del profesor	OK	

10.3.5 Eliminar profesor

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Eliminar un profesor sin clases asignadas	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Profesorado" 4. Buscar el profesor 5. Seleccionar el profesor (sin clases asignadas) 6. Botón "Eliminar profesor" 7. Confirmar que se desea eliminar el profesor	Debe eliminarse el profesor de la BBDD y al usuario le debe aparecer un mensaje indicándoselo ("El profesor ha sido eliminado.")	OK	
2	Eliminar un profesor con clases asignadas	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Profesorado" 4. Buscar el profesor 5. Seleccionar el profesor (con clases asignadas) 6. Botón "Eliminar profesor" 7. Confirmar que se desea eliminar el profesor	Debe aparecer el mensaje "No se ha podido eliminar el profesor ya que existen clases almacenadas impartidas por él."	OK	

10.4 Alumnos

10.4.1 Búsqueda de alumnos

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Búsqueda sin parámetros	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Buscar”	Debe aparecer un listado con todos los alumnos almacenados en la aplicación	OK	
2	Búsqueda por nombre	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Rellenar el campo “Nombre” 5. Botón “Buscar”	Debe aparecer un listado de los alumnos cuyo nombre contenga la cadena introducida	OK	
3	Búsqueda por curso	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Rellenar el campo “Curso” 5. Botón “Buscar”	Debe aparecer un listado de los alumnos del curso indicado	OK	
4	Buscar sólo actuales	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Marcar “Mostrar sólo actuales” 5. Botón “Buscar”	Debe aparecer un listado de los alumnos que no están dados de baja	OK	

10.4.2 Nuevo alumno

10.4.2.1 Nuevo alumno sin responsable

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nuevo alumno con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar todos los campos del formulario correctamente 6. Botón “Crear”	Debe almacenarse el nuevo alumno en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“El alumno ha sido añadido.”)	OK	
2	Nuevo alumno sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Nombre” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK	
3	Nuevo alumno sin rellenar campo “Apellido 1”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Apellido 1” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1'.”	OK	
4	Nuevo alumno sin indicar el campo “Curso”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Curso” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Curso'.”	OK	

5	Nuevo alumno sin rellenar campo “Teléfono”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Teléfono” 6. Botón “Crear”	Debe aparecer el mensaje “Debe indicar algún teléfono (ya sea del alumno o del responsable).”	OK	
6	Nuevo alumno sin rellenar campo “Correo”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Correo” 6. Botón “Crear”	Debe aparecer el mensaje “Debe indicar algún correo (ya sea del alumno o del responsable).”	OK	
7	Nuevo alumno con campo “NIF” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “NIF” 6. Botón “Crear”	Debe aparecer el mensaje “El NIF introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “NIF”
8	Nuevo alumno con campo “Teléfono” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “Teléfono” 6. Botón “Crear”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK	
9	Nuevo alumno con campo “Correo” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “Correo” 6. Botón “Crear”	Debe aparecer el mensaje “El correo introducido no es válido.”	OK	

10	Nuevo alumno con NIF repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor repetido en el campo “NIF” 6. Botón “Crear”	Debe aparecer el mensaje “Ya hay almacenado un alumno con ese NIF.”	OK	Esta comprobación se hace si se ha rellenado el campo “NIF”

10.4.2.2 Nuevo alumno con responsable

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nuevo alumno con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar todos los campos del formulario correctamente (incluida la información del responsable) 6. Botón “Crear”	Deben almacenarse el nuevo alumno y su responsable en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“El alumno ha sido añadido.”)	OK	
2	Nuevo alumno sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Nombre” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK	
3	Nuevo alumno sin rellenar campo “Apellido 1”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Apellido 1” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1'.”	OK	

4	Nuevo alumno sin indicar el campo “Curso”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Curso” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Curso'.”	OK	
5	Nuevo alumno con campo “NIF” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “NIF” 6. Botón “Crear”	Debe aparecer el mensaje “El NIF introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “NIF”
6	Nuevo alumno con campo “Teléfono” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “Teléfono” 6. Botón “Crear”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “Teléfono”
7	Nuevo alumno con campo “Correo” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “Correo” 6. Botón “Crear”	Debe aparecer el mensaje “El correo introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “Correo”
8	Nuevo alumno sin rellenar campo “Nombre” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Nombre” del Responsable 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste

9	Nuevo alumno sin rellenar campo “Apellido 1” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Apellido 1” del Responsable 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
10	Nuevo alumno sin rellenar campo “NIF” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “NIF” del Responsable 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'NIF' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
11	Nuevo alumno sin rellenar campo “Teléfono” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Teléfono” del Responsable 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Teléfono' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
12	Nuevo alumno sin rellenar campo “Correo” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario sin introducir el campo “Correo” del Responsable 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Correo' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
13	Nuevo alumno con campo “NIF” inválido del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Botón “Nuevo alumno” 5. Rellenar el formulario con valor inválido en el campo “NIF” del Responsable 6. Botón “Crear”	Debe aparecer el mensaje “El NIF del Responsable introducido no es válido.”	OK	La información del Responsable se valida si se introduce algún dato de éste

14	Nuevo alumno con campo "Teléfono" inválido del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Botón "Nuevo alumno" 5. Rellenar el formulario con valor inválido en el campo "Teléfono" del Responsable 6. Botón "Crear"	Debe aparecer el mensaje "El teléfono del Responsable introducido no es válido."	OK	La información del Responsable se valida si se introduce algún dato de éste
15	Nuevo alumno con campo "Correo" inválido del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Botón "Nuevo alumno" 5. Rellenar el formulario con valor inválido en el campo "Correo" del Responsable 6. Botón "Crear"	Debe aparecer el mensaje "El correo del Responsable introducido no es válido."	OK	La información del Responsable se valida si se introduce algún dato de éste
16	Nuevo alumno con NIF repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Botón "Nuevo alumno" 5. Rellenar el formulario con valor repetido en el campo "NIF" 6. Botón "Crear"	Debe aparecer el mensaje "Ya hay almacenado un alumno con ese NIF."	OK	Esta comprobación se hace si se ha rellenado el campo "NIF"

10.4.3 Datos personales

10.4.3.1 Alumno sin responsable

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver datos personales de un alumno	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón "Datos personales"	Deben mostrarse los datos personales del alumno	OK	

2	Modificar datos personales de un alumno con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar todos los campos del formulario correctamente con los nuevos datos 8. Botón “Modificar”	Deben almacenarse los nuevos datos del alumno en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“Los datos personales del alumno han sido modificados.”)	OK
3	Modificar datos personales de un alumno sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Nombre” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK
4	Modificar datos personales de un alumno sin rellenar campo “Apellido 1”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Apellido 1” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1'.”	OK
5	Modificar datos personales de un alumno sin rellenar campo “Curso”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Curso” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Curso'.”	OK

6	Modificar datos personales de un alumno sin rellenar campo “Teléfono”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Teléfono” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe indicar algún teléfono (ya sea del alumno o del responsable).”	OK	
7	Modificar datos personales de un alumno sin rellenar campo “Correo”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Correo” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe indicar algún correo (ya sea del alumno o del responsable).”	OK	
8	Modificar datos personales de un alumno con campo “NIF” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “NIF” 8. Botón “Modificar”	Debe aparecer el mensaje “El NIF introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “NIF”
9	Modificar datos personales de un alumno con campo “Teléfono” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Teléfono” 8. Botón “Modificar”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK	

10	Modificar datos personales de un alumno con campo "Correo" inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón "Datos personales" 7. Rellenar el formulario con valor inválido en el campo "Correo" 8. Botón "Modificar"	Debe aparecer el mensaje "El correo introducido no es válido."	OK	
11	Modificar datos personales de un alumno con NIF repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón "Datos personales" 7. Rellenar el formulario con valor repetido en el campo "NIF" 8. Botón "Modificar"	Debe aparecer el mensaje "Ya hay almacenado un alumno con ese NIF."	OK	Esta comprobación se hace si se ha rellenado el campo "NIF"

10.4.3.2 Alumno con responsable

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver datos personales de un alumno y su responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Alumnado" 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón "Datos personales"	Deben mostrarse los datos personales del alumno y de su responsable	OK	

2	Modificar datos personales de un alumno con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar todos los campos del formulario correctamente con los nuevos datos (incluida la información del responsable) 8. Botón “Modificar”	Deben almacenarse los nuevos datos del alumno y su responsable en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“Los datos personales del alumno han sido modificados.”)	OK
3	Modificar datos personales de un alumno sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Nombre” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK
4	Modificar datos personales de un alumno sin rellenar campo “Apellido 1”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Apellido 1” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1'.”	OK

5	Modificar datos personales de un alumno sin rellenar campo “Curso”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con los nuevos datos sin introducir el campo “Curso” 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Curso'.”	OK	
6	Modificar datos personales de un alumno con campo “NIF” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “NIF” 8. Botón “Modificar”	Debe aparecer el mensaje “El NIF introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “NIF”
7	Modificar datos personales de un alumno con campo “Teléfono” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Teléfono” 8. Botón “Modificar”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “Teléfono”
8	Modificar datos personales de un alumno con campo “Correo” inválido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Teléfono” 8. Botón “Modificar”	Debe aparecer el mensaje “El teléfono introducido no es válido.”	OK	Esta comprobación se hace si se ha rellenado el campo “Teléfono”

9	Modificar datos personales de un alumno sin rellenar campo “Nombre” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario sin introducir el campo “Nombre” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
10	Modificar datos personales de un alumno sin rellenar campo “Apellido 1” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario sin introducir el campo “Apellido 1” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Apellido 1' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
11	Modificar datos personales de un alumno sin rellenar campo “NIF” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario sin introducir el campo “NIF” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'NIF' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
12	Modificar datos personales de un alumno sin rellenar campo “Teléfono” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario sin introducir el campo “Teléfono” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Teléfono' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste

13	Modificar datos personales de un alumno sin rellenar campo “Correo” del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario sin introducir el campo “Correo” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Correo' del Responsable.”	OK	La información del Responsable se valida si se introduce algún dato de éste
14	Modificar datos personales de un alumno con campo “NIF” inválido del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “NIF” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “El NIF del Responsable introducido no es válido.”	OK	La información del Responsable se valida si se introduce algún dato de éste
15	Modificar datos personales de un alumno con campo “Teléfono” inválido del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Teléfono” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “El teléfono del Responsable introducido no es válido.”	OK	La información del Responsable se valida si se introduce algún dato de éste
16	Modificar datos personales de un alumno con campo “Correo” inválido del Responsable	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor inválido en el campo “Correo” del Responsable 8. Botón “Modificar”	Debe aparecer el mensaje “El correo del Responsable introducido no es válido.”	OK	La información del Responsable se valida si se introduce algún dato de éste

17	Modificar datos personales de un alumno con NIF repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Datos personales” 7. Rellenar el formulario con valor repetido en el campo “NIF” 8. Botón “Modificar”	Debe aparecer el mensaje “Ya hay almacenado un alumno con ese NIF.”	OK	Esta comprobación se hace si se ha rellenado el campo “NIF”

10.4.4 Clases

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver clases de un alumno	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Clases”	Debe mostrarse el horario con las clases del alumno en la parte inferior de la ventana que se abre. En la parte superior deben aparecer las clases disponibles para el alumno (las de su curso) y seleccionadas aquellas en la que el alumno está matriculado.	OK	Sólo se pueden ver las clases de los alumnos actuales
2	Seleccionar clase incompatible con horario	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Clases” 7. Seleccionar una nueva clase que sea incompatible con el horario del alumno	Debe aparecer el mensaje “No puede seleccionar esa clase, se pisa en horario con las actuales.”	OK	

3	Guardar clases de un alumno	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Clases” 7. Seleccionar las clases del alumno 8. Botón “Guardar”	Deben almacenarse las clases del alumno en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“Las clases del alumno se han guardado correctamente.”)	OK	

10.4.5 Dar de alta/baja

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Dar de baja un alumno	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Dar de alta / baja” 7. Confirmar que se desea dar de baja el alumno	Debe darse de baja el alumno (almacenándose en la fecha de baja el día actual) y al usuario le debe aparecer un mensaje indicándoselo (“El alumno ha sido dado de baja.”)	OK	El alumno se borra de todas las clases en las que estaba matriculado
2	Dar de alta un alumno (que estaba de baja)	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Alumnado” 4. Buscar el alumno 5. Seleccionar el alumno 6. Botón “Dar de alta / baja” 7. Confirmar que se desea dar de alta el alumno	Debe darse de alta el alumno (almacenándose en la fecha de alta el día actual y eliminándose la fecha de baja) y al usuario le debe aparecer un mensaje indicándoselo (“El alumno ha sido dado de alta.”)	OK	

10.5 Clases

10.5.1 Búsqueda de clases

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Búsqueda sin parámetros	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Botón “Buscar”	Debe aparecer un listado con todas las clases almacenadas en la aplicación	OK	
2	Búsqueda por curso	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Rellenar el campo “Curso” 5. Botón “Buscar”	Debe aparecer un listado de las clases del curso introducido	OK	
3	Búsqueda por asignatura	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Rellenar el campo “Asignatura” 5. Botón “Buscar”	Debe aparecer un listado de las clases de la asignatura introducida	OK	
4	Búsqueda por profesor	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Rellenar el campo “Profesor” 5. Botón “Buscar”	Debe aparecer un listado de las clases del profesor introducido	OK	

10.5.2 Nueva clase

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nueva clase con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Botón “Nueva clase” 5. Rellenar todos los campos del formulario correctamente 6. Botón “Crear”	Debe almacenarse la nueva clase en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La clase ha sido añadida.”)	OK	
2	Nueva clase sin rellenar campo “Asignatura”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Botón “Nueva clase” 5. Rellenar el formulario sin introducir el campo “Asignatura” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Asignatura'.”	OK	
3	Nuevo clase sin rellenar campo “Profesor”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Botón “Nueva clase” 5. Rellenar el formulario sin introducir el campo “Profesor” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Profesor'.”	OK	
4	Nueva clase sin rellenar campo “Tarifa”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Botón “Nueva clase” 5. Rellenar el formulario sin introducir el campo “Tarifa” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Tarifa'.”	OK	

5	Nueva clase con horario incompatible para el profesor	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Botón “Nueva clase” 5. Rellenar el formulario seleccionando un horario que el profesor tiene ocupado con otras clases. 6. Botón “Crear”	Debe aparecer el mensaje “El profesor tiene ocupado ese horario.”	OK	

10.5.3 Editar clase

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver detalle de una clase	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase 6. Botón “Editar clase”	Deben mostrarse los datos de la clase	OK	
2	Modificar datos de una clase con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase 6. Botón “Editar clase” 7. Rellenar todos los campos del formulario correctamente con los nuevos datos 8. Botón “Modificar”	Deben almacenarse los nuevos datos de la clase en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La clase ha sido modificada.”)	OK	

3	Modificar datos de una clase con horario incompatible para el profesor	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase 6. Botón “Editar clase” 7. Seleccionar un horario incompatible para el profesor 8. Botón “Modificar”	Debe aparecer el mensaje “El profesor tiene ocupado ese horario.”	OK
4	Modificar datos de una clase con horario incompatible para algún alumno	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase 6. Botón “Editar clase” 7. Seleccionar un horario incompatible para algún alumno 8. Botón “Modificar”	Debe aparecer el mensaje “Los siguientes alumnos tienen incompatibilidad con el nuevo horario: - Alumno1 - Alumno2 -…”	OK

10.5.4 Alumnos

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver alumnos de una clase	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase 6. Botón “Alumnos”	Debe mostrarse los alumnos matriculados en esa clase	OK	

10.5.5 Eliminar clase

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Eliminar una clase sin alumnos matriculados	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase (sin alumnos matriculados) 6. Botón “Eliminar clase” 7. Confirmar que se desea eliminar la clase	Debe eliminarse la clase de la BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La clase ha sido eliminada.”)	OK	
2	Eliminar una clase con alumnos matriculados	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Clases” 4. Buscar la clase 5. Seleccionar la clase (con alumnos matriculados) 6. Botón “Eliminar clase” 7. Confirmar que se desea eliminar la clase	Debe aparecer el mensaje “No se ha podido eliminar la clase ya que existen alumnos apuntados a ésta.”	OK	

10.6 Mantenimiento de Cursos

10.6.1 Nuevo curso

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nuevo curso con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Botón “Nuevo curso” 5. Rellenar todos los campos del formulario correctamente 6. Botón “Crear”	Debe almacenarse el nuevo curso en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“El curso ha sido creado.”)	OK	
2	Nuevo curso sin rellenar campo “Nivel”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Botón “Nuevo curso” 5. Rellenar el formulario sin introducir el campo “Nivel” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nivel'.”	OK	
3	Nuevo curso sin rellenar campo “Etapa”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Botón “Nuevo curso” 5. Rellenar el formulario sin introducir el campo “Etapa” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Etapa'.”	OK	

4	Nuevo curso con nivel y etapa repetidos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Botón “Nuevo curso” 5. Rellenar el formulario con valores repetidos en los campos “Nivel” y “Etapa” 6. Botón “Crear”	Debe aparecer el mensaje “Ya existe el curso introducido.”	OK	

10.6.2 Editar curso

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver detalle de un curso	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Seleccionar el curso 5. Botón “Editar curso”	Deben mostrarse los datos del curso	OK	
2	Modificar datos de un curso con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Seleccionar el curso 5. Botón “Editar curso” 6. Rellenar todos los campos del formulario correctamente con los nuevos datos 7. Botón “Modificar”	Deben almacenarse los nuevos datos del curso en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“El curso ha sido modificado.”)	OK	

3	Modificar datos de un curso sin rellenar campo “Nivel”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Seleccionar el curso 5. Botón “Editar curso” 6. Rellenar el formulario con los nuevos datos sin introducir el campo “Nivel” 7. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nivel'.”	OK
4	Modificar datos de un curso sin rellenar campo “Etapa”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Seleccionar el curso 5. Botón “Editar curso” 6. Rellenar el formulario con los nuevos datos sin introducir el campo “Etapa” 7. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Etapa'.”	OK
5	Modificar datos de un curso con etapa y nivel repetidos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Cursos” 4. Seleccionar el curso 5. Botón “Editar curso” 8. Rellenar el formulario con valores repetidos en los campos “Nivel” y “Etapa” 9. Botón “Modificar”	Debe aparecer el mensaje “Ya existe el curso introducido.”	OK

10.6.3 Eliminar curso

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Eliminar un curso sin asignaturas asignadas	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Mantenimiento de Cursos" 4. Seleccionar el curso (sin asignaturas asignadas) 5. Botón "Eliminar curso" 6. Confirmar que se desea eliminar el curso	Debe eliminarse el curso de la BBDD y al usuario le debe aparecer un mensaje indicándoselo ("El curso ha sido eliminado.")	OK	
2	Eliminar un curso con asignaturas asignadas	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Mantenimiento de cursos" 4. Seleccionar el curso (con asignaturas asignadas) 5. Botón "Eliminar curso" 6. Confirmar que se desea eliminar el curso	Debe aparecer el mensaje "No se ha podido eliminar el curso ya que existen asignaturas almacenadas de este curso."	OK	

10.7 Mantenimiento de Asignaturas

10.7.1 Nueva asignatura

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nueva asignatura con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Mantenimiento de Asignaturas" 4. Botón "Nueva asignatura" 5. Rellenar todos los campos del formulario correctamente 6. Botón "Crear"	Debe almacenarse la nueva asignatura en BBDD y al usuario le debe aparecer un mensaje indicándoselo ("La asignatura ha sido creada.")	OK	

2	Nueva asignatura sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Botón “Nueva asignatura” 5. Rellenar el formulario sin introducir el campo “Nombre” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK
3	Nueva asignatura sin rellenar campo “Curso”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Botón “Nueva asignatura” 5. Rellenar el formulario sin introducir el campo “Curso” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Curso'.”	OK
4	Nueva asignatura con nombre y curso repetidos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Botón “Nueva asignatura” 5. Rellenar el formulario con valores repetidos en los campos “Nombre” y “Curso” 6. Botón “Crear”	Debe aparecer el mensaje “Ya existe la asignatura introducida.”	OK

10.7.2 Editar asignatura

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver detalle de una asignatura	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura 5. Botón “Editar asignatura”	Deben mostrarse los datos de la asignatura	OK	
2	Modificar datos de una asignatura con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura 5. Botón “Editar asignatura” 6. Rellenar todos los campos del formulario correctamente con los nuevos datos 7. Botón “Modificar”	Deben almacenarse los nuevos datos de la asignatura en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La asignatura ha sido modificada.”)	OK	
3	Modificar datos de una asignatura sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura 5. Botón “Editar asignatura” 6. Rellenar el formulario con los nuevos datos sin introducir el campo “Nombre” 7. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK	

4	Modificar datos de una asignatura sin rellenar campo “Curso”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura 5. Botón “Editar asignatura” 6. Rellenar el formulario con los nuevos datos sin introducir el campo “Curso” 7. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Curso'.”	OK
5	Modificar datos de una asignatura con nombre y curso repetidos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura 5. Botón “Editar asignatura” 8. Rellenar el formulario con valores repetidos en los campos “Nombre” y “Curso” 9. Botón “Modificar”	Debe aparecer el mensaje “Ya existe la asignatura introducida.”	OK

10.7.3 Eliminar asignatura

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Eliminar una asignatura sin clases de ella	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura (sin clases de ella) 5. Botón “Eliminar asignatura” 6. Confirmar que se desea eliminar la asignatura	Debe eliminarse la asignatura de la BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La asignatura ha sido eliminada.”)	OK	

2	Eliminar una asignatura con clases de ella	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Asignaturas” 4. Seleccionar la asignatura (con clases de ella) 5. Botón “Eliminar asignatura” 6. Confirmar que se desea eliminar la asignatura	Debe aparecer el mensaje “No se ha podido eliminar la asignatura ya que existen clases almacenadas de esta asignatura.”	OK	

10.8 Mantenimiento de Tarifas

10.8.1 Nueva tarifa

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Nueva tarifa con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Botón “Nueva tarifa” 5. Rellenar todos los campos del formulario correctamente 6. Botón “Crear”	Debe almacenarse la nueva tarifa en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La tarifa ha sido creada.”)	OK	
2	Nueva tarifa sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Botón “Nueva tarifa” 5. Rellenar el formulario sin introducir el campo “Nombre” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK	

3	Nueva tarifa sin rellenar campo “Descripción”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Botón “Nueva tarifa” 5. Rellenar el formulario sin introducir el campo “Descripción” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Descripción'.”	OK
4	Nueva tarifa sin rellenar campo “Precio (€/hora)”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Botón “Nueva tarifa” 5. Rellenar el formulario sin introducir el campo “Precio (€/hora)” 6. Botón “Crear”	Debe aparecer el mensaje “Debe rellenar el campo 'Precio'.”	OK
5	Nueva tarifa con nombre repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Botón “Nueva tarifa” 5. Rellenar el formulario con valor repetido en el campo “Nombre” 6. Botón “Crear”	Debe aparecer el mensaje “Ya existe la tarifa introducida.”	OK

10.8.2 Editar tarifa

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Ver detalle de una tarifa	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Seleccionar la tarifa 5. Botón “Editar tarifa”	Deben mostrarse los datos de la tarifa	OK	

2	Modificar datos de una tarifa con todos los campos correctos	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Seleccionar la tarifa 5. Botón “Editar tarifa” 6. Rellenar todos los campos del formulario correctamente con los nuevos datos 7. Botón “Modificar”	Deben almacenarse los nuevos datos de la tarifa en BBDD y al usuario le debe aparecer un mensaje indicándoselo (“La tarifa ha sido modificada.”)	OK
3	Modificar datos de una tarifa sin rellenar campo “Nombre”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Seleccionar la tarifa 5. Botón “Editar tarifa” 6. Rellenar el formulario con los nuevos datos sin introducir el campo “Nombre” 7. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Nombre'.”	OK
4	Modificar datos de una tarifa sin rellenar campo “Descripción”	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Seleccionar la tarifa 5. Botón “Editar tarifa” 6. Rellenar el formulario con los nuevos datos sin introducir el campo “Descripción” 7. Botón “Modificar”	Debe aparecer el mensaje “Debe rellenar el campo 'Descripción'.”	OK

5	Modificar datos de una tarifa sin rellenar campo "Precio (€/hora)"	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Mantenimiento de Tarifas" 4. Seleccionar la tarifa 5. Botón "Editar tarifa" 6. Rellenar el formulario con los nuevos datos sin introducir el campo "Precio (€/hora)" 7. Botón "Modificar"	Debe aparecer el mensaje "Debe rellenar el campo Precio."	OK	
6	Modificar datos de una tarifa con nombre repetido	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Mantenimiento de Tarifas" 4. Seleccionar la tarifa 5. Botón "Editar tarifa" 6. Rellenar el formulario con valor repetido en el campo "Nombre" 7. Botón "Modificar"	Debe aparecer el mensaje "Ya existe la tarifa introducida."	OK	

10.8.3 Eliminar tarifa

Caso	Título Prueba	Descripción	Resultado Esperado	OK/NoOk	Observaciones
1	Eliminar una tarifa sin clases con dicha tarifa	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección "Mantenimiento de Tarifas" 4. Seleccionar la tarifa (sin clases con dicha tarifa) 5. Botón "Eliminar tarifa" 6. Confirmar que se desea eliminar la tarifa	Debe eliminarse la tarifa de la BBDD y al usuario le debe aparecer un mensaje indicándoselo ("La tarifa ha sido eliminada.")	OK	

2	Eliminar una tarifa con clases con dicha tarifa	1. Acceder a la aplicación 2. Loguearse en la aplicación 3. Entrar en la sección “Mantenimiento de Tarifas” 4. Seleccionar la tarifa (con clases con dicha tarifa) 5. Botón “Eliminar tarifa” 6. Confirmar que se desea eliminar la tarifa	Debe aparecer el mensaje “No se ha podido eliminar la tarifa ya que existen clases almacenadas con esta tarifa.”	OK

11 PUESTA EN PRODUCCIÓN

Las pasiones son necesarias como los vientos: son necesarias para poner en funcionamiento todas las cosas, aunque con frecuencia originan huracanes.

John Allen Paulos

En este capítulo se va a explicar cómo poner en marcha la aplicación web. Primero se indicará como instalar un servidor de base de datos MySQL y cómo crear el esquema SOPHISTAS. Posteriormente se describirá la instalación del servidor Apache Tomcat y el despliegue de la aplicación en dicho servidor.

El sistema operativo donde se instalará estos servidores será Windows 7, que es el del que disponen en la academia.

11.1 Instalación del servidor MySQL y creación del esquema SOPHISTAS

11.1.1 Instalación del servidor de base de datos MySQL

Para la instalación del servidor de base de datos MySQL se han de seguir los siguientes pasos:

- Desde la página web de MySQL, sección "Download MySQL Community Server" (https://dev.mysql.com/downloads/mysql/) hay que descargar el paquete "Windows (x86, 32-bit), MySQL Installer MSI".

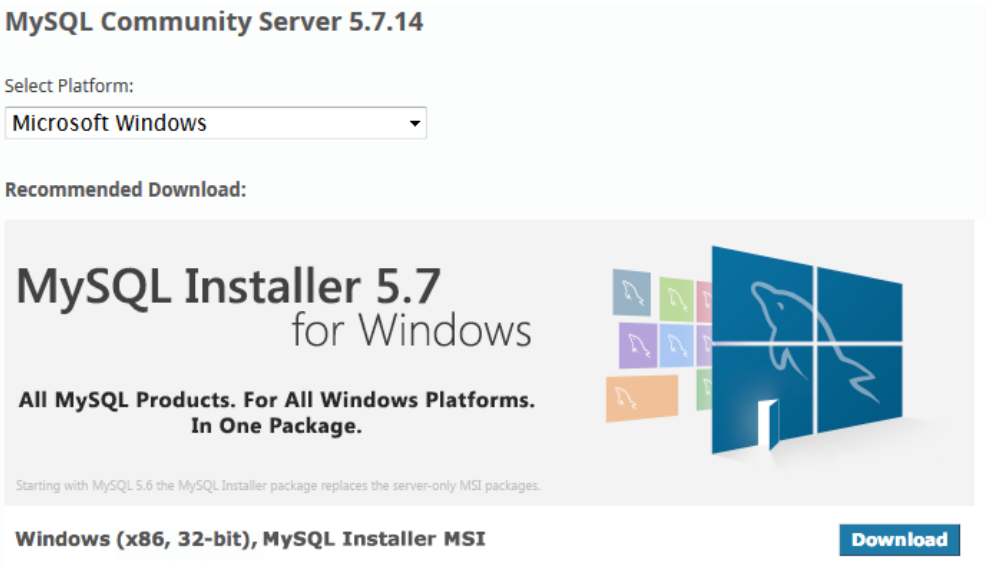


Figura 11-1. Descarga del MySQL Community Server 5.7.14

Se debe seleccionar la plataforma entre diversas opciones que se ofrecen. En este caso la elección va a ser el paquete mysql-installer-community-5.7.14.0.msi.

- Cuando ya esté la descarga completada, se ejecuta el fichero descargado. Se iniciará el asistente para la instalación.
- Se aceptan los términos de licencia.

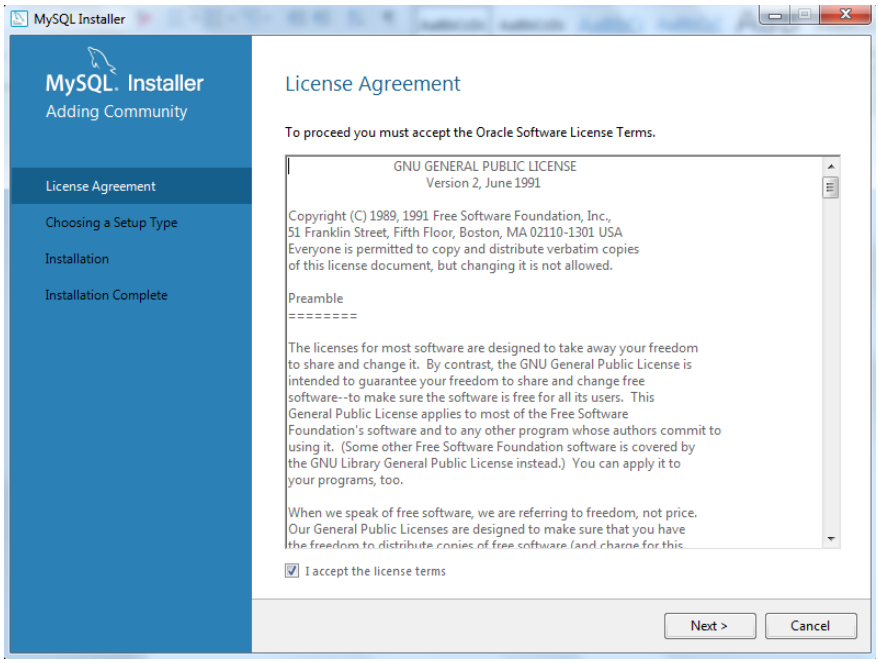


Figura 11-2. Aceptación de los *Oracle Software License Terms*

- Se elige el tipo de instalación “Developer Default”. Al seleccionar este tipo, se van a instalar, entre otros productos, el servidor MySQL y el MySQL Workbench, una aplicación GUI para la gestión del servidor.

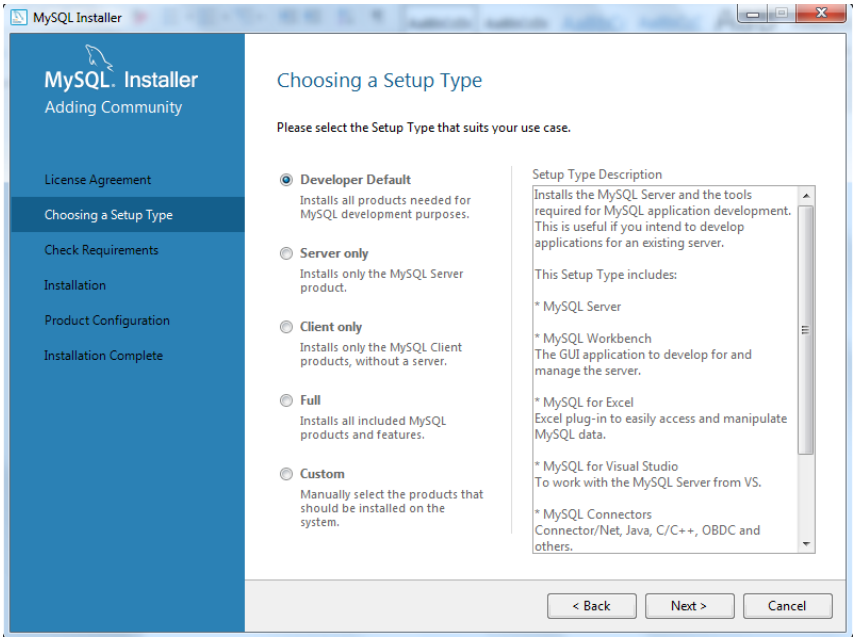


Figura 11-3. Tipo de instalación de MySQL

- Botón “Execute” para instalar los diferentes productos.

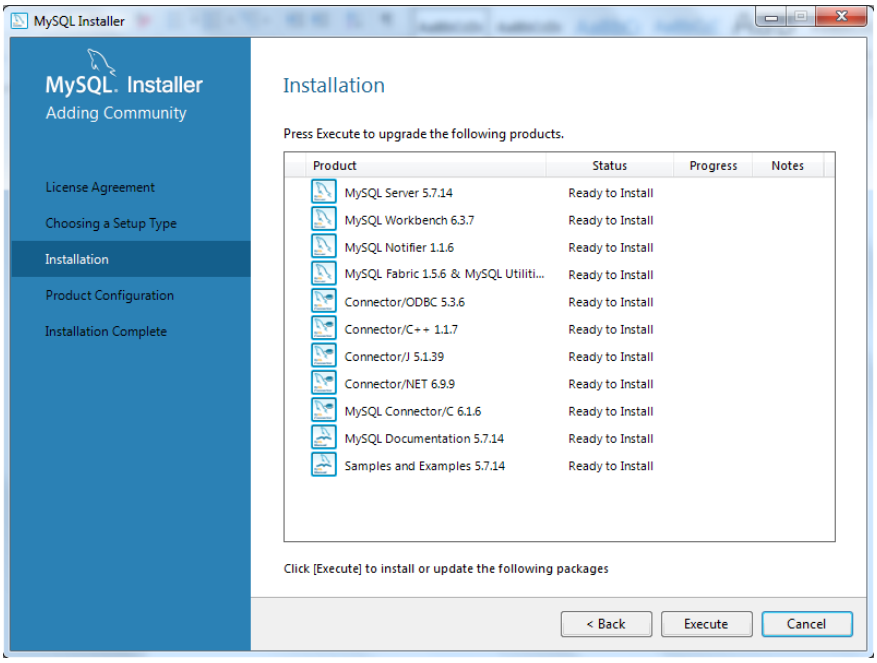


Figura 11-4. Instalación de los diferentes productos MySQL

- Posteriormente, se inicia el asistente de configuración de MySQL Server. En primer lugar, se elige el tipo de configuración para el servidor MySQL, "Development Machine"; y las opciones de red, que se van a dejar tal como están.

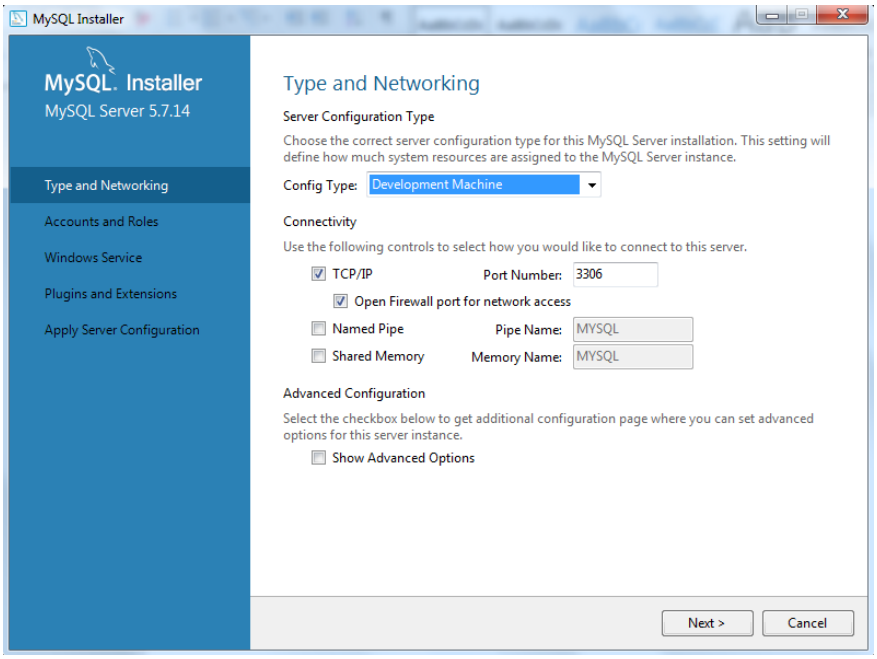


Figura 11-5. Tipo de configuración del servidor MySQL y opciones de red

- El siguiente paso es la configuración de los usuarios. Se debe escribir una contraseña para el usuario administrador y añadir el usuario "sophista" con contraseña "S0ph1st4" y rol "DB Manager".

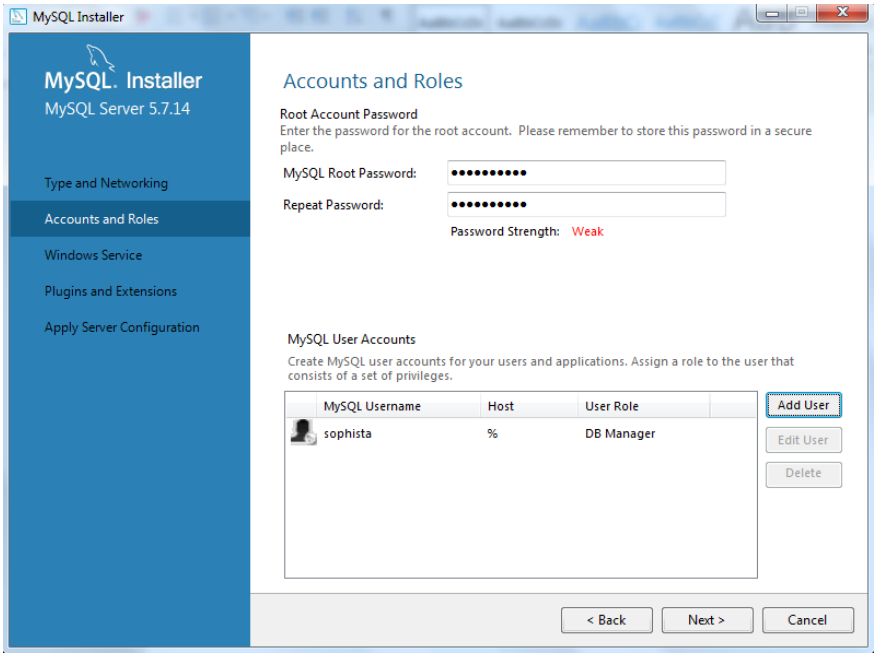


Figura 11-6. Usuarios del servidor MySQL

- Para el resto de opciones de configuración se dejan los valores por defecto. Finalmente se pulsa el botón “Execute” para aplicar la configuración del servidor.

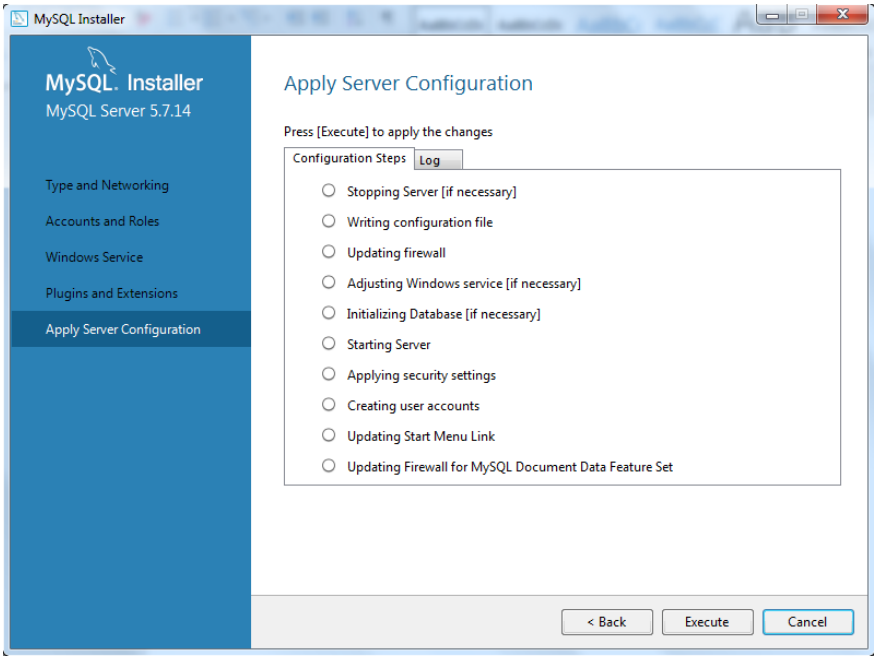


Figura 11-7. Aplicación de la configuración del servidor

Tras la instalación y configuración el asistente indicará que la instalación de MySQL Community Server se ha completado.

Se puede comprobar en los "Servicios" de Windows 7 que se ha creado el servicio "MySQL57":

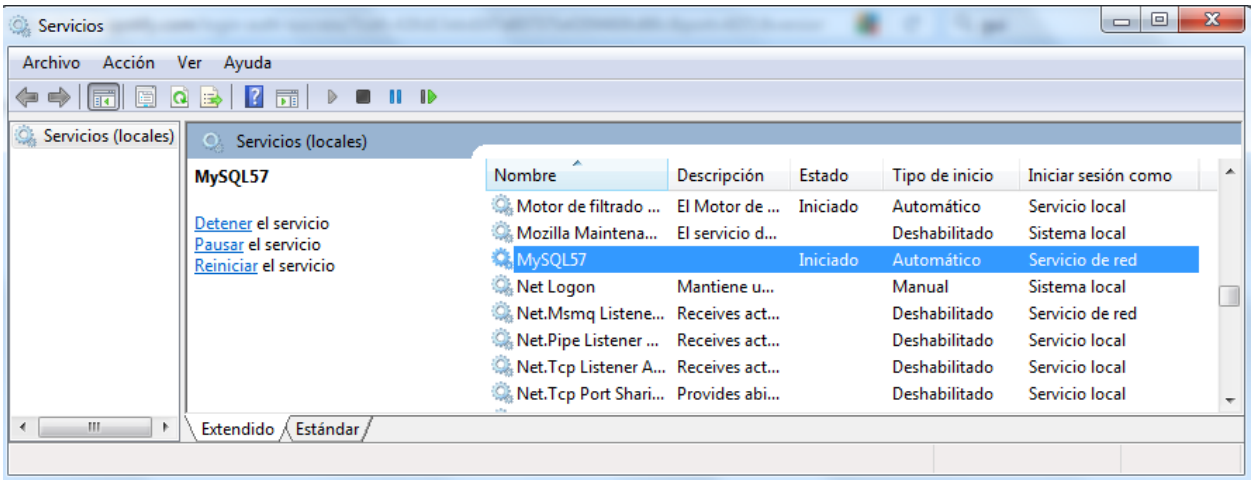


Figura 11-8. Servicio MySQL57

Además, pulsando el botón "Inicio" de Windows, en "Todos los programas" se tendrá la carpeta "MySQL" con los productos instalados:

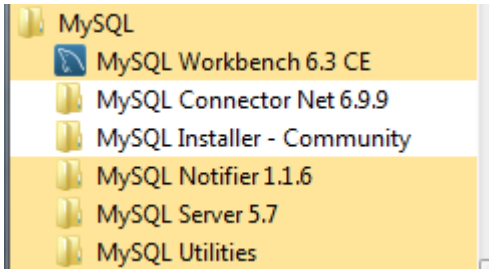


Figura 11-9. Productos MySQL instalados

11.1.2 Creación del esquema SOPHISTAS

Para la creación del esquema SOPHISTAS sobre el que se va a implementar el modelo de datos de la aplicación se va a hacer uso del MySQL Workbench.

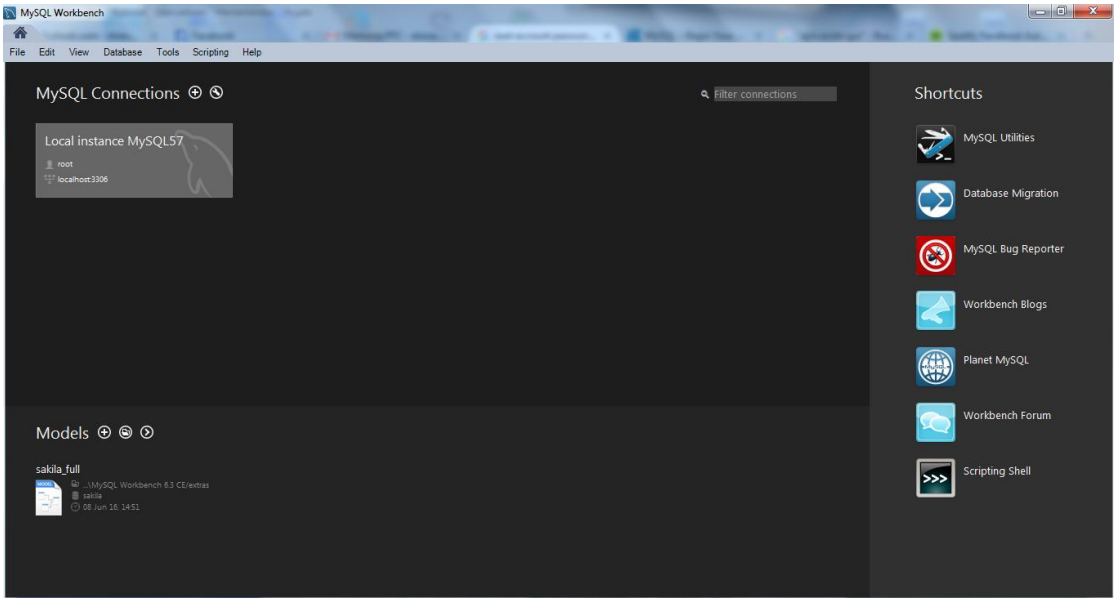


Figura 11-10. MySQL Workbench

Se selecciona la única conexión que existe en ese momento, “Local Instance MySQL57”. (Para conectarse hay que insertar la contraseña del administrador que se indicó en la instalación.)

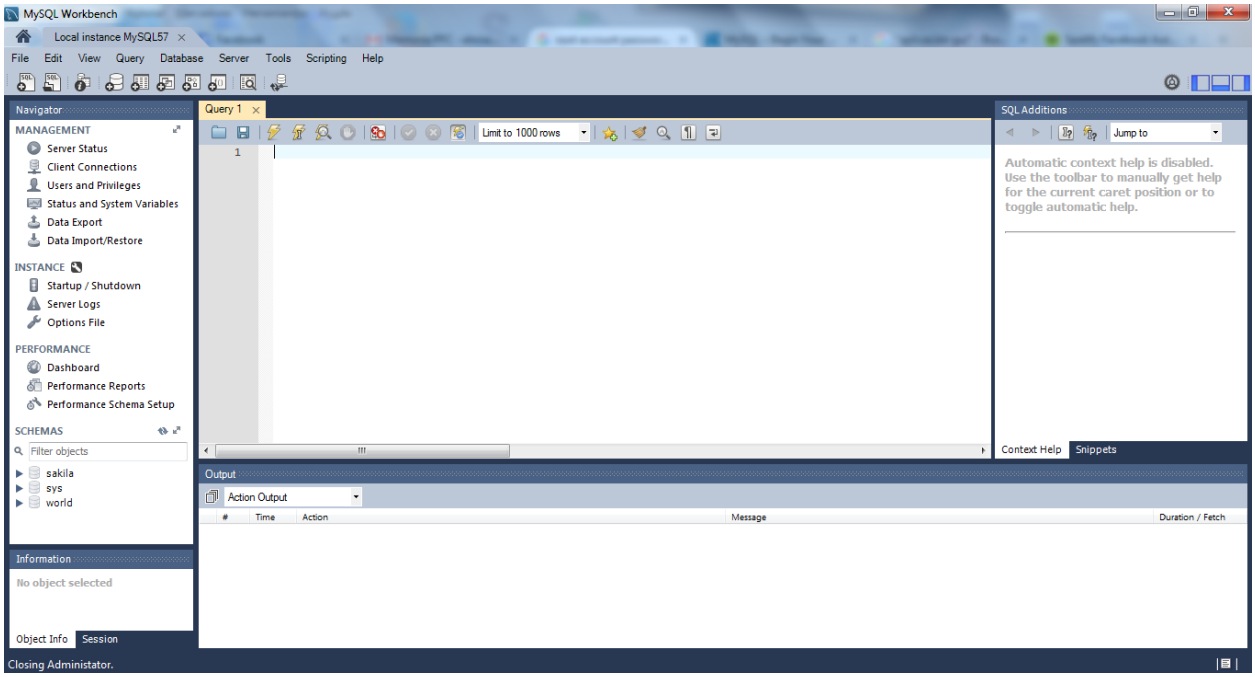


Figura 11-11. Local instance MySQL57

Ya conectados, se debe seleccionar la tarea “Data Import/Restore”, en el menú de la izquierda; y una vez en la pantalla de administración, seleccionar la opción “Import from Self-Contained File” y el fichero sophistas.sql (que viene incluido en el CD del proyecto), y pulsar el botón “Start Import”.

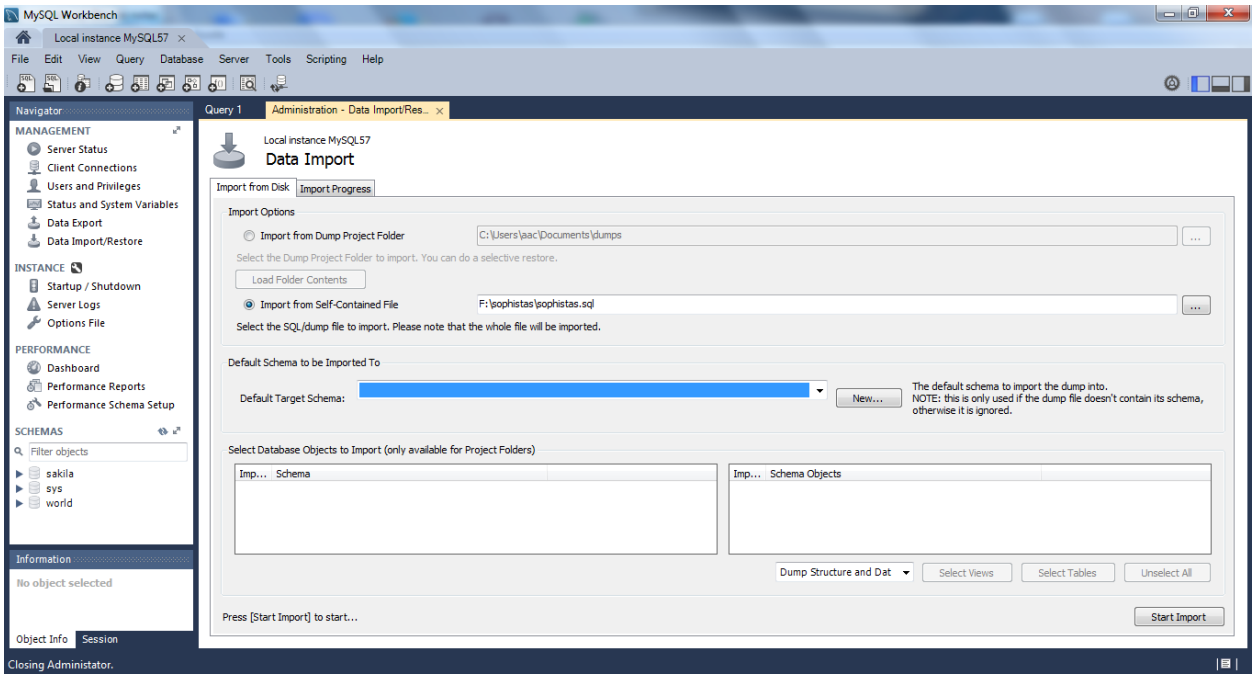


Figura 11-12. Data Import/Restore

En el menú de la izquierda, en la sección “SCHEMAS”, se puede observar cómo se ha creado el esquema SOPHISTAS con las tablas correspondientes del modelo de datos.

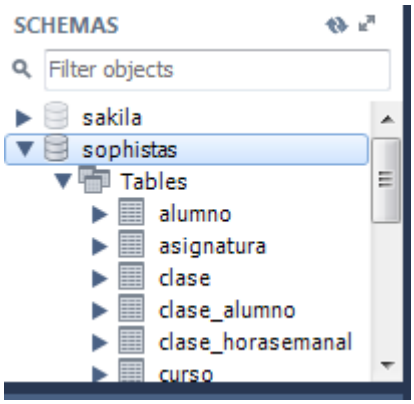


Figura 11-13. Esquema SOPHISTAS creado

Por último, al usuario "sophista" se le va a otorgar todos los permisos sobre este esquema. Esta labor se lleva a cabo desde "Users and Privileges" del menú de la izquierda.

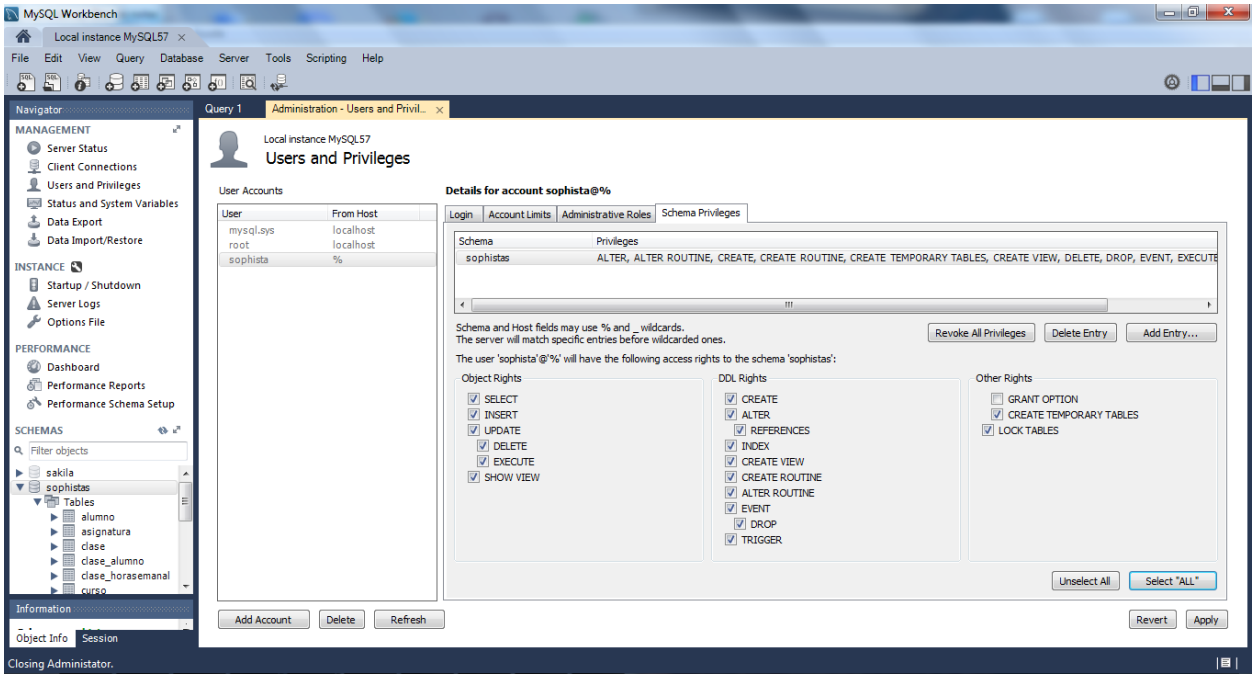


Figura 11-14. Permisos del usuario "sophista"

11.2 Instalación del servidor Apache Tomcat y despliegue de la aplicación

11.2.1 Servidor Apache Tomcat

El servidor Apache Tomcat [19] es un contenedor de código abierto de aplicaciones web basadas en Java. Implementa las especificaciones de los *servlets* y de JavaServer Pages (JSP) de Oracle Corporation (anteriormente de Sun Microsystems). En concreto, la versión 7 (que es la que se va a instalar), implementa las especificaciones de Servlet 3.0 y de JSP 2.2.

Fue creado en el marco del subproyecto Apache-Jakarta; sin embargo, debido a su popularidad, constituye ahora un proyecto Apache independiente, donde es soportado y mejorado por un grupo de voluntarios de la comunidad Java de código abierto. Apache Tomcat es muy estable y tiene todas las características de un contenedor de aplicaciones web comercial - sin embargo, se presenta bajo la licencia *Open Source Apache License*.

Figura 11-15. Logo del servidor Apache Tomcat

11.2.2 Instalación del servidor Apache Tomcat

Para la instalación del servidor Apache Tomcat se han de seguir los siguientes pasos:

- En primer lugar, se procede a la descarga del producto desde la página web oficial, https://tomcat.apache.org/download-70.cgi.

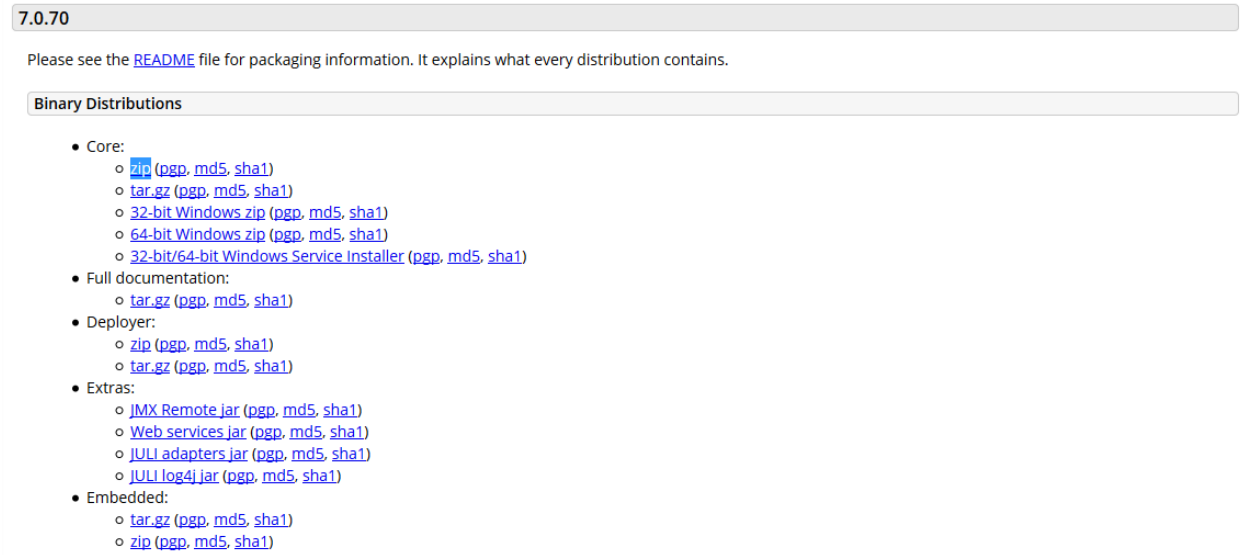


Figura 11-16. Descarga del servidor Apache Tomcat 7.0.70

Se va a usar la versión Core contenida en un zip (aunque es un tanto manual evita problemas que produce Windows 7).

- Se descomprime el archivo descargado y se renombra a tomcat7 como muestra la siguiente imagen.

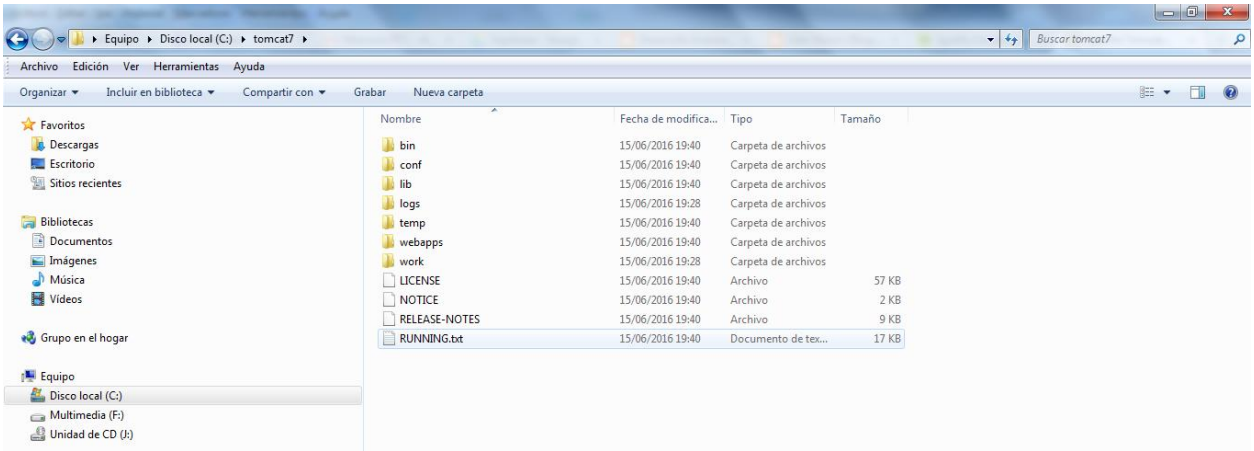


Figura 11-17. Directorio del servidor Apache Tomcat

- Se establecen las siguientes variables de entorno:
  - JRE_HOME: Debe apuntar a la instalación del *Java Runtime*.

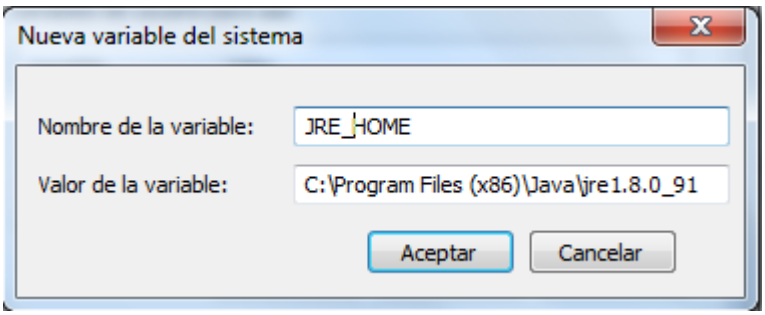


Figura 11-18. Variable de entorno JRE_HOME

  - CATALINA_HOME: Debe apuntar al directorio de instalación del Apache Tomcat.

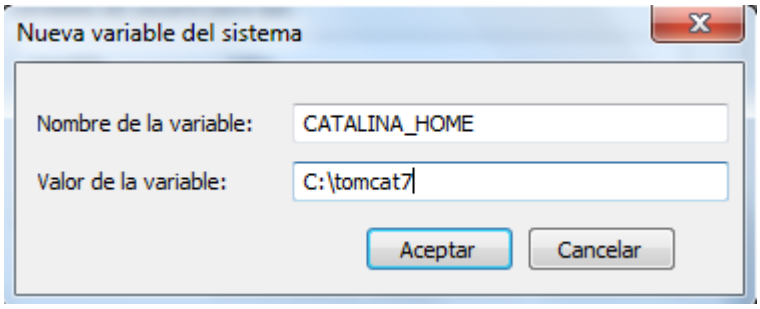


Figura 11-19. Variable de entorno CATALINA_HOME

Tras establecer las variables se recomienda reiniciar el equipo.

- Se crea un usuario administrador del Tomcat. Para ello hay que editar el fichero tomcat-users.xml dentro del directorio conf y colocar la credencial de administrador:

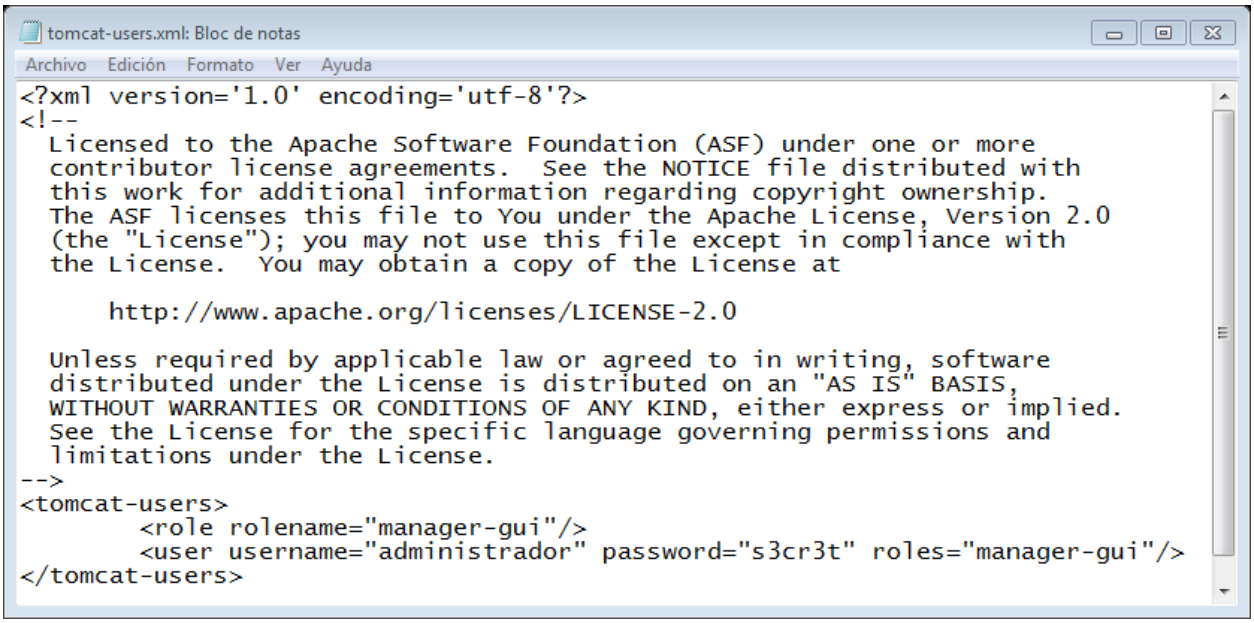


Figura 11-20. Usuario administrador del servidor Apache Tomcat

Ahora ya se puede iniciar el servidor utilizando el archivo startup.bat situado en el directorio bin.

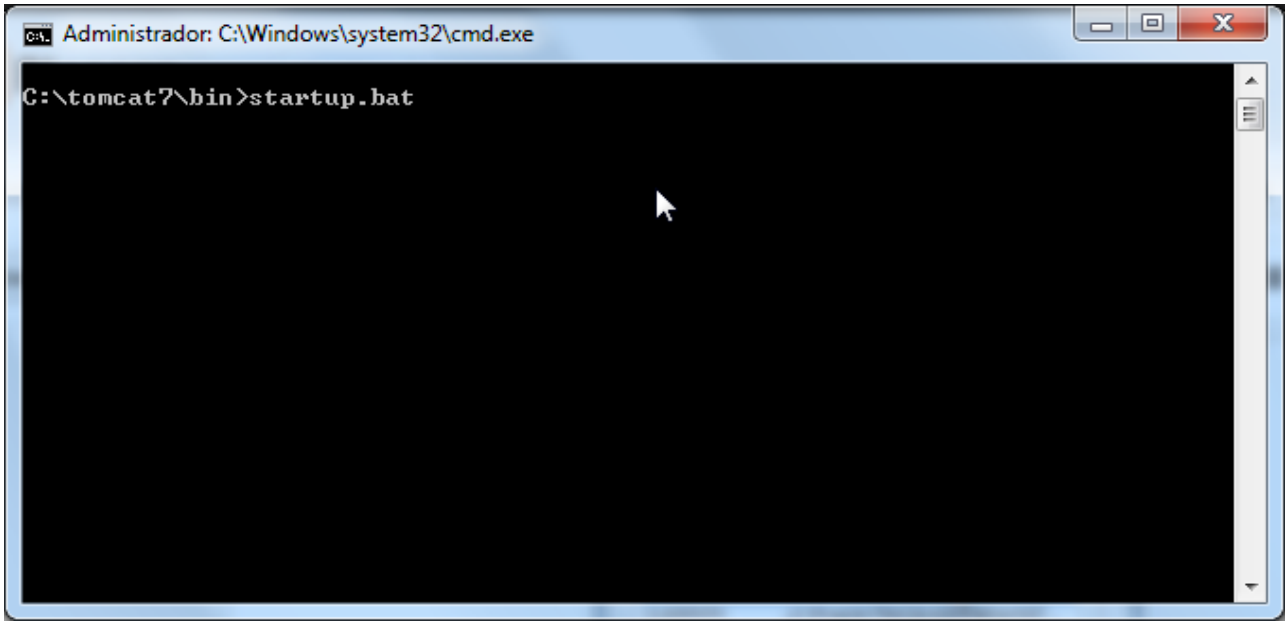


Figura 11-21. Iniciar el servidor Apache Tomcat

Para comprobar que la instalación ha sido correcta, se abre el navegador y se accede a la URL http://localhost:8080/. Debe mostrarse la página de bienvenida del servidor Apache Tomcat 7.

Figura 11-22. Página de bienvenida del servidor Apache Tomcat

11.2.3 Despliegue de la aplicación "Gestión de Sophistas Academia"

Para el despliegue de la aplicación en el servidor Apache Tomcat se han de seguir los siguientes pasos:

- En primer lugar, se abre la página inicial del Apache Tomcat, accediendo a la URL http://localhost:8080/.
- Se accede al "Gestor de Aplicaciones Web de Tomcat" desde el botón "Manager App" de la página de inicio. Se deben introducir el nombre de usuario y la clave del usuario administrador que se creó anteriormente.

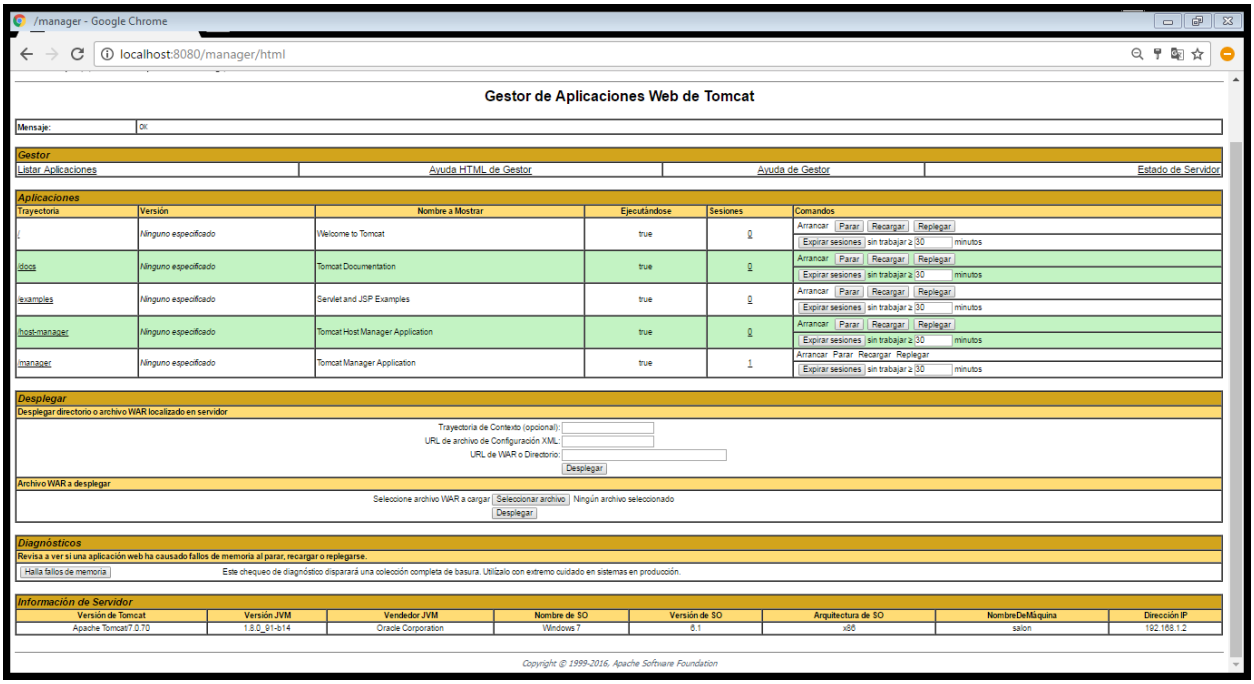


Figura 11-23. Gestor de Aplicaciones Web de Tomcat

- Casi al final, aparece un formulario llamado Desplegar – Archivo WAR a desplegar. En dicho formulario se debe seleccionar el fichero SOPHISTAS.war (que viene incluido en el CD del proyecto), y pulsar el botón "Desplegar".

  Si el despliegue ha ido bien, debe aparecer un mensaje de OK en la parte superior de la página.
- Si se accede desde este equipo a la URL http://localhost:8080/SOPHISTAS/ debe aparecer la página de login de la aplicación.

ANEXO A. TECNOLOGÍAS USADAS EN EL DESARROLLO

En este anexo se van a describir las tecnologías usadas para el desarrollo de la aplicación: el entorno de desarrollo Eclipse, la herramienta Maven y las pruebas unitarias con JUnit.

Eclipse

Para el desarrollo de la aplicación web "Gestión de Sophistas Academia" se ha hecho uso del "Eclipse Java EE IDE for Web Developers".

Eclipse [20] es un IDE, entorno de desarrollo integrado, que está compuesto por un conjunto de herramientas de programación de código abierto y multiplataforma para el desarrollo de aplicaciones.

Figura A-1. Logo de Eclipse

Está diseñado para ser extendido de forma indefinida a través de *plugins*. No tiene en mente un lenguaje específico, sino que es un IDE genérico, aunque goza de mucha popularidad entre la comunidad de desarrolladores del lenguaje Java usando el *plugin* JDT[20] que viene incluido en la distribución estándar del IDE.

Proporciona herramientas para la gestión de espacios de trabajo, escribir, desplegar, ejecutar y depurar aplicaciones.

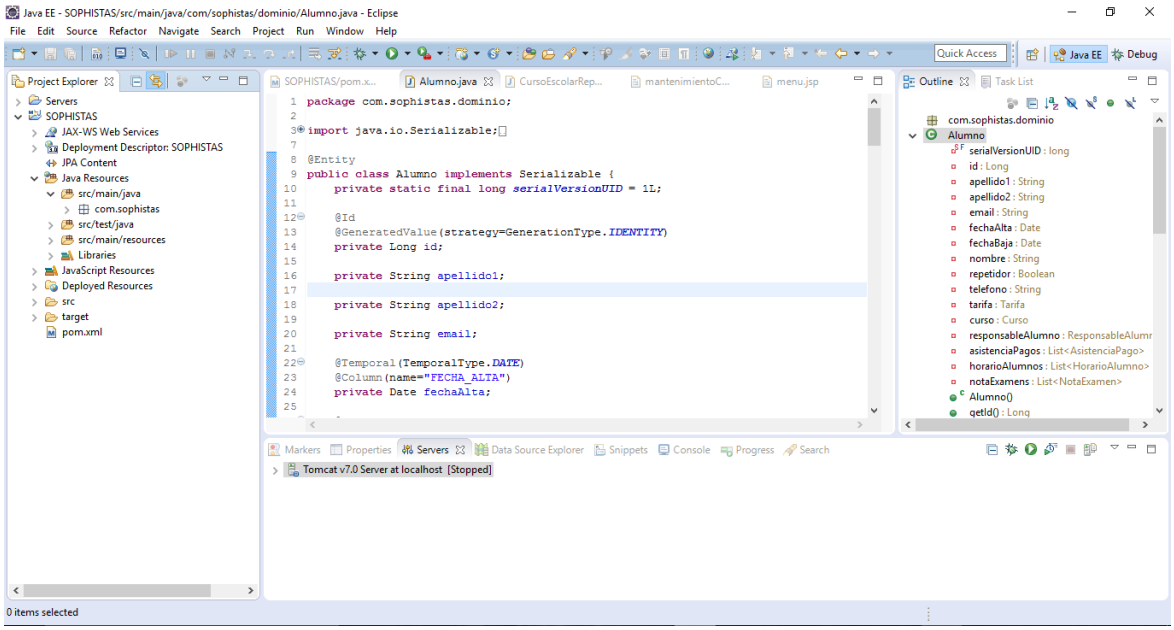


Figura A-2. Entorno de desarrollo Eclipse, perspectiva Java EE

[20] *Java Development Tools*

Eclipse fue desarrollado originalmente por IBM. Ahora es desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

La versión utilizada de Eclipse para el desarrollo de la aplicación es la versión Mars, es decir, la 4.5.

Maven

Maven [21] es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Actualmente es un proyecto (de código abierto) de nivel superior de la Apache Software Foundation.

Maven está basado en POM (*Project Object Model*) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Cada proyecto tiene la información necesaria en un descriptor XML (por defecto el fichero pom.xml).

Maven viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado.

Una característica clave de Maven es que está listo para usar en red. El motor incluido en su núcleo puede dinámicamente descargar *plugins* de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos *Open Source* en Java, de Apache y otras organizaciones y desarrolladores.

Crear proyecto Maven

Para empezar a desarrollar la aplicación, se creó un proyecto Maven en Eclipse. Para ello, se selecciona la opción del menú New -> Other… Aparece un listado del que se elige Maven Project:

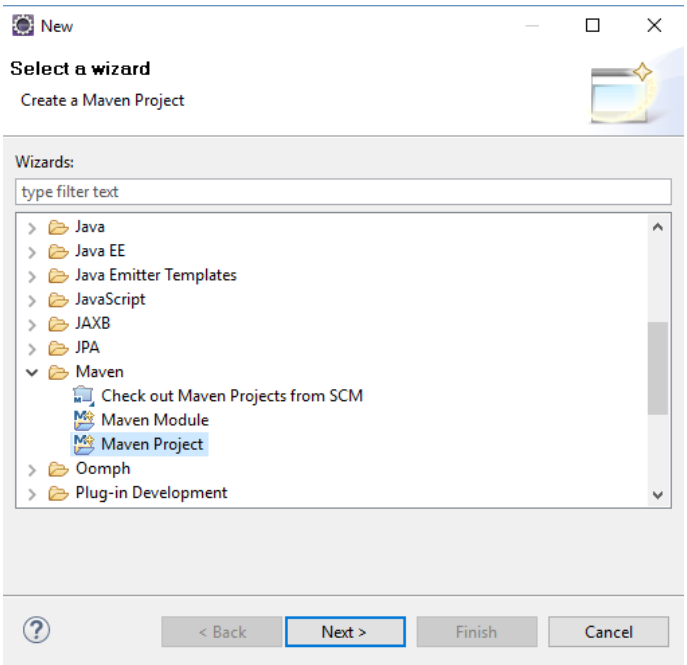


Figura A-3. Nuevo proyecto Maven

Se indica que se quiere crear un proyecto simple y que se desea usar el espacio de trabajo por defecto:

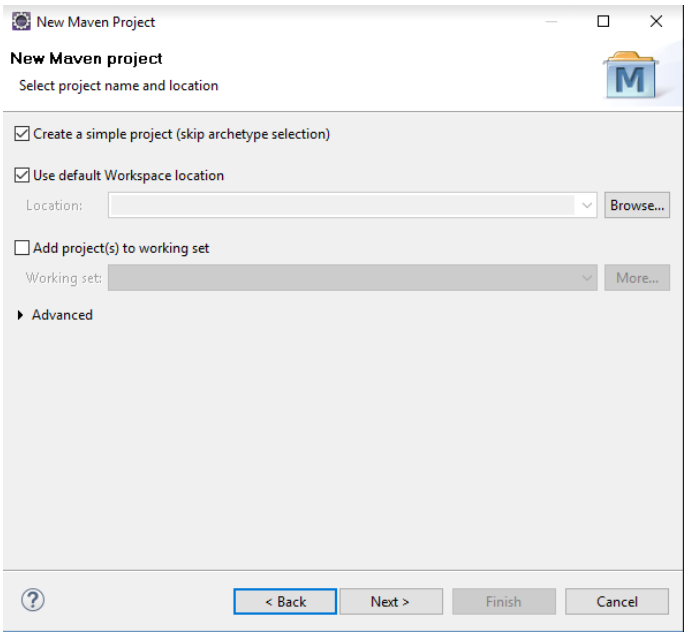


Figura A-4. Nuevo proyecto Maven 2

Por último, se indican las propiedades necesarias para crear el proyecto Maven:

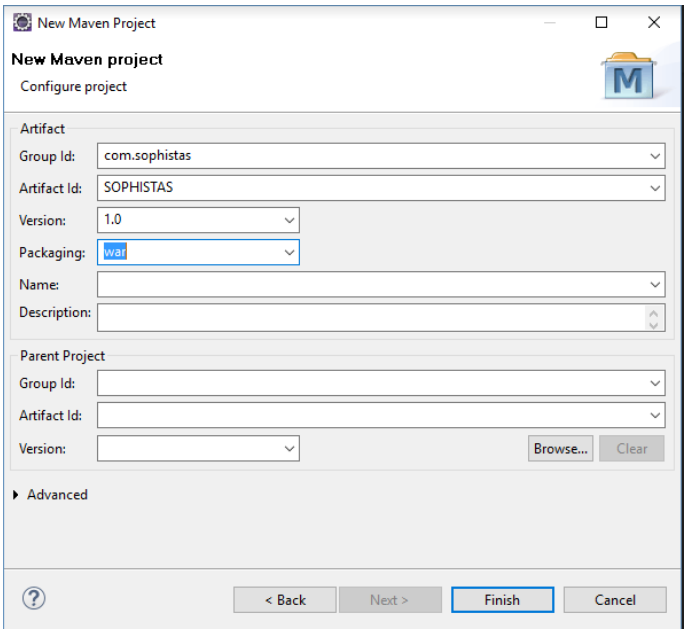


Figura A-5. Nuevo proyecto Maven 3

- *Group Id*: Es el paquete del proyecto.
- *Artifact Id*: Es el nombre del proyecto.
- *Version*: Es el número de versión del proyecto.
- *Packaging*: Es el tipo de empaquetado del proyecto.

Una vez creado el proyecto SOPHISTAS, se genera dentro de él la estructura descrita en el Diseño técnico (capítulo 8).

Fichero pom.xml

El fichero pom.xml, como se ha indicado, es el que contiene toda la información necesaria del proyecto. A continuación, a modo de ejemplo, se muestra el fichero pom.xml del presente proyecto:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
	http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.sophistas</groupId>
	<artifactId>SOPHISTAS</artifactId>
	<version>1.0</version>
	<packaging>war</packaging>

	<properties>

		<!-- Codificacion -->
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

		<!-- Versiones -->
		<java.version>1.7</java.version>
		<spring-framework.version>4.2.2.RELEASE</spring-framework.version>
		<spring-data.version>1.9.1.RELEASE</spring-data.version>
		<spring-security.version>4.0.3.RELEASE</spring-security.version>
		<jta.version>1.1</jta.version>
		<hibernate.version>5.0.5.Final</hibernate.version>
		<hibernate-validator.version>5.2.4.Final</hibernate-validator.version>
		<jstl.version>1.2</jstl.version>
		<servlet.version>2.5</servlet.version>
		<jsp.version>2.2</jsp.version>
		<tilesjsp.version>3.0.5</tilesjsp.version>
		<mysql.version>5.1.37</mysql.version>
		<junit.version>4.11</junit.version>
		<jackson.version>2.6.5</jackson.version>
		<maven.version>2.6.5</maven.version>

	</properties>

	<dependencies>

		<!-- Spring MVC -->
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-webmvc</artifactId>
			<version>${spring-framework.version}</version>
		</dependency>


		<!-- Spring and Transactions -->
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-tx</artifactId>
			<version>${spring-framework.version}</version>
		</dependency>


		<!-- Spring Data -->
		<dependency>
			<groupId>org.springframework.data</groupId>
			<artifactId>spring-data-jpa</artifactId>
			<version>${spring-data.version}</version>
		</dependency>


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

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


		<!-- Java Transaction API -->
		<dependency>
			<groupId>javax.transaction</groupId>
			<artifactId>jta</artifactId>
			<version>${jta.version}</version>
		</dependency>


		<!-- Hibernate -->
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-entitymanager</artifactId>
			<version>${hibernate.version}</version>
		</dependency>


		<!-- Hibernate Validator -->
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-validator</artifactId>
			<version>${hibernate-validator.version}</version>
		</dependency>


		<!-- Other Web dependencies -->
		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>jstl</artifactId>
			<version>${jstl.version}</version>
		</dependency>
		<dependency>
			<groupId>org.apache.tiles</groupId>
			<artifactId>tiles-jsp</artifactId>
			<version>${tilesjsp.version}</version>
		</dependency>


		<!-- MySQL -->
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<version>${mysql.version}</version>
		</dependency>


		<!-- Test Artifacts -->
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-test</artifactId>
			<version>${spring-framework.version}</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>junit</groupId>
```

```
                        <artifactId>junit</artifactId>
                        <version>${junit.version}</version>
                        <scope>test</scope>
                </dependency>


                <!-- Jackson Databind -->
                <dependency>
                        <groupId>com.fasterxml.jackson.core</groupId>
                        <artifactId>jackson-databind</artifactId>
                        <version>${jackson.version}</version>
                </dependency>


        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-compiler-plugin</artifactId>
                                <version>${maven.version}</version>
                                <configuration>
                                        <source>${java.version}</source>
                                        <target>${java.version}</target>
                                </configuration>
                        </plugin>
                </plugins>
        </build>

</project>
```

JUnit, pruebas unitarias

Una prueba unitaria es un código Java utilizado para probar otro código Java. JUnit [22] es una librería de pruebas unitarias que homogeiniza este tipo de pruebas.

Las dependencias que se necesitan añadir al pom.xml son:

```
<!-- Test Artifacts -->
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring-framework.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
</dependency>
```

Las clases con los tests van a ir dentro de la carpeta src/test/java.

A modo de ejemplo se muestra un test generado para comprobar que se creaban correctamente los nuevos cursos, es decir, para testear el método *save()* de la clase *CursoRepository*.

```
package com.sophistas.persistencia;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
```

```
import com.sophistas.configuracion.RootConfig;
import com.sophistas.dominio.Curso;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=RootConfig.class)
public class CursoRepositoryTest {

	@Autowired
	private CursoRepository cursoRepository;

	@Test
	public void test() {
		Curso cursoTest = new Curso();
		cursoTest.setNivel(1);
		cursoTest.setEtapa("PRIMARIA");
		cursoRepository.save(cursoTest);
	}

}
```

Para ejecutarlo, se navega hasta la clase en el *Project Explorer* de Eclipse y clic con el botón derecho -> Run as… -> JUnit Test.

REFERENCIAS

[1] C. Walls, Spring, Madrid: Anaya, 2015.

[2] «Spring Framework» [En línea]. Available: https://projects.spring.io/spring-framework/. [Último acceso: 04 Septiembre 2016].

[3] «MySQL» [En línea]. Available: https://www.mysql.com/. [Último acceso: 04 Septiembre 2016].

[4] «Hibernate ORM» [En línea]. Available: http://hibernate.org/orm/. [Último acceso: 04 Septiembre 2016].

[5] «Java Persistence API» [En línea]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. [Último acceso: 04 Septiembre 2016].

[6] E. Pérez Martínez, Hibernate: Persistencia de objetos en JEE, Paracuellos de Jarama: Ra-Ma, 2015.

[7] «Spring Data JPA» [En línea]. Available: http://projects.spring.io/spring-data-jpa/. [Último acceso: 04 Septiembre 2016].

[8] «MVC architecture» [En línea]. Available: https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture. [Último acceso: 04 Septiembre 2016].

[9] «Spring Web MVC framework» [En línea]. Available: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html. [Último acceso: 04 Septiembre 2016].

[10] «Using Bean Validation - The Java EE 6 Tutorial» [En línea]. Available: http://docs.oracle.com/javaee/6/tutorial/doc/gircz.html. [Último acceso: 04 Septiembre 2016].

[11] «Hibernate Validator» [En línea]. Available: http://hibernate.org/validator/. [Último acceso: 04 Septiembre 2016].

[12] «Apache Tiles» [En línea]. Available: https://tiles.apache.org/. [Último acceso: 04 Septiembre 2016].

[13] «JavaServer Pages Technology» [En línea]. Available: http://www.oracle.com/technetwork/java/jsp-138432.html. [Último acceso: 04 Septiembre 2016].

[14] «JavaSript | MDN» [En línea]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript. [Último acceso: 04 Septiembre 2016].

[15] «jQuery» [En línea]. Available: https://jquery.com/. [Último acceso: 04 Septiembre 2016].

[16] «Ext JS 6.0.0 - Classic Toolkit» [En línea]. Available: http://docs.sencha.com/extjs/6.0.0-classic/index.html. [Último acceso: 04 Septiembre 2016].

[17] «Cascading Style Sheets» [En línea]. Available: https://www.w3.org/Style/CSS/. [Último acceso: 04 Septiembre 2016].

[18] «Spring Security» [En línea]. Available: http://projects.spring.io/spring-security/. [Último acceso: 04 Septiembre 2016].

[19] «Apache Tomcat» [En línea]. Available: http://tomcat.apache.org/. [Último acceso: 04 Septiembre 2016].

[20] «Mars Eclipse» [En línea]. Available: https://eclipse.org/mars/. [Último acceso: 04 Septiembre 2016].

[21] «Maven» [En línea]. Available: https://maven.apache.org/. [Último acceso: 04 Septiembre 2016].

[22] «JUnit» [En línea]. Available: http://junit.org/junit4/. [Último acceso: 04 Septiembre 2016].

GLOSARIO

ACL: *Access Control List*

AI: *Auto Increment*

AJAX: *Asynchronous JavaScript And XML*

AOP: *Aspect-Oriented Programming*

API: *Application Programming Interface*

BBDD: Base de Datos

CAS: *Central Authentication Service*

CRUD: *Create, Read, Update and Delete*

CSRF: *Cross-Site Request Forgery*

CSS: *Cascading Style Sheets*

DB: *Data Base*

DI: *Dependency Injection*

DOM: *Document Object Model*

DSL: *Domain Specific Language*

DTD: *Document Type Definition*

DTO: *Data Transfer Object*

EE: *Enterprise Edition*

FK: *Foreign Key*

GNU: *GNU's Not Unix*

GPL: *General Public License*

GUI: *Graphical User Interface*

HTML: *HyperText Markup Language*

HTTP: *Hypertext Transfer Protocol*

IDE: *Integrated Development Environment*

JDBC: *Java Database Connectivity*

JDT: *Java Development Tools*

JMS: *Java Message Service*

JPA: *Java Persistence API*

JPQL: *Java Persistence Query Language*

JRE: *Java Runtime Environment*

JS: *JavaScript*

JSON: *JavaScript Object Notation*

JSP: *JavaServer Pages*

JSR: *Java Specification Request*

JVM: *Java Virtual Machine*

LDAP: *Lightweight Directory Access Protocol*

LGPL: *Lesser General Public License*

MSI: *Microsoft Installer*

MVC: *Model–View–Controller*

NIF: Número de Identificación Fiscal

ORM: *Object-Relational Mapping*

PK: *Primary Key*

POJO: *Plain Old Java Object*

POM: *Project Object Model*

SpEL: *Spring Expression Language*

SQL: *Structured Query Language*

SSL: *Secure Sockets Layer*

UI: *User Interface*

UQ: *Unique*

URI: *Uniform Resource Identifier*

URL: *Uniform Resource Locator*

WAR: *Web application ARchive*

XML: *eXtensible Markup Language*