UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

ESTUDIO DEL FRAMEWORK DE DESARROLLO WEB DJANGO

Roberto Caldera Vergara

Septiembre / 2017

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Carrera

ESTUDIO DEL FRAMEWORK DE DESARROLLO WEB DJANGO

Autor: **Roberto Caldera Vergara**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: ______________________________________

Vocal 1º: ______________________________________

Vocal 2º: ______________________________________

Calificación: ____________________

Alcalá de Henares a de de 2017

Agradecer a mi familia, en especial a mi madre Auxiliadora, mi padre José y mi hermano Carlos. También a mi pareja Laura por todo su apoyo.

Por otro lado, a mi tutor, Salvador Otón por su paciencia y dedicación.

1 Índice resumido

1 Índice resumido 7
2 Índice detallado 8
3 Índice de figuras 12
4 Índice de tablas 14
5 Resumen 16
6 Objetivo del trabajo 17
7 Conceptos básicos 18
8 Proyecto Django 26
9 Capa del Modelo en Django 31
10 Capa de la Vista 51
11 Capa de Plantillas 63
12 Formularios en aplicaciones web 75
13 Usuarios en Django 83
14 Sitio de administración de Django 93
15 Requisitos aplicación a desarrollar 100
16 Implementación de la aplicación 103
17 Manual de usuario de la aplicación 118
18 Conclusiones 133
19 Bibliografía 135

2 Índice detallado

1 Índice resumido 7
2 Índice detallado 8
3 Índice de figuras 12
4 Índice de tablas 14
5 Resumen 16
6 Objetivo del trabajo 17
7 Conceptos básicos 18
7.1 Aplicación Web 18
7.2 Definición de Framework Web 18
7.3 Patrón Modelo-Vista-Controlador 20
7.3.1 Definición de Patrón Software 20
7.3.2 Patrón Modelo – Vista – Controlador 21
7.4 Introducción a Django 22
7.4.1 Patrón Modelo – Vista – Controlador en Django 23
7.4.2 Requisitos de Django 25
8 Proyecto Django 26
8.1 Introducción 26
8.2 Diferencia entre Proyecto y Aplicación Django 26
8.3 Crear Proyecto en Django 26
8.4 Servidores Web en Django 29
9 Capa del Modelo en Django 31
9.1 Introducción 31
9.2 Modelos 31
9.2.1 Modelos en Django 31
9.2.2 Campos 32
9.2.2.1 Opciones de campos 32
9.2.2.2 Tipos de campos 33
9.2.2.3 Claves primarias automáticas 34
9.2.2.4 Nombre de campo Verbose 35
9.2.2.5 Relaciones entre campos 35
9.2.2.6 Los modelos a través de archivos 36
9.2.2.7 Restricciones a los nombres de los campos 36
9.3 Opciones Meta 37

9.4 Manager de un modelo 38
9.5 Métodos de los modelos 38
9.6 Herencia de modelos 39
9.6.1 Clases base abstractas 39
9.6.2 Herencia multi-tablas 40
9.6.3 Modelo Proxy 40
9.7 Paquetes de modelos 40
9.8 Usar los modelos 41
9.9 Consultas sobre los modelos 41
9.9.1 Crear y actualizar objetos 41
9.9.2 Seleccionar objetos 43
9.9.2.1 Seleccionar todos los objetos 43
9.9.2.2 Seleccionar objetos específicos aplicando filtros 43
9.9.2.3 Incrustar SQL sin procesar 45
9.9.2.4 Eliminar objetos 45
9.9.3 Cacheo de consultas 46
9.10 Migraciones 47
9.11 Configuración del motor de base de datos subyacente 49
10 Capa de la Vista 51
10.1 Introducción 51
10.2 Vistas 51
10.2.1 Mapeo de URLs - URLconf 52
10.2.1.1 Expresiones regulares 53
10.2.1.2 Función vista 54
10.2.1.3 Diccionario de argumentos 54
10.2.1.4 Nombre de patrones 54
10.2.1.5 Añadir URLconf adicionales 55
10.2.1.6 Acoplamiento débil del código 55
10.2.2 Resolución inversa de URLs 55
10.2.3 Proceso petición web 56
10.2.4 Vistas basadas en clases 56
10.2.4.1 Mixins y decorators 58
10.2.5 Vistas de error 59
10.2.5.1 Manejo de errores HTTP comunes 60
10.2.6 Componentes Middleware de Django 61

11 Capa de Plantillas 63
11.1 Plantilla en Django 63
11.2 Configuración y uso 64
11.3 Backends 66
11.4 Django template language (DTL) 66
11.4.1 Variables 67
11.4.2 Filtros 67
11.4.3 Etiquetas de bloques 67
11.4.4 Comentarios 68
11.5 Escapar código HTML 68
11.6 Herencia de plantillas 69
11.7 Incluyendo otras plantillas 70
11.8 Contexto y renderizado de plantillas 70
11.8.1 Procesadores de contexto 71
11.8.2 Renderizado de plantillas 72
11.9 Incluir funcionalidad extra a las plantillas 74
12 Formularios en aplicaciones web 75
12.1 Métodos del formulario 75
12.2 Formularios en Django 76
12.2.1 Form y ModelForm 76
12.2.2 Formularios en las vistas 78
12.2.3 Renderización de formularios 80
12.2.4 Validación de formularios 81
12.2.5 Resumen procesamiento formulario 82
13 Usuarios en Django 83
13.1 Tipos usuarios disponibles en Django 84
13.2 Sistema de autenticación en Django 85
13.2.1 Backends de autenticación en Django 86
13.3 Permisos y grupos 86
13.4 Proceso de autenticación en peticiones Web 87
13.4.1 Vistas y plantillas 87
13.4.2 Formularios 90
13.5 Sesiones de los usuarios 90
13.6 Limitar accesos a usuarios registrados 91
14 Sitio de administración de Django 93

14.1 Introducción 93
14.2 Preparación entorno administrador 93
14.3 Creación de un superusuario 94
14.4 Registrar modelos en el sitio 94
14.5 Clases ModelAdmin 95
14.6 Cambiar apariencia del sitio de administración 96
14.6.1 Personalización de Plantillas 96
14.6.2 Personalización de vistas 97
14.6.3 Personalizar formularios 97
14.7 Gestión de usuarios en el sitio de administración 97
15 Requisitos aplicación a desarrollar 100
16 Implementación de la aplicación 103
16.1 Notas iniciales 103
16.2 Flujo de la aplicación 104
16.3 Desarrollo del modelo 105
16.4 Implementación de vistas y formularios para la parte del cliente 108
16.5 Desarrollo del URLConf 112
16.6 Implementación de las plantillas de la parte del cliente 113
16.7 Configuración del sitio de administración de Django 115
17 Manual de usuario de la aplicación 118
18 Conclusiones 133
19 Bibliografía 135

3 Índice de figuras

Ilustración 1 – Diagrama petición – respuesta HTTP. 18
Ilustración 2 - Logo del Framework de Programación Web Django 22
Ilustración 3 - Diagrama correspondencia MVC – MTV. 24
Ilustración 4 - Diagrama colaboración capas MTV.. 24
Ilustración 5 - Caché de consultas en Django... 46
Ilustración 6 - Proceso de modificación de un modelo junto con migraciones. 48
Ilustración 7 - Manejo de solicitudes y respuestas HTTP por la capa de la vista. 52
Ilustración 8 - Ejemplo de Traceback de un error en una aplicación. 61
Ilustración 9 - Ejemplo de herencia de plantillas. 70
Ilustración 10 - Logotipo de Ajax 74
Ilustración 11 - Diagrama de petición – respuesta de métodos GET y POST de HTTP.75
Ilustración 12 - Proceso de autenticación de usuarios en Django. 88
Ilustración 13 - Ejemplo de contenido tabla ‘session’.. 91
Ilustración 14 - Sección ‘Autenticación y autorización’ en el sitio del administrador... 98
Ilustración 15 - Añadir grupo en el sitio del administrador... 98
Ilustración 16 - Permisos de usuario en el sitio del administrador. 99
Ilustración 17 - Asignación de grupos a usuarios en el sitio del administrador 99
Ilustración 18 - Gestión permisos de usuarios en el sitio del administrador 99
Ilustración 19 - Flujo de la aplicación desarrollada.. 104
Ilustración 20 - Modelado de la aplicación desarrollada. 105
Ilustración 21 - Diagrama de herencia de plantillas. 114
Ilustración 22 - Inicio del sitio de administración de Django...................................... 116
Ilustración 23 - Comprobación superusuario en 'Usuarios' del sitio de administración116
Ilustración 24 - Página principal del gestor de cines. 118
Ilustración 25 - Formulario de registro para nuevos clientes de la aplicación. 119
Ilustración 26 - Error de usuario existente en la aplicación. 119
Ilustración 27 - Registro confirmado para un cliente en la aplicación. 120
Ilustración 28 - Página de inicio de sesión de clientes en la aplicación. 120
Ilustración 29 - Error de usuario y contraseña incorrectos para un cliente en el login de la aplicación de gestión de cines. 121
Ilustración 30 - Error si un administrador intenta acceder al apartado del cliente en la aplicación. 121
Ilustración 31 - Página del cliente en la aplicación de gestión de cines. 122
Ilustración 32 - Formulario de cambio de contraseña para un usuario de la aplicación. 123
Ilustración 33 - Error en el formulario de cambio de contraseña de la aplicación. 123
Ilustración 34 - Mensaje de cambio de contraseña correcto en el formulario de la aplicación. 124
Ilustración 35 - Cartelera de la aplicación de gestión de cines.................................... 124
Ilustración 36 - Información de una película en la aplicación. 125
Ilustración 37 - Selección de asientos de una sala en la aplicación de gestión de cines. 126
Ilustración 38 - Formulario de cuenta bancaria en la aplicación de gestión de cines... 126
Ilustración 39 - Mensaje de confirmación de compra en la aplicación. 127

Ilustración 40 - Resumen de compra de la aplicación. 127
Ilustración 41 - Listado de entradas futuras en perfil de cliente en la aplicación......... 127
Ilustración 42 - Cierre de sesión en la aplicación. 128
Ilustración 43 - Login del sitio de administración de la aplicación. 128
Ilustración 44 - Error de login de cliente en formulario del sitio de administración de la aplicación. 129
Ilustración 45 - Vista de la página de administración de la aplicación. 129
Ilustración 46 - Visión de las películas de la aplicación en el sitio del administrador. 130
Ilustración 47 - Formulario registro película del administrador en la aplicación......... 130
Ilustración 48 - Inserción de nueva sala en el sitio del administrador de la aplicación. 131
Ilustración 49 - Visión de los usuarios de la aplicación en el sitio del administrador.. 131
Ilustración 50 - Formulario de alta de usuario administrador en la aplicación. 132
Ilustración 51 - Opciones de permisos para un usuario de la aplicación..................... 132
Ilustración 52 - Gestión de permisos para un usuario de la aplicación. 132

4 Índice de tablas

Tabla 1 - Tabla de Frameworks de programación Web. 19
Tabla 2 - Tabla de versiones de Django. 23
Tabla 3 - Tabla de versiones futuras de Django. 23
Tabla 4 - Tabla de las opciones de campos. 33
Tabla 5 - Tabla de los tipos de campos. 34
Tabla 6 - Tabla de caracteres especiales. 53
Tabla 7 - Tabla de vistas genéricas basadas en clase de Django. 57
Tabla 8 - Tabla de grupos de Mixins de Django. 58
Tabla 9 - Tabla de errores HTTP junto con subclases de HTTPResponse. 59
Tabla 10 - Tabla de cargadores de plantillas. 65
Tabla 11- Tabla de filtros en DTL. 67
Tabla 12 - Tabla de etiquetas de bloques en DTL. 67
Tabla 13 - Tabla de procesadores de contexto de Django. 71
Tabla 14 - Tabla de campos para un formulario con sus widgets. 77
Tabla 15 - Tabla de argumentos de campos. 77
Tabla 16 - Tabla de opciones de renderizado de formularios. 80
Tabla 17 - Tabla de campos de la clase User. 85
Tabla 18 - Tabla de vistas de autenticación. 89
Tabla 19 - Formularios predefinidos en Django. 90
Tabla 20 - Tabla de opciones de ModelAdmin. 95
Tabla 21 - URLConf de la aplicación desarrollada. 112

5 Resumen

El Trabajo de Fin de Grado tiene como finalidad realizar una evaluación del Framework de programación de aplicaciones Web, Django. El objetivo es mostrar al lector las ventajas y desventajas de las distintas herramientas que éste posee y su posición en relación con otros Frameworks existentes.

Con el objeto de realizar la evaluación, se han analizado cada una de las partes de Django y finalmente se ha implementado una aplicación de gestión de cines para observar los beneficios reales que éste aporta junto con su eficiencia.

Con todo ello se extraen conclusiones del Framework para conocer cuando es conveniente utilizarlo en aplicaciones reales.

Abstract

The final purpose of this project is to evaluate the Web Framework Django. The objective is to show the advantages and disadvantages of the different tools that it has and its position in relation to other existing Frameworks.

To carry out the evaluation, each one of the parts of Django has been analysed and finally a cinema management application has been implemented to observe the real benefits that it brings along and its efficiency.

At the end, we draw conclusions from the Framework to know when it is convenient to use it in real applications.

Palabras clave:

Framework, Web, Django, Python, Patrón Software.

6 Objetivo del trabajo

Como objetivo general de este trabajo se puede destacar el siguiente:

- Estudiar las diferentes tecnologías aportadas por Framework de desarrollo Web, Django.

Dentro de este objetivo general, se pueden extraer algunos específicos como pueden ser:

- Analizar la implementación particular de Django sobre el patrón software Modelo – Vista – Controlador.
- Realizar una aplicación Web a modo de ejemplo para poder estudiar de primera mano el comportamiento de las diferentes herramientas que este Framework aporta a los programadores. Esta aplicación Web deberá de cumplir con los requisitos expuestos por la supuesta empresa de gestión de cines.
- Sacar conclusiones de las características del Framework sobre de las partes analizadas y de la aplicación implementada.

7 Conceptos básicos

7.1 Aplicación Web

Una aplicación web, es una aplicación distribuida, la cual es accedida por los clientes finales mediante la conexión con un servidor vía web por una determinada red, ya sea Internet o una Intranet. Es posible en término general, describirla como un programa informático que es ejecutado en un navegador del cliente, debido a que la codificación de la misma es con algún lenguaje soportado por éstos, como, por ejemplo, HTML con JavaScript, entre otros. Dichos navegadores serán capaces de renderizar la aplicación para mostrarla a los usuarios y que puedan interactuar con ella. [1]

La comunicación existente entre los servidores y los clientes finales se realiza mediante el protocolo HTTP (*Hypertext Transfer Protocol*). Dicho protocolo de la capa de aplicación sigue el esquema petición – respuesta entre los clientes y servidores, el cual, permite la transferencia de información en la *World Wide Web*. La forma que tiene el usuario de realizar peticiones al servidor es mediante las URLs [2]. A continuación, se muestra un resumen de un intercambio de peticiones entre un servidor y un cliente:

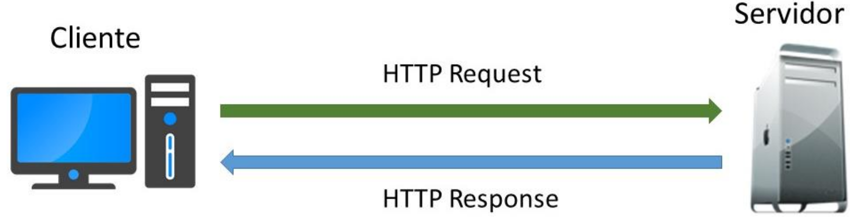


Ilustración 1 – Diagrama petición – respuesta HTTP.

En este trabajo, se pretende mostrar al lector las ventajas que poseen los frameworks a la hora del desarrollo de aplicaciones web. En especial, se va a centrar el estudio en el Framework de alto nivel Django, escrito en Python.

7.2 Definición de Framework Web

Es posible definir un framework web como un conjunto de componentes software que construyen un diseño reutilizable que facilita y agiliza el desarrollo de una aplicación Web robusta. Se puede considerar como una aplicación web incompleta y configurable a la que se le pueden añadir nuevas funcionalidades para conseguir el comportamiento deseado por el grupo de programadores. Estas herramientas y funcionalidades pueden ser: librerías de clases, funciones, scripts, etc. [3]

Como objetivos fundamentales de los frameworks en general, se tienen:

- Acelerar el proceso de desarrollo, ya que el programador de la aplicación no necesita plantearse una estructura global de ella, debido a que el framework le proporciona un esqueleto predefinido sobre el que tendrá que trabajar.
- Reutilizar código ya existente y por lo tanto evitar reescribir un mismo fragmento de código.
- Promover buenas prácticas de desarrollo como el uso de patrones de diseño.
- Es más fácil disponer de herramientas adaptadas al framework concreto para agilizar y facilitar el desarrollo.
- Permitir tener una mayor fiabilidad de las páginas de la aplicación web ya que el Framework ha sido previamente sometido a diferentes pruebas.

Existen diferentes tipos de Frameworks Web orientados a distintos lenguajes de programación, funcionalidades, etc. A continuación, se muestra en una tabla los Frameworks más conocidos, junto con una breve descripción de los mismos:

Framework	Lenguaje	Descripción breve
Spring MVC	Java	Sigue el modelo MVC para aplicaciones de Internet y aplicaciones de seguridad. Ofrece amplia gama de servicios.
JSF	Java	Framework Java que no tiene dependencias externas, posee muchas características.
Struts 2	Java	Framework Java para aplicaciones web con Java EE.
Django	Python	Framework Python que promueve el desarrollo rápido y el diseño limpio.
Pylons	Python	Framework MVC de código abierto que enfatiza la flexibilidad y el desarrollo rápido. Usa el estándar WSGI.
Ruby on Rails	Ruby	Basado en Ruby, orientado al desarrollo de aplicaciones Web.
CakePHP	PHP	Framework MVC para PHP de desarrollo rápido.

Tabla 1 - Tabla de Frameworks de programación Web.

En cuanto a la relación entre un Framework y una aplicación, cabe comentar que una de ellas puede tener uno o más Frameworks. Mientras que un Framework, tal y como se ha descrito antes no se corresponde con una aplicación completa, sino que se puede describir como una base para el desarrollo de éstas.

Aparte de las ventajas de los Frameworks de programación Web que se han comentado, también nos podemos encontrar con algunos inconvenientes. Por ejemplo, la utilización y el conocimiento que se tiene sobre un determinado Framework no implica conocer los demás, ya que cada uno lleva una tecnología e implementación diferente por detrás. Otra posible desventaja, en cuanto al rendimiento, es que un Framework está pensado

generalmente para poder correr independientemente del entorno usado, lo que probablemente para una determinada aplicación no es necesario.

Como conclusión, la incorporación de los Frameworks al desarrollo web ha traído muchas ventajas para los programadores de dicho ámbito y ha sido un gran avance en la evolución de las aplicaciones en la red.

7.3 Patrón Modelo-Vista-Controlador

7.3.1 Definición de Patrón Software

Los patrones software aportan una solución previamente probada y documentada a problemas comúnmente dados en el ámbito de la programación. Tienen una serie de ventajas en su utilización, como pueden ser las siguientes: ahorro de tiempo a la hora de desarrollar un sistema software, establecen un lenguaje común para que distintos desarrolladores puedan comunicarse fácilmente sabiendo en cada momento de que elemento está hablando el contrario y permiten al autor estar seguro de la validez del código implementado. [4]

Para que una determinada solución software sea considerada un patrón debe de ser tanto reutilizable como eficaz y estar perfectamente probada.

Los patrones software están compuestos de los siguientes elementos:

- **Nombre**: Permite identificar al patrón software.
- **Problema**: Describe cuando utilizar el patrón.
- **Solución**: Técnicas a aplicar para llevar a cabo el desarrollo del patrón.
- **Consecuencias**: Efectos positivos y negativos de aplicar el patrón software.
- **Patrones relacionados**: Similitudes con otros patrones de diseño.

Los patrones de diseño se pueden clasificar de la siguiente manera:

- **Patrones de Creación**

  Son los patrones que facilitan la tarea de creación de nuevos objetos, para que se produzca un desacoplamiento entre la implementación y el proceso de creación.

- **Patrones Estructurales**

  Estos patrones software facilitan la modelización del software especificando la manera en que unas determinadas clases se relacionan con otras.

- **Patrones de Comportamiento**

  Gestionan los algoritmos, relaciones y responsabilidades entre los objetos de un sistema software. Permiten definir la comunicación entre los objetos de un sistema, así como el flujo de información entre ellos.

7.3.2 Patrón Modelo – Vista – Controlador

El patrón Modelo – Vista – Controlador es un patrón de diseño que posee un objetivo claro: separar los datos de la aplicación y el estado de la misma, frente al mecanismo usado para representar el modelo al usuario final.

Como un patrón de diseño que es, se basa en la reutilización de código y en la separación de las distintas responsabilidades de la aplicación, con lo que se pretende un mejor mantenimiento del código, así como, ayudar al programador con su tarea de implementación. Como el resto de patrones también tiene como ocupación simplificar la comunicación entre los desarrolladores de un equipo de trabajo, a la hora de desarrollar en este ámbito.

Como nota importante, cabe destacar que este patrón tiene tres componentes principales, que son: el modelo, la vista y el controlador. De tal manera que existirá, por un lado, la encapsulación de los datos, la interfaz o vista por otro y por último la lógica interna de la aplicación o el controlador. Se pasa a comentar estos tres componentes fundamentales:

- **Modelo**

  En el modelo se encuentra el núcleo de la funcionalidad de la aplicación y se encapsula el estado de la misma. Como es en esta capa donde se trabaja con los datos, tendrá los mecanismos necesarios para acceder a la información y actualizar su estado. Esta parte es independiente de las otras dos (controlador y vista), ya que no contiene enlaces a ellas.

- **Vista**

  Esta capa es la encargada de la presentación de los datos del modelo. Para poder realizar correctamente la presentación de los datos, la vista puede acceder al modelo, mediante una referencia que tiene, pero no puede modificar su estado.

- **Controlador**

  El controlador es el encargado de reaccionar ante las peticiones realizadas por el usuario de la aplicación, ejecutando la acción adecuada y creando el modelo pendiente. Es una capa que sirve de enlace entre las vistas y los modelos, ya que su función es servir de enlace entre ellas para implementar las necesidades de la aplicación Web.

Este patrón también posee algunas desventajas: ya que algunos desarrollos son más complejos siguiendo dicha estructura que si no la siguiera, y otra desventaja puede ser el número de partes que será necesario mantener ya que se encuentran distribuidas.

7.4 Introducción a Django

Ilustración 2 - Logo del Framework de Programación Web Django

Django, es un Framework Web de alto nivel escrito en Python, el cual nació en el año 2003, con un equipo de desarrolladores Web del diario *Lawrence Journal-World*, en Lawrence, Kansas (EE. UU.). Dicha unidad de programadores dejó de lado PHP para empezar a utilizar Python para desarrollar sus aplicaciones Web. [5]

El equipo responsable de la producción y mantenimiento de numerosos portales de noticias, *The World Online*, mostraba ciertas restricciones de tiempo para añadir nuevas características a las aplicaciones Web y, además, había una intensa demanda de periodismo. De la necesidad de crear un gran número de entradas correctas en tan poco tiempo, surgió la aparición del framework Web Django, donde lo fueron modificando constantemente, para irlo adaptando a sus necesidades, durante 2 años. Por lo tanto, sigue los principios DRY (*Don't Repeat Yourself*; No Te Repitas) para evitar escribir código duplicado e invertir el menor esfuerzo posible para ahorrar tiempo en el desarrollo de la aplicación.

Este Framework Web, fue nombrado de esta manera en alusión al guitarrista de Jazz gitano *Django Reinhardt* (1910 – 1953) de 1930 hasta principios de los años 50. El cual es considerado uno de los mejores guitarristas de ese estilo.

Como se puede comprobar por lo comentado anteriormente, Django comenzó como un sistema de publicación de noticias, aunque se pueden desarrollar aplicaciones de cualquier tipo. Es posible deducir que Python es utilizado en todas las partes de este Framework, lo que facilita que el desarrollador escriba código ágilmente, haciendo mención al desarrollo rápido y diseño limpio en el que se basa Django.

Más adelante en el 2005, *The World Online* liberó el proyecto bajo licencia BSD, para acabar en 2008, en manos de la recién formada *Django Software foundation*, los cuales se harán cargo de Django. Este proyecto no sería posible sin la existencia de otros proyectos de licencia abierta como Python, SQLite, PostgreSQL, entre otros.

En la actualidad numerosos sitios Web utilizan Django, se nombran alguno de ellos:

I. Lawrence: http://www.ljworld.com/
II. Instagram analytics: https://smartmetrics.co/
III. NASA Science: http://science.nasa.gov/
IV. Prezi: https://prezi.com/
V. Python learning: https://python.web.id/

Como se puede deducir tras analizar los seis casos anteriores, Django es válido para construir cualquier tipo de aplicación sin limitación para un determinado ámbito.

En cuanto a las versiones de Django, se pasa a numerar cada una de ellas junto con la fecha de su lanzamiento:

Versión	Fecha
0.9	16/11/2005
0.91	11/01/2006
0.95	29/07/2006
0.96	23/03/2007
1.0	03/09/2008
1.1	29/07/2009
1.2	17/05/2010
1.3	23/03/2011
1.4	23/03/2012
1.5	26/02/2013
1.6	06/11/2013
1.7	02/09/2014
1.8	01/04/2015
1.9	01/10/2015
1.10	17/01/2017
1.11	01/04/2017

Tabla 2 - Tabla de versiones de Django.

En el sitio web de Django, donde reside toda su documentación oficial [6], se informa de las futuras versiones que tienen pendiente de lanzamiento junto con su fecha de liberación, las cuales se resumen en la siguiente tabla:

Futuras versiones	Fecha
2.0	Diciembre del 2017
2.1	Agosto del 2018
2.2	Abril del 2019
3.0	Diciembre del 2019
3.1	Agosto del 2020
3.2	Abril 2021

Tabla 3 - Tabla de versiones futuras de Django.

Es posible apreciar que este Framework Web está en constante movimiento y actualizándose continuamente, ya que cada nueva versión añade funcionalidades o corrigen determinados elementos de las anteriores lo que van haciendo a este Framework más potente.

7.4.1 Patrón Modelo – Vista – Controlador en Django

En cuanto a la arquitectura de Django, cabe comentar que fue diseñado para que exista un acoplamiento débil entre las distintas partes de la aplicación y una clara separación de las mismas. Por lo que, como se puede deducir, es posible realizar cambios en una parte específica de la aplicación sin afectar a las demás piezas de la misma. Django, por lo tanto, tiene una implementación especial del Modelo – Vista – Controlador.

En las aplicaciones Django, debido a que el Controlador es manejado por el mismo Framework, y la parte más significativa se la llevan los modelos (*Models*), vistas (*Views*) y plantillas (*Templates*), Django es conocido también como un Framework MTV (*Models – Views – Templates*). Se pasa a describir posteriormente este significado.

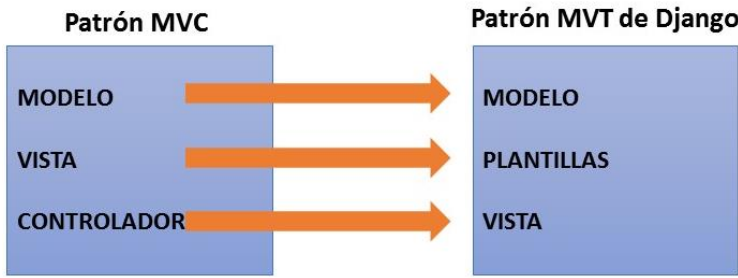


Ilustración 3 - Diagrama correspondencia MVC – MTV.

En Django, se conoce al Controlador como "vista" y a la vista como "plantilla". Ya que en su implementación la vista describe el dato que se muestra al usuario de la aplicación. Por lo que se centran más en qué dato ve y no cómo lo ve. Por lo tanto, lo que decide entonces qué dato ve el usuario es la vista, y cómo lo ve se lo delega a la plantilla. La vista, es la capa de la lógica de negocios, donde se encuentra la inteligencia que accede al modelo y la delega a la plantilla adecuada. Por último, el modelo, no cambia con respecto a la explicación del patrón Modelo – Vista – Controlador convencional, sigue siendo la capa de acceso a la base de datos, que contiene, cómo acceder a ellos, cuál es su comportamiento, como validarlos y las relaciones entre los datos.

Se pasa a mostrar un esquema de la colaboración existente entre los distintos componentes del patrón:

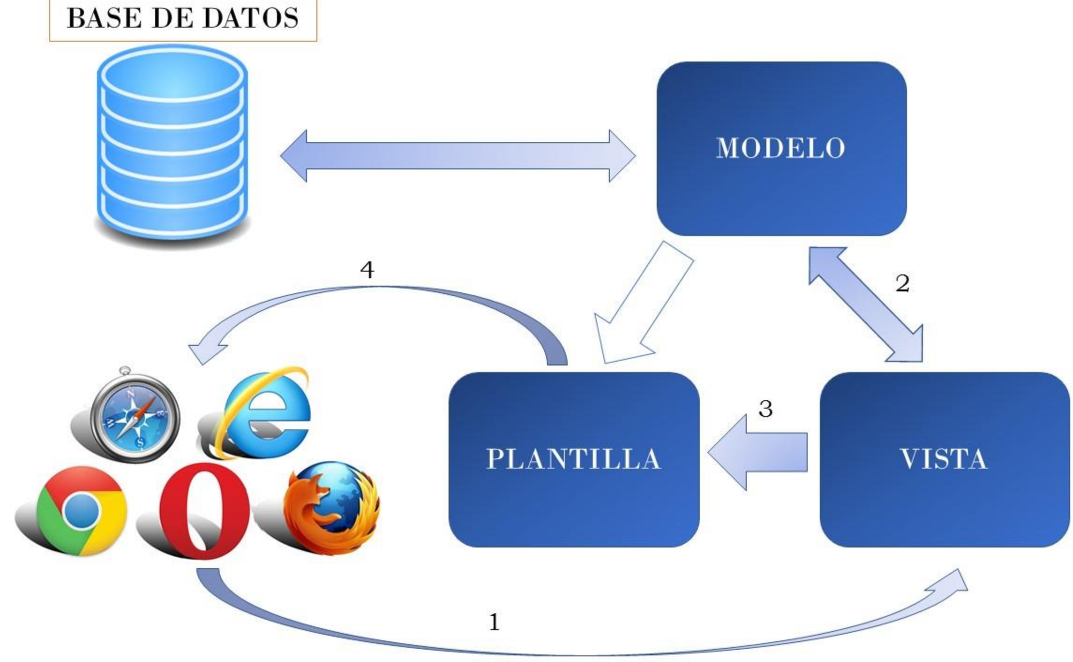


Ilustración 4 - Diagrama colaboración capas MTV.

I. El cliente desde su navegador realiza una solicitud de un servicio web de la aplicación, por lo que se pone en contacto con la capa de la vista.
II. La capa de la vista necesita tener contacto directo con la del modelo ya que es de ella de donde cogerá los datos que solicita el usuario. Por lo que entra en acción la capa del modelo. Ésta última capa, gracias a las consultas a la base de datos devuelve los objetos a la capa de la vista, que ella ya se encargará de filtrar y devolver explícitamente los que el usuario ha solicitado en la petición web.
III. El siguiente paso es que la capa de la vista envíe a la capa de la plantilla los datos requeridos por el usuario, tras previo procesamiento de los mismos. En este paso, ha actuado indirectamente el modelo, ya que es quien proporcionó los datos a la vista.
IV. En último lugar, la capa de la plantilla es la encargada de cómo se deben mostrar los datos al usuario. Por lo que, con los datos recibidos de la vista, los incluye en su respectiva plantilla para que los navegadores puedan procesarlo y el usuario visualice correctamente la información.

Con estos cuatro pasos, se ha pretendido mostrar al lector una visión global de como intervienen las distintas capas de este patrón tras una solicitud del usuario de la aplicación hasta que le devuelve los objetos requeridos.

Estas consideraciones anteriores marcan una diferencia notable entre Django y otros frameworks web como puede ser Ruby on Rails, ya que como se ha comentado ofrece una modificación particular sobre el patrón Modelo – Vista – Controlador.

7.4.2 Requisitos de Django

Para poder realizar una aplicación web, usando Django, es necesario conocer los siguientes puntos:

- Como Django es un Framework para Python, se debe de tener instalado Python en el computador.
- En cuanto a la base de datos subyacente, Django viene con SQLite3 por defecto, pero es posible disponer de otro motor de base de datos como: MySQL, PostgreSQL y Oracle.
- En cuanto al servidor Web donde correrá Django, se pueden diferenciar dos partes:
    - Para la parte de desarrollo de la aplicación, Django aporta un servidor propio.
    - Para la parte de producción, se puede montar sobre otros servidores como Apache, entre otros.

8 Proyecto Django

8.1 Introducción

Antes de proceder a la explicación de las distintas partes de una aplicación Django, en este capítulo se pretende mostrar al lector los diferentes archivos generados automáticamente al crear una nueva aplicación con este Framework. El objetivo principal es que pueda conocerlos para que cuando sean mencionados en la descripción profunda de cada parte sepa a cuál de ellos se refiere.

Como nota importante para el desarrollo de esta documentación cabe mostrar las versiones que se usarán para ello.

- Python 3.4.4
- Django 1.11

En este apartado se van a utilizar los nombres de la aplicación que posteriormente se va a desarrollar para estudiar posteriormente las ventajas y desventajas del Framework que se está utilizando.

8.2 Diferencia entre Proyecto y Aplicación Django

Django permite a los desarrolladores crear proyectos y aplicaciones web. Lo primero que se debe de conocer es la diferencia entre ambos. Una aplicación Web es un sitio Web que hace algo, por ejemplo, un foro, página de noticias, entre otras muchas posibilidades. En cambio, un proyecto es una colección de configuraciones y aplicaciones para un sitio web en particular. La relación entre ambas es que un proyecto puede contener numerosas aplicaciones y una misma aplicación puede estar en distintos proyectos.

8.3 Crear Proyecto en Django

Como se ha comentado en la introducción del trabajo, Django trabaja sobre Python, por lo que será necesario tenerlo instalado previamente donde se vaya a generar la aplicación.

Como nota de recomendación, cabe mencionar que es interesante instalar Django en un entorno virtual, también conocido como *virtualenv*. Gracias a ello es posible aislar la configuración para cada proyecto residente en el computador actual.

Para instalar Django, es necesario ejecutar el siguiente comando dentro del entorno virtual:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines>pip install
django==1.11
Collecting django==1.11
  Downloading Django-1.11-py2.py3-none-any.whl (6.9MB)
    100% |################################| 6.9MB 130kB/s
Collecting pytz (from django==1.11)
  Using cached pytz-2017.2-py2.py3-none-any.whl
Installing collected packages: pytz, django
Successfully installed django-1.11 pytz-2017.2
```

Una vez que se ha instalado correctamente la versión deseada de Django mediante *pip*, se pasa a crear el nuevo proyecto.

```
(GestorCines) C:\Users\Rober\Documents\GestorCines>python
.\Scripts\django-admin.py startproject GestorCines
```

Cuando se ha generado el proyecto correctamente está todo preparado para generar la aplicación:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py startapp appcine
```

Con el proyecto y la aplicación en Django creadas se genera una estructura de archivos y directorios los cuales se pretende describir brevemente cada uno de ellos.

```
src/
      manage.py
      appcine/
            migrations/
                  __init__.py
            __init__.py
            admin.py
            apps.py
            models.py
            tests.py
            views.py
      GestorCines/
            __init__.py
            setttings.py
            urls.py
            wgsi.py
      db.sqlite
```

Pasamos a comentar cada archivo o carpeta que se aprecia en la imagen anterior:

- **src/**
  Esta primera carpeta es el contenedor del proyecto en la cual residirán los archivos y directorios que éste usará. El nombre de este directorio se puede modificar en cualquier momento sin tener que variar ningún parámetro adicional.

- **manage.py**
  manage.py es una línea de comandos mediante la cual se puede interactuar con el proyecto de diferentes maneras como pueden ser: crear aplicaciones, levantar el servidor de desarrollo, posibilidad de llamar a ejecutar migraciones, entre otras. Es parecido a *django-admin* utilizado para crear el proyecto.

- **src/GestorCines**
  Es el paquete Python del proyecto. En este directorio se permite importar cualquier herramienta al proyecto. El nombre de esta carpeta generada sí que es importante para Django ya que será utilizado para importar librerías en él.

- **GestorCines/__init__.py**
  Es un fichero de Python vacío que indica que el directorio en el que se encuentra debe de ser considerado como un paquete de Python. En este fichero inicialmente vacío se puede ejecutar código de inicialización del paquete.

- **GestorCines/settings.py**
  Este archivo contiene la configuración del proyecto Django. No es necesario cambiar las configuraciones si no se necesitan, ya que Django asigna un valor por defecto para cada una de ellas. Más adelante se verá cómo se modifica este archivo para alterar la funcionalidad del sitio web creado.

- **GestorCines/urls.py**
  Se puede describir como una tabla de contenidos del sitio Web en Django. En él se encuentran las declaraciones URL para el proyecto.

- **GestorCines/wsgi.py**
  Es el archivo encargado para la compatibilidad con el servidor Web. Se puede considerar como un punto de entrada para que los servidores Web que son compatibles con WSGI, puedan servir el proyecto.

- **appcine/migrations**
  En este directorio es donde se almacenarán todas las migraciones que se realizarán en el proyecto. Quedan registradas con el objetivo de poder analizar los cambios que se han ido produciendo en la parte del modelo de la aplicación.

- **appcine/admin.py**
  En el archivo *admin.py* es donde se definirán los modelos a registrar para el sitio de administración de Django junto con otra información adicional.

- **appcine/apps.py**
  En este archivo se permite al programador incluir la configuración para la aplicación actual. En él se almacenarán metadatos de la misma.

- **appcine/models.py**
  En este archivo Python será donde quedarán definidos los modelos a utilizar en la aplicación. Se declararán los atributos de cada uno, así como las relaciones entre distintos ellos y otra información adicional.

- **appcine/tests.py**
  En *test.py* se almacenarán las pruebas a realizar para la aplicación donde resida dicho archivo.

- **db.sqlite**
  Se corresponde con la base de datos SQLite3 generada automáticamente en los pasos anteriores. Es posible reemplazar esta base de datos por otra como por ejemplo Postgresql.

8.4 Servidores Web en Django

En cuanto a los servidores donde se alojarán las aplicaciones Web, es posible diferenciar dos apartados:

- **Fase de desarrollo**: Esta fase se da cuando el equipo de desarrolladores está implementando la aplicación. Por lo tanto, se utilizará un servidor escrito en Python que incluye Django con el objetivo de no tenerlo que montar sobre otro y centrarse en el desarrollo de la misma. Este servidor aporta ventajas como: es ligero, se reiniciará cuando detecte cambios en la aplicación, entre otras.
  Para lanzar el servidor de desarrollo bastará con ejecutar el siguiente comando:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
September 03, 2017 - 12:10:26
Django version 1.11, using settings 'GestorCines.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

  Es posible cambiar tanto la dirección IP como el puerto del servidor de desarrollo si el programador lo desea. Para ello bastará con indicarlo en la llamada:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py runserver 8080
```

- **Fase de producción**: Para esta última etapa, Django aporta distintas posibilidades sobre los servidores web a utilizar ya que soporta la especificación WSCGI. Se puede lanzar sobre Apache, SCGI, entre otros.

9 Capa del Modelo en Django

9.1 Introducción

Antes de definir la capa del modelo en Django, cabe comentar que las aplicaciones web modernas tienen una lógica, que muy a menudo está relacionada con una base de datos. Estas aplicaciones, se conectan con el servidor de base de datos correspondiente y recupera los datos que estime oportunos para mostrarlos de una forma amigable para el usuario. Dicho usuario puede realizar modificaciones de esos datos, según el objetivo principal de la aplicación.

En cuanto a la capa del modelo de Django, cabe mencionar que es una de las capas del patrón Modelo – Vista – Controlador, que para este Framework se conoce como *Models – Templates - Views*, como se ha comentado en la introducción, la cual, es la capa de acceso a la base de datos de la aplicación.

En esta capa de Django, es donde se sitúa toda la información sobre los datos que tendrá la aplicación web. Algunos ejemplos de la información que contiene pueden ser: cómo se validan los datos, cómo se acceden a ellos, cual es el comportamiento que tienen y las relaciones que habrá entre los mismos, además de muchas otras.

A continuación, se pasa a comentar cada parte de esta capa.

9.2 Modelos

9.2.1 Modelos en Django

Un modelo es la fuente de información sobre los datos que habrá en la aplicación. Contiene los campos, métodos, metadatos y comportamientos de los datos que almacena. Normalmente, se corresponde con una tabla en la base de datos.

En el caso de Django, un modelo se define en el lenguaje Python y tienen las siguientes características. La primera es que cada modelo creado es una clase Python que, a su vez, es una subclase de *django.db.models.Model*. La clase *Model*, contiene todas las herramientas para hacer que los objetos que se han creado puedan interactuar con la base de datos de la aplicación. Cada modelo, está compuesto de una serie de atributos, que representan un campo de la tabla correspondiente. Por lo tanto, Django utiliza los modelos para ejecutar código SQL por detrás y devolver las estructuras de datos útiles en Python que representarán las filas de las tablas de la base de datos.

Django ofrece automáticamente una API Python de alto nivel que proporciona acceso a las bases de datos. El objetivo de esta API es ejecutar código SQL y devolver estructuras de datos Python que representan filas de las tablas en la base de datos. De tal manera, este proceso quedará transparente para el programador y será independiente del motor de base de datos que se utilice. También pretende ayudar al desarrollador para que pueda

representar conceptos de alto nivel, ya que algunos de ellos no pueden ser manejados con el lenguaje SQL de forma sencilla y también, le permite un ahorro de líneas de código. De esta manera, mediante su ORM (*Object-Relational mapping*), Django realizará el mapeo de los objetos Python con la base de datos.

En cuanto al archivo Python donde se trabajará esta capa es "*models.py*". En este archivo se encuentra la descripción de las tablas de la base de datos, pero definidas como clases Python. Usando dichas clases, se pueden crear, eliminar y actualizar objetos, entre otras operaciones, que se corresponderán con entradas en la tabla correspondiente. Por lo que es en este fichero donde se crearán las clases que se necesiten para la aplicación web.

Uno de los objetivos principales de esta importante capa del Framework, es abstraer a los programadores el motor de base de datos que tendrán por detrás de tal manera que el código incluido en esta capa no tenga un fuerte acoplamiento con el tipo de base de datos subyacente. De esta manera, se programará el modelo sin conocer necesariamente el motor que tendrá la aplicación en desarrollo.

9.2.2 Campos

Como se ha comentado en el apartado anterior, un modelo está compuesto de numerosos campos, que son las columnas que contendrá la correspondiente tabla. Se puede considerar que un campo es la pieza fundamental en distintas API de Django, sobre todo en modelos y *Querysets*.

A continuación, se pasa a ver cómo tratar los campos dentro de este Framework. Lo primero que se debe de *conocer* es que los campos se especifican mediante atributos de clase y se corresponden con una columna de la tabla de la base de datos en particular. Cada uno de ellos debe de ser una instancia de la clase abstracta "*Field"*. Al igual que en una base de datos, se debe de indicar el tipo de dato que será dicho campo, es decir, si es de tipo *VARCHAR*, *DATETIME*, entre otros. También es necesario definir los requisitos de validación de los campos, para determinar que los datos tengan el formato que el programador desee. Además, es posible definir el formato HTML que tendrá dentro del formulario en el que se encuentre la solicitud de dicho campo.

9.2.2.1 Opciones de campos

Cada campo del modelo de datos toma un determinado conjunto de argumentos específicos de dicho campo, que usará la base de datos para almacenarlo con las particularidades que el programador desee. Los argumentos que se pasan a comentar a continuación son válidos para todo tipo de campo y son opcionales:

Opciones de campos	Descripción
Field.null	Permite almacenar valores vacíos, como NULL en la tabla si se encuentra activada esta opción
Field.blank	Si se activa, se permite dejar en blanco el valor el valor de dicho campo en el formulario, si no, será un campo obligatorio.

Field.choices	Permite establecer un diccionario de elementos a un objeto para que los valores solo sean los incluidos en ese diccionario.
Field.db_column	Permite determinar el nombre de la columna de la base de datos donde Django almacenará dicho campo.
Field.db_index	Admite la indexación del campo en las búsquedas de Django.
Field.db_tablespace	El nombre del espacio de la tabla de la base de datos que se utilizará para el índice de este campo, si está indexado.
Field.default	Indica el valor por defecto para un campo del formulario.
Field.editable	Si es Falso, el campo no se mostrará en el sitio de Administración ni en ningún otro ModelForm.
Field.error_messages	Permite anular los mensajes predeterminados de error que puede generar el campo.
Field.help_text	Se puede añadir más texto de ayuda adicional que se mostrará con el formulario.
Field.primary_key	Si esta opción está a *True*, ese campo es la clave primaria del modelo.
Field.unique	Si esta opción se activa, el campo deberá de ser único en la tabla. Se activa tanto en la base de datos como en la validación del modelo.
Field.unique_for_date	Si se establece esta opción con el nombre de un *DateField* o *DateTimeField,* ese campo será único.
Field.unique_for_month	Igual que el anterior, pero requiere que el campo sea único con respecto al mes.
Field.unique_for_year	Igual que el anterior, pero con respecto al año.
Field.verbose_name	Nombre legible para el campo. Si no se proporciona, Django lo crea automáticamente.
Field.validators	Lista de validadores para ejecutar para ese campo en cuestión.

Tabla 4 - Tabla de las opciones de campos.

9.2.2.2 Tipos de campos

Otro aspecto muy importante en cuanto a los campos es el tipo de los mismos, ya que servirá para determinar el tipo de columna de la tabla de la base de datos en cuestión, entre otra información que se pasa a comentar en este apartado.

Este Framework trae por defecto una lista de diferentes tipos de campos predefinidos que se pasan a comentar a continuación:

Tipo de campo	Descripción breve
AutoField	Campo el cual se incrementa automáticamente de acuerdo con los identificadores disponibles.
BigAutoField	Entero de 64-bit, parecido al anterior. Desde 1 a 9223372036854775807. Nuevo en Django 1.10.
BigIntegerField	Almacena valores de tipo entero de valores muy grandes.
BinaryField	Campo que permite el almacenamiento de datos binarios sin procesar.
BooleanField	Tipo booleano, valores posibles: True y False.

CharField	Campo para almacenar pequeñas cadenas de texto.
DateField	Almacena campos de fecha.
DateTimeField	Mismas opciones que el anterior. Pero guarda fecha y hora.
DecimalField	Número decimal de precisión fija.
DurationField	Campo que permite almacenar periodos de tiempo
EmailField	Campo de tipo e-mail. Comprueba que la dirección de correo sea correcta.
FileField	Campo de carga de ficheros.
FilePathField	Campo para mostrar los archivos accesibles de una carpeta siguiendo unas restricciones.
FloatField	Número de punto flotante.
ImageField	Campo idéntico a *FileField* pero comprueba que sea una imagen.
IntegerField	Recoge números enteros.
NullBooleanField	Campo igual que *BooleanField*, pero acepta nulos.
PositiveIntegerField	Campo de tipo número entero positivo.
PositiveSmallIntegerField	Como el anterior, pero como máximo valor permitido tiene: 65535
SlugField	Etiqueta que contiene letras, números y guiones. Su uso es extendido en la definición de URLs.
SmallIntegerField	Campo de tipo número entero comprendido entre: -32768 y 32768.
TextField	Campo de tipo texto de longitud máxima ilimitada.
TimeField	Campo de tipo hora. Mismas condiciones que *DateField* y que *DateTimeField*.
URLField	Campo que almacena una dirección HTML y valida que sea correcta. Es como un *CharField* para una URL.
UUIDField	Usando la clase UUID de Python, permite guardar Identificadores Universales Únicos

Tabla 5 - Tabla de los tipos de campos.

Con estos tipos de campos, no se describe únicamente el tipo de la columna de la base de datos, sino que también se establece el widget HTML predeterminado que tendrá dicho campo en el formulario y unos requisitos mínimos de validación para los formularios del modelo.

Es posible disponer de tipos de campos personalizados en Django. Si se quiere utilizar un tipo que no está definido, se puede escribir fácilmente creando una subclase de *Field*. La definición de un campo personalizado puede ayudar a conseguir resultados más específicos y que se ajusten más a los requerimientos del programador.

9.2.2.3 Claves primarias automáticas

Django agrega a cada modelo que se crea en la aplicación una clave primaria única que se incrementa automáticamente para identificar de forma única un determinado objeto. El código que Django incrusta por detrás es el siguiente:

```
id = models.AutoField(primary_key=True)
```

Django lo realiza de forma automática a no ser que se desee introducir otra clave primaria distinta. Para ello se debe de indicar *primary_key=True* en el campo que se quiera convertir en clave primaria. Si esto se cumple, Django no añadirá automáticamente la columna *id.*

9.2.2.4 Nombre de campo Verbose

Todos los tipos de campos toman un argumento opcional en la primera posición, un "nombre detallado", que es una cadena de texto con el cual se describe al campo. Si este campo opcional no se especifica, Django lo crea automáticamente con el nombre de atributos del campo, donde los "_", lo toma como espacios en blanco. Esto sirve para tener un nombre entendible por las personas que utilizan la aplicación, ya que será dicho nombre el que aparezca en los formularios, a priori.

Para los tipos *ForeignKey*, *ManyToManyField* y *OneToOneField*, hay una excepción y es que deben de tener como primer argumento la clase del modelo con el que se relaciona.

Por ejemplo:

```
class Cliente(models.Model):
      user = models.OneToOneField(User, on_delete=models.CASCADE)
      fecha_nacimiento = models.DateField('Fecha de nacimiento',
auto_now = False, auto_now_add = False, blank = True, null = True)
      numero_telefono = models.IntegerField('Número de teléfono',
primary_key=True)
```

9.2.2.5 Relaciones entre campos

Las tablas de una base de datos pueden encontrarse relacionadas y estas relaciones no son siempre iguales, ya que dependiendo de la correlación que mantengan dos tablas se pueden obtener comportamientos diferentes.

Django, por lo tanto, ofrece diferentes formas para definir los diferentes tipos de relaciones existentes. Las más comunes son:

1. **Relación Muchos – a – Muchos (*ManyToManyField)***

   Con la clase *ManyToManyField* se puede definir una relación Muchos – a – Muchos, usándola como cualquier otro campo e incluyéndola como un atributo de clase del modelo.

   Es necesario pasarle como parámetro la clase a la cual se relaciona el modelo.

   Django realiza por detrás una tabla intermedia para representar la relación Muchos – a – Muchos. No es necesario que el campo *ManyToManyField* esté en los dos modelos, con que se encuentre en uno de ellos es correcto.

```
class Pelicula(models.Model):
      actores = models.ManyToManyField(Actor)
```

2. **Relación Uno – a – Uno (*OneToOneField*)**

Haciendo uso de esta clase, *OneToOneField,* es posible definir una relación Uno – a – Uno, usándola como cualquier otro campo e incluyéndola como un atributo de clase del modelo.

Como en el caso anterior, es necesario pasarle como parámetro la clase a la cual se relaciona el modelo.

Este tipo de relación es muy similar a la del siguiente punto, *ForeingKey*, con un parámetro que sea único.

```
class Cliente(models.Model):
      user = models.OneToOneField(User, on_delete=models.CASCADE)
```

3. **Relación Muchos – a – Uno (*Many-to-one*)**

Este tipo de relación es también conocida como, relación foránea. Para definir una relación muchos – a – uno, se utiliza la clase *django.db.models.ForeingKey*, la cual se usa como cualquier tipo de campo metiéndola como un atributo de clase del modelo.

Es necesario pasarle a *Foreing Key* como argumento, la clase que está relacionando el modelo.

Cada vez que se crea una relación de clave foránea, Django crea un índice en la base de datos de forma automática. Si no se desea que cree dicho índice, se debe de poner a *False* la opción del campo *db_index*.

```
class Comentario(models.Model):
      pelicula = models.ForeignKey(Pelicula)
```

9.2.2.6 Los modelos a través de archivos

Django nos permite relacionar un modelo con otro de otra aplicación de forma muy sencilla. Para ello, se debe importar el modelo que se quiera relacionar, para poder referirse al modelo cuando sea necesario sin tener que realizar nada adicional.

```
from otra_app.models import modelo2
class Cine(models.Model):
      campo_aux = models.ForeingKey(modelo2)
```

9.2.2.7 Restricciones a los nombres de los campos

Hay una serie de restricciones al asignar nombres a los campos en Django, éstas limitaciones son:

- Como se trabaja sobre Python, no se puede utilizar como nombre de campo una palabra reservada para este lenguaje, ya que esto produciría un error de sintaxis en de Python.
  Por ejemplo, no se permite poner de nombre a un campo: "*False*", "*from*", "*global*", entre muchas otras.

El código que se muestra a continuación sería erróneo:

```
class Ejemplo(models.Model):
    False  =  models.char.BooleanField()  #  'False'  es  una  palabra
reservada
```

- Otra limitación que impone Django es que no se debe incluir dos o más guiones bajos consecutivos ("__") en una fila. Esto se debe a la manera que trabaja la sintaxis de las consultas de Django, la cual se explicará más adelante en este Trabajo de Fin de Grado.
  El fragmento de código que se muestra a continuación es incorrecto:

```
class Ejemplo(models.Model):
    num__dni = models.IntegerField() # 'num__dni' tiene dos guiones
bajos
```

Estas limitaciones que impone Django se pueden solventar fácilmente debido a que el nombre del campo no tiene que ser el nombre de la columna de la base de datos, como se ha comentado en la tabla 4, por la opción de campo *db_column.*

Por el contrario, Django permite asignar como nombre de campo palabras reservadas de SQL, tales como "*select*" o "*join*", ya que utiliza la sintaxis de comillas de su motor de base de datos en particular por lo que no infiere en el resultado.

9.3 Opciones Meta

Con el objetivo de tener un mayor control de los datos, se pueden definir metadatos propios para cada modelo de Django, para ello, se tiene que agregar una clase interna *Meta* al modelo. Esta característica permite influenciar en cómo se comportan o como se muestran los datos del mismo. También, se puede establecer el nombre de visualización, tanto en singular como en plural para el modelo, el orden de visualización y el nombre de la tabla de la base de datos, entre otros.

Establecer metadatos para los modelos se puede realizar de manera opcional, por lo que no es obligatorio definir una clase *Meta* para cada modelo de la aplicación.

Django aporta distintas opciones *Meta*, como, por ejemplo: *abstract*, *db_table*, *ordering*, *permissions*, *proxy*, *verbose_name*, *select_on_save*, entre muchas otras. Como novedad en Django 1.11, añade *options.indexes* para poder incluir una lista de los índices que se van a definir en el modelo.

Se muestra un ejemplo de un modelo con su clase interna *Meta*:

```
class Cartelera(models.Model):
      identificador        =       models.CharField(max_length=100,
primary_key=True)
      nombre_pelicula = models.ForeignKey(Pelicula)
      nombre_sala = models.ForeignKey(Sala)
      fecha_hora = models.DateTimeField(auto_now = False, auto_now_add
= False)

      def __str__(self):
            return self.identificador

      class Meta:
            verbose_name_plural="Listado Cartelera"
            verbose_name = "Pelicula"
            ordering = ['nombre_pelicula']
```

9.4 Manager de un modelo

Dentro de los atributos del modelo, es necesario resaltar uno debido a su alta importancia, el *Manager*. *Manager* es la interfaz mediante la que se proporcionan consultas de base de datos a los modelos de Django y se usa para recuperar instancias de la misma.

En toda aplicación Django tiene que haber uno para cada modelo definido en dicha aplicación. No es necesario definir un *Manager* con un nombre determinado, ya que, si no Django le pone el nombre de *objects* por defecto. Al igual que se puede modificar el nombre del *Manager*, también es posible personalizar extendiendo de la clase base *Manager* e instanciándolo en el modelo correspondiente. De tal manera se podrán generar nuevos métodos que podrá usar posteriormente a nivel de tabla.

En cuanto a la accesibilidad de este atributo, hay que comentar que solamente son accesibles a través de las clases del modelo, no de las instancias de estos.

Por ejemplo, un uso del *Manager* de un modelo es cuando se desea realizar una consulta a la base de datos:

```
Cartelera.objects.filter(nombre_pelicula = titulo)
```

Donde *objects* es el nombre del *Manager* de la clase "*Cartelera*", ya que no se ha modificado su nombre por defecto.

9.5 Métodos de los modelos

Django permite que se puedan definir métodos personalizados en un modelo con la finalidad de añadir funcionalidades a los objetos a nivel de fila. A diferencia del método *Manager*, que trabajan a nivel de tabla, los métodos de los modelos actúan en un caso de ese propio modelo en particular.

Para cada modelo, hay una serie de métodos predefinidos. Se pasan a comentar los más importantes:

- ***__str__()***
  Este método, es para Python 3 o superiores (para Python 2 es *__unicode__()*) y es un método que devuelve una representación Unicode de cualquier objeto, por lo que se usará cada vez que una instancia de un modelo tenga que ser coaccionada. Se muestra como una cadena de texto simple.
  Como nota importante, cabe mencionar que debe obligatoriamente devolver un *string*. Se pueden agregar las cadenas y opciones que el programador estime oportuno siempre y cuando se cumpla la condición anterior. Normalmente se suele poner un *string* identificativo de un objeto.
  NOTA: Un objeto Unicode es una cadena que puede manejar muchos tipos distintos de caracteres.
- ***get_absolute_url()***
  Este método se usa en la interface de administración de Django y cuando se necesita saber la dirección URL de un objeto, ya que este método ayuda a Django a como calcular dicha URL para un objeto de forma exclusiva.

Estos son los métodos más importantes, aunque también existen otros como *__hash__()* y *__eq__()*.

Los métodos predefinidos de Django se pueden modificar de tal manera que el desarrollador pueda añadir o quitar alguna funcionalidad y así transformar su comportamiento ajustándose a los requisitos de la aplicación. Como nota importante de este caso, es que es necesario llamar al método que se está modificando de la superclase, para garantizar la correcta ejecución de dicho método.

Como se ha podido observar de una manera muy clara en este apartado, se puede agregar cualquier tipo de comportamiento al modelo de datos de Django. Por lo que dicho modelo, no solo describe la configuración de las tablas de la base de datos, si no que intenta ir un paso más allá.

9.6 Herencia de modelos

Django permite realizar herencia entre distintos modelos de una aplicación de forma similar a la herencia de las clases corrientes en Python. Como una norma de codificación, la clase base debe de ser una subclase de *django.db.models.Model.*

A continuación, se pasa a comentar los distintos tipos de herencia que ofrece Django:

9.6.1 Clases base abstractas

Este tipo de herencia se da cuando se requiere tener datos en común entre varios modelos de una misma aplicación. En esta herencia existe una clase padre, que será abstracta ya que no creará una tabla en la base de datos, y distintas clases hijas que heredarán de la ella. De esta forma no será necesario reescribir el mismo código en cada subclase.

Como nota importante, hay que resaltar que la clase padre no se puede utilizar aisladamente ya que es una clase abstracta, como se ha comentado en el párrafo anterior.

9.6.2 Herencia multi-tablas

Este tipo de herencia se debe de utilizar cuando se desee que cada modelo tenga su propia tabla en la base de datos de la aplicación, es decir, cuando cada clase de la jerarquía formada es un modelo en sí mismo. A diferencia del tipo de herencia anterior, se puede crear y consultar cada modelo de forma individual.

Todos los campos de la superclase estarán disponibles en el objeto de su respectiva subclase, aunque en la base de datos tengan tablas distintas.

Por ejemplo, es conveniente usar este tipo de herencia cuando se está subclasificando un modelo existente en otra aplicación.

9.6.3 Modelo Proxy

Se debe de usar este tipo de herencia cuando se quiera modificar el comportamiento de un modelo, pero a nivel de Python. Esta herencia no requiere cambiar los campos de estos modelos.

Un ejemplo de un cambio de comportamiento puede ser añadir un nuevo método al modelo.

El comportamiento sería crear un *Proxy* a modelo del que se quiera cambiar el comportamiento y realizar lo que el programador estime oportuno, de tal forma que se almacena sin modificar el original. Esta implementación se corresponde con el patrón de diseño *Proxy*.

9.7 Paquetes de modelos

Una gran ventaja que nos ofrece este Framework de programación es que permite estructurar en diferentes archivos los distintos grupos de modelos que tenga una aplicación.

En Capítulo 8 se comentó, que cuando se crea una nueva aplicación Django, se genera el archivo *models.py*, que es donde se definen los distintos modelos. Para aplicaciones grandes, conviene eliminar dicho archivo y generarse los que se estimen oportuno dentro de un nuevo directorio en el proyecto, */models* que contendrá el archivo *__init__.py* y los ficheros para almacenar los modelos. De esta manera quedará mucho más limpio el código de la aplicación, ya que el archivo original *models.py* sería muy grande.

9.8 Usar los modelos

Cuando se ha terminado de escribir los modelos de la aplicación, es necesario comunicar a Django que cree las tablas correspondientes en la base de datos que tiene configurada. Para ello, hay que modificar el archivo de configuración "*settings.py*", donde en la variable *INSTALLED_APPS*, se añade la ruta Python completa de la aplicación, la cual contendrá el archivo models.py donde se han definido los modelos que usará.

Inicialmente, como no se ha comunicado nada a la base de datos, es necesario ejecutar el siguiente comando para dicha sincronización.

```
$> python manage.py syncdb
```

Posteriormente, sí que están creadas las tablas en la base de datos, y mediante la API que facilita Django para el manejo de los objetos, es posible realizar distintas operaciones sobre ellos. Dichas operaciones sobre los modelos y objetos de la aplicación se mostrarán en el siguiente apartado.

9.9 Consultas sobre los modelos

Como se ha comentado en el apartado anterior, Django ofrece a los programadores una API Python que proporciona acceso a las bases de datos para que éstos no necesiten escribir código en SQL si no se requiere. Esta API ofrece una abstracción, la cual, permite interactuar con objetos, ya sea creándolos, eliminándolos o haciendo algún tipo de modificación o selección.

En este punto del trabajo se pasa a comentar como realizar distintas operaciones sobre los modelos a través de la API.

De este apartado cabe comentar que el código que se va a utilizar se va a implementar en la siguiente capa, la capa de la vista, que será donde Django trabaje con los datos almacenados en el modelo. Se explica en este capítulo el manejo de dichas facilidades ya que es el modelo con el que se trabaja.

9.9.1 Crear y actualizar objetos

Como se ha comentado previamente, Django representa una clase del modelo de la aplicación como una tabla en la base de datos, y para representar un registro en dicha tabla, se utiliza una instancia de dicha clase.

Comenzando con la creación de objetos de una clase, Django ofrece dos maneras distintas de llevar a cabo dicha acción:

- La primera de ellas es instanciando el objeto pasándole los parámetros requeridos, que serán los valores para cada campo, y después llamar al método *save()* del modelo. Este método lo guarda en la base de datos de la aplicación.

```
comentario = comentario(comentario = "Comentario de prueba", película
= pelicula1, usuario = usuario1)
comentario.save()
```

- Otra forma es mediante el método *create()* el cual lo crea y lo almacena en la base de datos en un mismo paso.

```
Comentario.objects.create(comentario = "Comentario de prueba",
película = pelicula1, usuario = usuario1)
```

De esta manera tan sencilla es posible crear instancias de objetos y por lo tanto agregar nuevas filas a la tabla correspondiente de la base de datos, pero lo que realiza Django por detrás es más complejo. Django, cuando se produce una llamada al método *save()*, accede a la tabla, en la que ejecutará un *SQL INSERT* del objeto correspondiente. Si no se realiza la llamada a dicho método explícitamente, Django no lo insertará en la base de datos. Cabe recordar que la llamada al método *create()*, realiza la inserción sin necesidad de llamar a dicho método. Como una observación de este punto, es que, si el modelo tiene una clave primaria automática, Django calcula el valor de ese registro y lo almacena, quedando invisible al programador.

Por ejemplo, el resultado de la llamada anterior (para cualquiera de las dos opciones) sería en términos de SQL:

```
INSERT INTO appcine_comentario
(id, comentario, película_id, usuario_id)
VALUES (1, 'Comentario de prueba', película1, usuario1)
```

Con esto, quedaría el objeto incluido en la tabla correspondiente.

Si posteriormente se vuelve a producir una llamada al método *save()* para esa instancia, se realizará una operación de actualización de los campos, es decir, ejecutará una operación *SQL UPDATE*. Cabe comentar que Django por defecto, actualizará todos los campos para ese registro, es decir, actualizará todas las columnas de una fila, como se puede apreciar en el siguiente ejemplo:

```
UPDATE appcine_comentario SET
id = 1,
comentario = 'comentario nuevo',
pelicula_id = pelicula_id,
usuario_id = usuario_id
WHERE id = 1;
```

Al igual que hay un método para crear objetos, también se dispone de un método *update()* para actualizarlos, al que se puede incluir cualquier consulta, por lo que permitirá actualizar varios objetos a la vez, que serán los que retorne dicha *query*. Este método, devuelve un *integer* que representa el número de registros actualizados, es decir, el número de filas de la tabla actualizadas. La forma de hacer dichas consultas se verá en el siguiente capítulo del trabajo.

Si se intenta tanto crear un registro como actualizarlo, con un valor de tipo incorrecto, Django mostrará una excepción al programador.

9.9.2 Seleccionar objetos

Aparte de crear y actualizar objetos, un aspecto muy importante es recuperar un determinado conjunto de ellos. Esto puede resultar de gran interés para casi todo tipo de aplicaciones, ya que en ellas el usuario solicitará un determinado subconjunto específico de objetos que será necesario devolverle para que interactúe con ellos.

Para recuperar objetos de un modelo en Django, se tiene que construir un *Queryset* usando el *Manager* de su modelo. Un *Queryset* representa un conjunto de filas a partir de una tabla de la base de datos. Estos *Querysets* pueden llevar filtros con el objetivo de poder recuperar los objetos que cumplan unas determinadas condiciones que se le impondrán.

El *Manager* en este caso, actúa como una clase raíz la cual, tendrá todos los objetos de esa tabla, a partir de la cual se podrá aplicar filtros y seleccionar los datos deseados. Por lo que, para recuperar objetos, se debe de llamar a los métodos del *Manager* junto con el modelo del que se quieran seleccionar datos.

9.9.2.1 Seleccionar todos los objetos

Para recuperar todos los objetos de la tabla sin aplicar ningún tipo de filtro, es necesario llamar al método *all()* del *Manager*. Este método retorna un *Queryset* de todas las filas de la base de datos.

```
lista_sesiones = Cartelera.object.all()
```

9.9.2.2 Seleccionar objetos específicos aplicando filtros

Como se ha comentado en la introducción de este punto del trabajo, no solo es posible extraer todas las instancias de un modelo, si no que se puede recuperar un subconjunto de ellos que satisfacen una serie de requisitos. Esto es posible gracias al métodos como por ejemplo *filter()* y *exclude()*. La principal diferencia de estos dos métodos, es que *filter()*, devuelve un nuevo *Queryset* que tiene los objetos que coinciden con los parámetros de búsqueda dados, y por el contrario, *exclude()*, se queda con los datos que no cumplen dicho estado.

En términos de SQL, si usamos dichos métodos, sería una consulta del tipo:

```
SELECT …
FROM …
WHERE …
```

Django aporta distintos métodos para retornar distintos resultados como pueden ser: *exclude(), distinct(), order_by(), raw() y value_list()*, entre otras muchas opciones.

Estos métodos pueden ser encadenados para lograr el resultado tan preciso como requiera el problema. Cada resultado de la aplicación de uno de los métodos comentados anteriormente devuelve un nuevo *Queryset*, que no tiene que ver con el *Queryset* anterior, y pueden ser almacenados y reutilizados. En términos de SQL, equivaldría a ir añadiendo nuevas clausulas con el operador "*AND*".

```
lista_entradas_futuras = Entrada.objects.filter(id_cliente = id_usuario).filter(fechaHora__gte=datetime.datetime.now()).order_by('id_entrada')
```

Es posible elegir la opción que queremos que tenga la cláusula *SQL WHERE* que se han mencionado con anterioridad, mediante búsquedas como *exact, iexact, startswith* y *endswith* entre muchas otras. Con estas condiciones podemos indicar que, el resultado contenga exactamente una descripción o que tenga una determinada palabra, que comience por un *string* específico, etc.

Aparte de encadenar condiciones con el operador "*AND*", es posible realizar una consulta algo más completa utilizando el operador "*OR*" o el valor "*NOT*". Para este tipo de consultas se deben de usar objetos *django.db.models.Q,* ya que si no por defecto no se pueden ejecutar dichas operaciones. Un objeto Q, encapsula un conjunto de argumentos de palabras clave, que son patrones de búsqueda. Para cada *Queryset*, puede incorporar varios objetos Q como argumento.

```
from django.db.models import Q
q = request.GET.get('q', '')
querys = (Q(nombre_pelicula__titulo__icontains=q) |
Q(nombre_pelicula__director__icontains=q))
cartelera = Cartelera.objects.filter(querys)
```

Si se conoce que una determinada condición la cumple solo un elemento, es posible recuperarlo haciendo uso del método *get()* en un *Manager*. *get(),* por lo tanto, devolverá un solo objeto, por lo que si la condición que se le pasa como argumento a dicho método, la cumplen varios objetos o ninguno, Django mostrará una excepción.

Se muestra una ejemplo de un uso de *get()*.

```
lista_cartelera = Cartelera.objects.get(identificador =
identificador)
```

Otra opción que aporta Django, al igual que ocurre en el caso de SQL, es la de limitar el número de resultados de un *Queryset* haciendo uso de la sintaxis de Python de rebanado de listas. La cláusula equivalente a lo anterior en SQL es *LIMIT* y *OFFSET*.

```
def vista_pelicula(request, titulo):
      item_peli = get_object_or_404(Pelicula,titulo = titulo)
      lista_comentarios = Comentario.objects.filter(pelicula = item_peli).order_by('-fecha_comentario')[:5]
```

Si se quiere realizar una búsqueda de un objeto por el valor de la clave primaria (*pk,* del inglés *Primary Key*) del modelo correspondiente, es posible referirse a esta cuando se aplica un determinado filtro, mediante la palabra reservada "*pk*".

```
comentario_a_eliminar = Comentario.objects.get(pk=id)
```

Si en un *Queryset* es necesario realizar una comprobación con un campo de otro modelo relacionado, es posible en Django. Para ello hay poner el nombre del campo de la relación seguido del determinado campo deseado de ese modelo. Ambos separados por "__". Se muestra un ejemplo a continuación:

```
Q(nombre_pelicula__titulo__icontains=q)
```

9.9.2.3 Incrustar SQL sin procesar

Si las opciones que ofrece la API de consultas de Django no valen para determinar un resultado específico que desee el programador, puede incrustar SQL en crudo que Django ejecutará directamente. Esto es posible utilizando el método *raw()* del *Manager*, *Manager.raw()*. Este método devolverá instancias del modelo, al igual que en los casos anteriores de esta documentación y podrá ser iterado de la misma manera.

Un ejemplo de uso de SQL es el siguiente:

```
Cartelera.objects.raw('SELECT * FROM appcine_cartelera GROUP BY
nombre_pelicula_id')
```

9.9.2.4 Eliminar objetos

Al igual que se crean y actualizan objetos, como se ha comentado en los apartados anteriores, Django también nos ofrece la posibilidad de eliminarlos. Para ello, existe el método *delete()*, que no devuelve ningún valor y elimina directamente el objeto individual. Si por el contrario, se desea eliminar un conjunto específico de objetos, se debe de llamar al método *delete()* en el resultado de cualquier consulta, de tal manera que elimina todos sus miembros. Este método es el único método *Queryset* que no está expuesto a un *Manager* en sí mismo.

En términos del lenguaje SQL, esto que se acaba de comentar, equivale al código siguiente con el comportamiento *ON DELETE CASCADE*:

```
DELETE …
FROM …
```

Esto quiere decir que si se elimina un objeto con una clave a la que hacen referencia claves externas de objetos existentes en otras tablas, todos esos objetos que contienen dichas claves también se eliminarán. Es comportamiento puede ser modificado si se ve necesario.

9.9.3 Cacheo de consultas

Una característica que es necesario conocer, es cómo se evalúan los distintos *Queryset* que se encuentran en las vistas. Django no obtiene los resultados de las diferentes consultas hasta que éstas no son llamadas. Cuando se produce esta última casuística, el *Queryset* se evalúa accediendo a la tabla correspondiente de la base de datos, devolviendo el resultado con los datos que cumplen esas condiciones.

Con el fin de optimizar al máximo el tiempo que tarda en evaluarse un *Queryset*, así como, las operaciones contra la base de datos, Django aporta a cada *Queryset* una cache propia. En un primer momento, la caché se encuentra vacía, pero cuando es evaluada la consulta, se almacenan los resultados en ella con el fin de que cuando vuelva a ser llamado no tenga que realizar la misma operación sobre la base de datos.

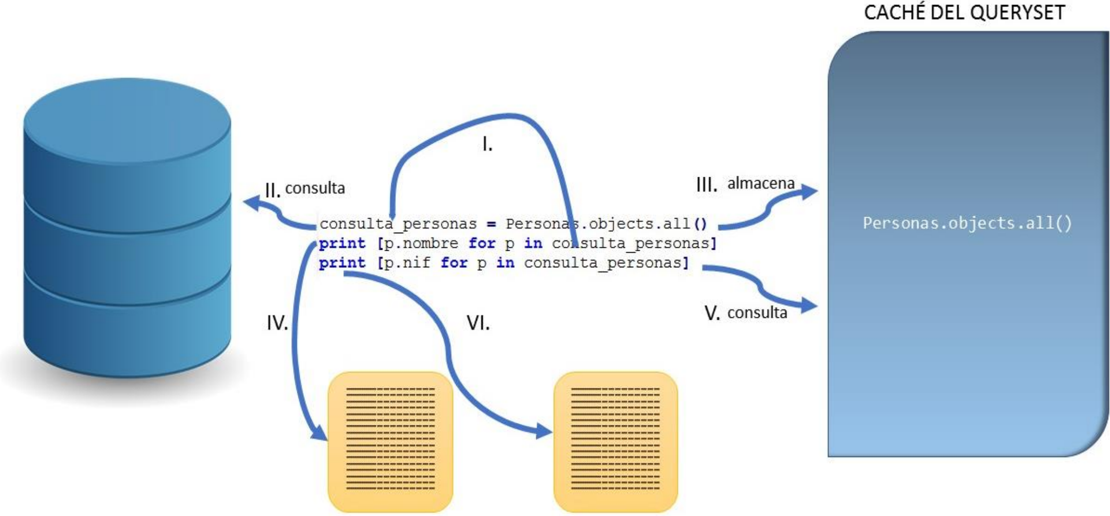


Ilustración 5 - Caché de consultas en Django.

Como se puede observar en este ejemplo anterior, la primera consulta, lee los datos de la correspondiente tabla cuando es evaluada, y almacena el resultado en una variable. Debido a que tiene la caché libre, introducirá los resultados en ella, para que las siguientes evaluaciones de ese *Queryset*, no beban de la base de datos, optimizando así su uso.

Este sistema de cacheo de consultas ofrece una particularidad y es que no siempre se almacenan los resultados de las consultas en la caché. Esto se produce, cuando no se llena dicha memoria, al evaluar una parte del *Queryset*. Normalmente esto se da cuando se evalúa teniendo en cuenta un índice de la tabla.

Aunque en una aplicación con una base de datos pequeña no tiene un gran impacto de rendimiento, en una aplicación de un alcance mucho mayor, se apreciará perfectamente dicho potencial que nos ofrece este Framework.

9.10 Migraciones

Es posible ir modificando el archivo *models.py* y por lo tanto los modelos durante el desarrollo de una aplicación Django. Por ejemplo, añadiendo nuevos campos o eliminando campos existentes, nuevas relaciones entre clases, entre otras modificaciones. Dichas operaciones que se realizan contra los modelos surgen efecto gracias a las migraciones de Django. Estas migraciones funcionan sobre las aplicaciones Django, y permiten gestionar y trabajar con el esquema de la base de datos mediante una abstracción que aporta al programador. Gracias a las migraciones, el programador no tendrá que ocuparse personalmente de realizar dichos cambios manualmente sobre la base de datos, ya que de ello se encargará Django mediante las migraciones.

También cabe comentar que existe un sistema de control de versiones, en el cual se registrarán los cambios que se realicen sobre los modelos, pudiendo volver a un punto anterior si es oportuno.

Existen diferentes comandos para interactuar con las migraciones:

- **Migrate**: se encarga de crear y eliminar las tablas de la base de datos. Se puede entender como una sincronización de los modelos con la base de datos existente.
- **Makemigrations**: crea nuevas migraciones basadas en los cambios aplicados al modelo.
- **Sqlmigrate**: no crea ni modifica la base de datos, solo imprime la salida para poder ver el script SQL que ejecutaría si hiciera la migración con el objetivo de que pueda ser analizado por el programador para ver si se realizan los cambios que el estime oportuno.
- **Showmigrations**: permite ver las migraciones que ha tenido un proyecto y su estado.

Gracias a las migraciones los programadores pueden cambiar los modelos existentes sin necesidad de eliminar las tablas o la base de datos que está utilizando, para generar otra nueva, ya que realiza actualizaciones sobre la actual.

Esta potente funcionalidad de Django permite el ahorro de tiempo y de líneas de código a los desarrolladores de un proyecto.

Se pasa a comentar los pasos a seguir cuando se realizan cambios en un determinado modelo.

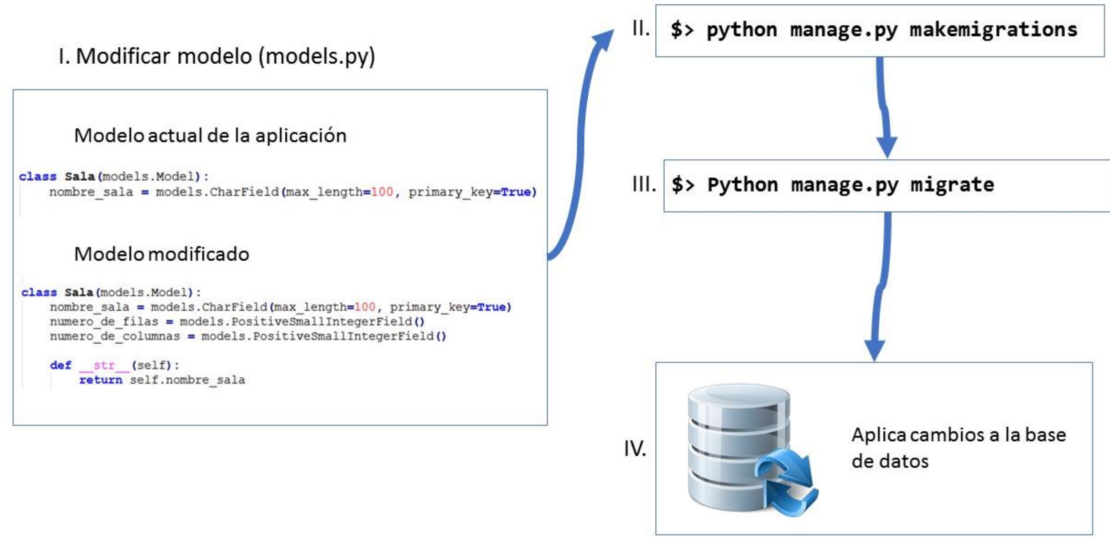


Ilustración 6 - Proceso de modificación de un modelo junto con migraciones.

Como se puede apreciar en la imagen anterior, tras realizar cambios en el modelo, hay que realizar una llamada a *makemigrations*, donde dichos modelos se escanearán y se compararán con las versiones que tienen los archivos de migración, para posteriormente, crear un nuevo conjunto de éstas. Cuando tenemos los nuevos archivos de migración, mediante el comando *migrate*, se aplican los cambios correspondientes a la base de datos. Como queda registrado como un único *commit* en el control de versiones, los próximos desarrolladores, obtendrán todos los cambios que se han producido en el modelo sin tener que ejecutar todos los archivos de migración existentes.

En cuanto a los archivos de migración, son ficheros Python que se almacenan en una carpeta llamada "*\migrations*" dentro de la aplicación. Esos archivos definen una clase *Migration* que es una subclase de *django.db.migrations.Migration*. Dicha subclase consta de distintas partes como por ejemplo las más frecuentes que son:

- **Dependencias**: es una lista de migraciones de las que depende.
- **Operaciones**: es una lista de clases de operación que implica lo que realizará dicha migración.
- **initial**: indica si la migración es la inicial del proyecto.

A continuación, se muestra un ejemplo de una migración:

```
from __future__ import unicode_literals
from django.conf import settings
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):
    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
        ('appcine', '0004_delete_cliente'),
    ]

    operations = [
        migrations.CreateModel(
            name='Cliente',
            fields=[
                ('id', models.AutoField(auto_created=True,
primary_key=True, serialize=False, verbose_name='ID')),
                ('fecha_nacimiento', models.DateTimeField()),
                ('numero_telefono', models.IntegerField()),
                ('user',
models.OneToOneField(on_delete=django.db.models.deletion.CASCADE,
to=settings.AUTH_USER_MODEL)),
            ],
        ),
    ]
```

Una migración es inicial si dentro de la clase *Migrations*, tiene el valor de *initial* a *True* y la lista de dependencias está vacía. Son las migraciones que crean la primera versión de las tablas que usará la aplicación.

```
from __future__ import unicode_literals
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):
    initial = True
    dependencies = [
    ]
    operations = [
        migrations.CreateModel(
            name='Cliente',
            fields=[
                ('usuario',          models.CharField(max_length=100,
primary_key=True, serialize=False)),
                ('password', models.CharField(max_length=40)),
                ('email', models.EmailField(max_length=254)),
            ],
        ),

...

      ]
```

Como se ha descrito brevemente en la introducción de este capítulo, el objetivo de Django es reducir al máximo el acoplamiento entre las distintas partes de la aplicación. Es por ello, que esta capa del modelo está diseñada para que no tenga una limitación a un motor

de base de datos específico o que, para cada uno de los distintos motores, tenga que programarse de una forma u otra.

De esta manera, se podrá implementar los diferentes modelos para una aplicación sin un requisito previo que sea conocer dicha base de datos subyacente.

Es necesario conocer, que cuando se crea un nuevo proyecto en Django, en el archivo de configuración settings.py, es donde se debe de indicar el *backend* de la base de datos a utilizar en el proyecto. Esta configuración se llevará a cabo en el apartado *DATABASES* de dicho archivo. En dicho diccionario Python, se deberán de indicar los *backends* de la base de datos junto con su configuración predeterminada para el proyecto, así como, la configuración de otras bases de datos adicionales si se requieren. Por defecto viene de la siguiente manera:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Para SQLite3 que viene por defecto, no es necesario añadir nada adicional, pero si se desea utilizar otra como por ejemplo PostgreSQL, se deberá de añadir un usuario y contraseña, así como la dirección IP y el puerto de la base de datos.

10 Capa de la Vista

10.1 Introducción

Con la capa del modelo comentada en el apartado anterior, quedan definidos los datos que usará una aplicación Django, así como la relaciones entre ellos, quedando almacenado en la base de datos. Una vez que se tiene establecida dicha información, tiene que ser procesada por la capa de la vista para devolver al usuario el conjunto de elementos o elementos individuales que solicite. Por lo tanto, la capa de la vista se encargará de dicha gestión de la información almacenada por el modelo para pasárselo a la siguiente capa, la capa de las plantillas y que ella se encargue finalmente de cómo se muestra al usuario.

De esta forma se muestra la relación existente entre las tres capas de este patrón software. Ya que la capa del modelo contendrá los datos que usará la vista para procesarlos y determinar cuáles de ellos muestra al cliente y, por último, se los mandará a la siguiente capa para que los exponga al usuario.

10.2 Vistas

Las vistas en Django, por lo tanto, son las encargadas de manejar las peticiones Web que un usuario produce en una aplicación. Estas vistas son, en un principio, funciones Python, las cuales, reciben como argumento una solicitud Web, que será un objeto *HttpRequest*, y devuelve la respuesta requerida mediante un objeto *HttpResponse*. De esta forma se mostrará al usuario final el conjunto de datos solicitados, lo cual, añade un punto de dinamismo a las páginas web, ya que no solo se devolverán páginas estáticas.

Entre muchas de las opciones posibles que se le puede devolver a un usuario están: una página HTML, una lista de elementos, una imagen, un código de error si la solicitud no se puede aportar, entre otras opciones.

Para conseguir lo anteriormente comentado, las vistas contienen toda la lógica requerida para manejar la petición del usuario y mandarle la respuesta. Dichas vistas que se van a utilizar en la aplicación se implementarán en el archivo que se genera automáticamente al crear un proyecto Django, *views.py*.

El contenido de una página de la aplicación dependerá de la función vista que tenga asociada una determinada URL. Las URLs son el medio que tiene el usuario para relacionarse con la aplicación y poder solicitar objetos. Por lo tanto, es necesario establecer una relación entre dichas URLs y las distintas vistas disponibles en la aplicación, para que devuelvan exactamente los datos solicitados por el cliente.

Se pasa a mostrar un diagrama para entender como la capa de la vista, maneja interiormente las peticiones HTTP que se producen en una aplicación:

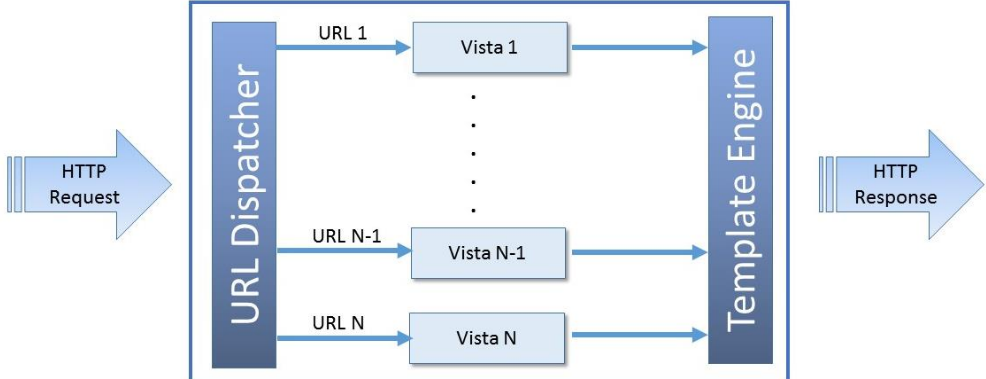


Ilustración 7 - Manejo de solicitudes y respuestas HTTP por la capa de la vista.

En este diagrama anterior se puede apreciar cómo una petición *HttpRequest* que realiza un cliente es manejada por la capa de la vista:

I. Se recibe la petición del usuario de la aplicación.
II. Dicha dirección URL es manejada por el mapeador de URLs que dispone la capa de la vista, la cual realizará el mapeo entre la URL recibida y la vista que tiene asociada en el *URLconf*.
III. La vista se pasa al motor de plantillas, el cual, se comentará en el siguiente capítulo.
IV. De esta manera, se devuelve al cliente una respuesta *HttpResponse*.

10.2.1 Mapeo de URLs - URLconf

Como se ha comentado en el punto anterior, una vista es una función Python, por lo que será necesario llamarla desde alguna parte de la aplicación para que realice su desempeño. Para ello, es necesario establecer una correspondencia entre una la vista y una determinada dirección URL ya que esta será la manera de realizar solicitudes a la aplicación por parte del usuario. De esta manera Django permite al programador tener bajo un mayor control el nombre de las URL que estarán disponible para su aplicación, a diferencia de otros frameworks de programación web.

Para realizar dicho mapeo de un determinado patrón URL y una vista, se utiliza el *URLconf*. El *URLconf* es un archivo donde se implementarán una serie de patrones en el cual Django buscará una correspondencia con la URL recibida en un determinado momento, para determinar que función vista se debe de utilizar.

Al crear un proyecto web, Django automáticamente crea un *URLconf*, llamado *urls.py*. En ese archivo Python, se encuentra la variable *urlpatterns* que debe de ser una lista de instancias *django.conf.urls.url()*. Esta función *url()* es la que contendrá el mapeo entre las posibles URLs y las funciones que serán llamadas por cada una de ellas con el objetivo de poder atender las peticiones que llegan a la aplicación.

Cada llamada al método url() tendrá la siguiente estructura:

```
(Expresión regular, vista, diccionario, nombre)
```

De esa tupla anterior, se va a comentar cada una de sus partes en los siguientes apartados.

10.2.1.1 Expresiones regulares

En cuanto al primer argumento, que son las expresiones regulares (*regex*), cabe comentar que sirven para buscar patrones en un texto, aunque en este caso especial, será sobre una dirección URL. Con ello, cuando se realice una petición web, se buscará el primer patrón coincidente con la dirección URL dada de la lista contenida en *urlpatterns*. Con ello se podrá establecer qué función, que ocupa el segundo valor de la tupla, será llamada para esa dirección. Las expresiones regulares deben de ser compatibles con el módulo *re* de Python. Existen diferentes caracteres especiales para usar en las expresiones. A continuación, se pasan a comentar alguno de ellos:

Caracteres especiales	Descripción
'.'	Cualquier carácter.
'^'	Marca el comienzo de una cadena.
'$'	Marca el final de una cadena. VER
'*'	Cero o más ocurrencias de la expresión regular anterior.
'+'	Una o más ocurrencias de la expresión regular anterior.
'?'	0 o 1 veces la expresión regular anterior.
'\'	Permite escapar caracteres especiales.

Tabla 6 - Tabla de caracteres especiales.

Esto es solo una pequeña lista de caracteres especiales, la lista entera está disponible en [7].

Es posible capturar el contenido de una URL con el fin de poder hacer más dinámico el contenido de las páginas. Para ello, cuando se define un patrón, la notación que se pone entre paréntesis será capturada para poderla pasar a la función correspondiente como argumentos posicionales. El objetivo de esto es poder mostrar unos resultados u otros dependiendo de lo que el usuario solicite a través de la URL ya que captura los valores contenidos en una cierta posición. Existe una forma más eficiente de realizar dichas expresiones, que es utilizando argumentos clave en lugar de argumentos posicionales. Esto es debido a la forma que tiene Python de trabajar con las funciones, ya que es posible pasarla argumentos de ambos tipos. La principal diferencia es que, con los posicionales, se debe de tener en cuenta el orden de dichos argumentos en la llamada a la función, mientras que en los argumentos claves, se tiene que especificar el nombre de los mismos junto con los valores correspondientes. Para ello se realiza de la siguiente manera: *?P<nombre>patrón*, donde nombre será el nombre del grupo y después se pone el patrón a buscar. Esto permite al programador no tener que variar el orden de los parámetros de las vistas si lo invierte en la URL, pese a que el código queda algo más engorroso.

```
url(r'^pelicula/(?P<titulo>[^/]+)/$', views.vista_pelicula),
```

Como nota importante de este primer argumento, es que las expresiones regulares que son muy complejas pueden aportar un bajo rendimiento en la búsqueda de patrones. Por

lo que es necesario no depender estrictamente de esta parte ya que se podría ralentizar el sistema.

Las expresiones regulares definidas en *urlpatterns* se compilan la primera vez que se accede a ellas, cuando se carga el módulo *URLconf*. El objetivo principal de dicha compilación inicial es poder hacer el sistema más veloz.

10.2.1.2 Función vista

La segunda parte, consta de la función que se llamará si el patrón anteriormente descrito coincide con la URL. Por lo que en caso de coincidencia Django llamará a esa función específica. Dicha llamada, recibirá como argumento obligatorio un objeto *HttpRequest*. Aparte de dicho argumento obligatorio puede requerir otros adicionales, que como se ha comentado antes, serán capturados de la URL y podrán ser tanto argumentos posicionales como argumentos clave.

Para poder tener acceso a dichas funciones, es necesario incluir el módulo de Python donde se incluyen éstas. Como nota importante, cabe comentar que se pueden llamar a vistas de otras aplicaciones siempre que se importen. A continuación, se muestra un ejemplo donde se puede apreciar que se utilizan dos vistas de aplicaciones distintas.

```
url(r'^cliente/registrar/$', views.registrar_cliente),
url(r'^cliente/login/$', auth_views.LoginView.as_view()),
```

10.2.1.3 Diccionario de argumentos

La tercera parte que puede recibir la función *django.conf.urls.url()*, es un diccionario de argumentos, que es opcional a la hora de definir un patrón en Django. Estos valores, son argumentos de palabra clave que se pasarán a la función vista correspondiente. Agregan una potente ayuda al programador, ya que permiten pasar dichos argumentos a la vista con el objetivo de evitar tener que repetir código. Por ejemplo, es muy útil cuando se pasen unos argumentos fijos a la función que no dependan de la URL, entre otras opciones. Estos argumentos extra, en caso de conflicto con argumentos capturados de la URL, tienen precedencia sobre estos últimos.

```
url(r'^cliente/registrar/$', views.registrar_cliente, {mi_variable =
True}),
```

10.2.1.4 Nombre de patrones

Es posible nombrar a los patrones URL mediante un nombre para poder referirnos a él explícitamente. Ese nombre puede ser cualquier cadena de caracteres que se desee, pero es necesario asegurarse de que solo ese patrón se llamará con ese nombre para eliminar la posibilidad de conflictos con otras partes de la aplicación. Para especificar el nombre se realiza asignando una cadena de caracteres a *name*.

```
url(r'^cartelera/$', views.cartelera, name='cartelera'),
```

Como se ha comentado en el párrafo anterior, es posible nombrar a las URLs con *name*. Pero esto añade un problema, ya que dentro de un proyecto Django, varias aplicaciones pueden usar el mismo nombre para la URL. Por ello, es necesario distinguir a que aplicación se refiere. Esto que se acaba de comentar se puede solucionar añadiendo espacios de nombres a *URLconf*, con *app_name.*

```
app_name = 'appcine'
urlpatterns = { … }
```

Nombrar a los patrones de URL puede ser muy útil cuando se pretenden resolver URLs inversas, lo cual se comentará en el siguiente apartado.

10.2.1.5 Añadir URLconf adicionales

A un urlpatterns es posible añadirle otros *URLconf* con la finalidad de no tener que repetir esos patrones o parte de ellos que ya están definidos. Esta adición se realiza mediante el método *django.conf.urls.include()*, metiendo entre paréntesis el módulo que se desea añadir. Aparte de poder incluir otros *URLconfs*, se pueden meter patrones adicionales usando una lista de instancias de *url()*. Con todo ello, es posible disponer de un código más legible y eliminar redundancias que existían en el código.

10.2.1.6 Acoplamiento débil del código

Se puede llegar a la conclusión que para desarrollar páginas Web usando este Framework, se deben de escribir funciones de vista y después relacionarlas con una dirección URL para que sean llamadas. Con ello, Django aporta un acoplamiento débil, ya que la definición de una dirección URL y una función de vista, se realiza por separado, por lo que es posible implementar una de las partes sin afectar a la otra. Esto resalta en cuanto a otros Frameworks de programación Web, ya que aporta esta facilidad extra al programador, en comparación por ejemplo con PHP. Otra mejora que aporta este Framework es que permite realizar URLs que sean amigables a los usuarios y desarrolladores, ya que permite un nivel de personalización muy alto.

10.2.2 Resolución inversa de URLs

Django permite utilizar las URLs definidas en el *URLconf* de dos maneras. La primera y más común, como se ha comentado hasta ahora, es el usuario mediante su navegador quien realiza una petición mediante la dirección URL y posteriormente se llama a la vista correspondiente asociada a ese patrón URL. La segunda forma de resolución de URL se da cuando es conocida una vista a la que es necesario llamar junto con los argumentos que tendrá la misma, por lo que Django deberá de componer dicha URL. A esta segunda manera se la conoce con el nombre de resolución inversa de URLs.

Es necesario que existan estas dos maneras de usar el *URLconf* ya que a parte de las solicitudes que realiza el usuario, a veces es necesario obtener las URLs finales, para realizar las redirecciones, incluir elementos en ellas, entre otras funcionalidades.

Existen distintos mecanismos para realizar este segundo tipo de resolución dependiendo de en qué lugar de la aplicación se vaya a aplicar. Para relacionar el manejo de URLs de

instancias de modelos, Django aporta el método *get_absolute_url()*. Para las plantillas se puede obtener dicha resolución mediante la etiqueta *url* y por último, en código Python, usando la función *reverse()*.

Para realizar este tipo de resolución, es necesario utilizar patrones de URL con nombre, tal y como se ha comentado anteriormente en el Capítulo 10, apartado 2.1.4.

Ejemplo en vista:

```
return HttpResponseRedirect(reverse('compra_ok', args=(id_compra,)))
#redirigir a pagina de compra
```

Ejemplo en plantilla:

```
<a class = "titulo" href="{% url 'pelicula_detalle' sesion.nombre_pelicula.titulo %}">{{ sesion.nombre_pelicula.titulo }}</a>
```

10.2.3 Proceso petición web

En este apartado se va a comentar el proceso de una petición web que realiza el usuario, hasta que se le devuelve lo que éste desea.

- Primero, el cliente, mediante una URL solicita el recurso que desea.
- Django, busca en *settings.py* el valor de la variable *ROOT_URLCONF*, ya que apunta a la *URLconf* a usar en la aplicación actual.
- Django carga ese módulo de Python y dentro de la variable *urlpatterns* busca la primera coincidencia con el patrón URL introducido por el usuario final.
    - Si encuentra una coincidencia llama a la vista dada, pasándole como parámetros, una instancia de *HttpRequest* y los valores obtenidos de la URL, si tiene.
        - Una vez que la vista es llamada, realiza todos los pasos internos y devuelve un objeto *HttpRespone*. Ese objeto, Django lo convierte en una respuesta HTTP, que se muestra en forma de página web al usuario para una mayor legibilidad.
    - Si no encuentra ninguna coincidencia en la variable *urlpatterns* o se ha producido algún tipo de error, Django llama a una vista de manejo de errores. Cuando Django llama a esa función, se devuelve una página al usuario indicando que se ha producido un error.

10.2.4 Vistas basadas en clases

Django da una característica extra a las vistas de la aplicación, ya que aparte de poder ser funciones Python que devuelven un *HttpResponse***,** es posible que las vistas sean clases. En las primeras versiones de Django no se permitía, pero la introducción de ello permitió revolucionar el uso de las vistas, ya que, antes se basaban en funciones y por lo tanto no permitían la herencia. El objetivo de dicha utilización de clases como vistas es el ahorro y simplificación del código, ya que, ese es el objetivo principal de Django. Este ahorro se

debe a que abstraen expresiones y patrones que son frecuentemente utilizados en el desarrollo web y también por el uso de herencia y *mixins*.

Este framework de programación web, ofrece algunas vistas basadas en clase predefinidas con el objetivo de ayudar al programador con las tareas más comunes existentes en este ámbito. Algunas de esas tareas pueden ser, mostrar datos de una tabla de la base de datos y el detalle, redirigir a otra página de la aplicación, entre otras. Estas vistas que nos ofrece Django toman el nombre de vistas genéricas.

A continuación, se nombran algunas vistas genéricas que ofrece Django:

USO	Vista genérica
Vistas genéricas para mostrar datos	DetailView y ListView
Vistas genéricas de edición	FormView, CreateView, UpdateView, DeleteView.
Vistas genéricas de fechas	ArchiveIndexView, YearArchiveView, MonthArchiveView, WeekArchiveView, DayArchiveView, TodayArchiveView, DateDetailView.

Tabla 7 - Tabla de vistas genéricas basadas en clase de Django.

Gracias a la utilización de estas clases, Django permite al desarrollador no tener que implementar dichos comportamientos que son considerados comunes dentro de la programación web.

Para usar dichas vistas genéricas, existen varias formas. La primera de ellas y más potente es mediante la creación de una subclase, es decir, la implementación de una clase que heredará de la vista genérica existente. Con ello, es posible modificar atributos y métodos para que cumplan los requisitos del programador. La segunda forma, más sencilla que la anterior, es crearlas directamente en el *URLconf*, con una llamada al método *as_view()*. Es necesario realizarla con una llamada a ese método que traen dichas clases ya que Django llama a una función, no a una clase. Con ese método creará una instancia de la clase en cuestión y llamará a su método *dispatch()* para determinar la petición enviada por el usuario en la URL. Esta segunda forma conviene cuando no se cambian muchos aspectos de la clase y solo se modifican algunos atributos. En este último caso, los atributos modificados de la clase se meterán como argumento de dicha llamada y reemplazarán a los atributos de la clase existentes.

Django aporta distintas opciones que se pueden utilizar en las vistas genéricas basadas en clase, con el objetivo de poder personalizarlas al máximo. Algunas de ellas son:

- De las clases anteriores que manejan conjunto de objetos, contienen en una variable *object_list*, la cual, será necesario iterar para recorrer todo su conjunto. Este nombre, no es identificativo para un determinado modelo, por lo que puede causar confusión si se usan varias vistas de este tipo. Para ello, mediante *context_object_name* se puede indicar el nombre para dicho conjunto de objetos.
- Otra opción es la de añadir contenido extra a una vista para mostrar información más allá del contenido de la misma. Se puede realizar creando una subclase de la

vista genérica elegida y modificando el método g*et_context_data* donde se podrá añadir el contenido deseado.

- A esas clases, se la pueden pasar un modelo específico para que opere con sus datos. Pero aparte de poder pasar un modelo específico, se puede un subconjunto del mismo. Para ello, mediante la variable *queryset*, se definirá una consulta con los datos filtrados según los requisitos del programador.

Todas estas vistas que se acaban de comentar heredan de la vista base *View*. Existen 3 vistas base que se pueden considerar que son el padre de las demás. Se pasa a describir brevemente cada una de ellas:

- *View (django.views.generic.base.View)*. Todas las vistas basadas en clases heredan de ella. Maneja las conexiones de la vista y las URLs.
- *TemplateView (django.views.generic.base.TemplateView)*. Renderiza una plantilla dada basándose en los argumentos capturados de la URL.
- *RedirectView (django.views.generic.base.RedirectView)*. Se encarga del redireccionamiento HTTP.

Como se puede apreciar, estas clases base se encargan de las conexiones de las vistas y de las URLs. Es posible heredar de ellas o usarlas en sí mismas, aunque por si solas no aportan todas las necesidades para un determinado proyecto. Para extender su funcionalidad se pueden utilizar las vistas genéricas o mezclarlas con *mixins*.

Aparte de estas vistas que nos aporta Django, es posible crear una estructura de vistas con el fin de que cumplan cuidadosamente los requisitos del programador.

10.2.4.1 Mixins y decorators

Con el objetivo de aumentar y personalizar la funcionalidad de las vistas, Django aporta a los programadores *mixins* y *decoradores* para tal fin.

Un *mixin* permite la herencia múltiple en las vistas, por lo que se puede utilizar y modificar los métodos de las distintas clases padres. Los *mixins* permiten la reutilización de código y de esta manera evitar tener que duplicar los métodos. Dicha reutilización de código es el principal objetivo de Django.

Django aporta distintos *mixins* a los programadores. Se pasa a nombrar cada uno de ellos.

Grupo	Mixins
Mixins simples	ContextMixin, TemplateResponseMixin.
Mixins para recuperar objetos	SingleObjectMixin, SingleObjectTemplateResponseMixin.
Mixins para buscar multiples objetos	MultipleObjectMixin, MultipleObjectTemplateResponseMixin.
Mixins de edición	FormMixin, ModelFormMixin, ProcessFormView, DeletionMixin.
Mixins para fechas	YearMixin, MonthMixin, DayMixin, WeekMixin, DateMixin, BaseDateListView.

Tabla 8 - Tabla de grupos de Mixins de Django.

Al igual que sucede con las vistas genéricas, se pueden escribir *mixins* propios si los mencionados anteriormente no realizan la tarea que necesita el programador exactamente.

Los *mixins* aparte de producir todas esas ayudas y mejoras al programador, cabe comentar que cuanto más se eleva su utilización en la aplicación, el código deja de ser tan legible debido a que estará más disperso.

Aparte de la utilización de los *mixins* para aumentar la funcionalidad, existe la posibilidad de utilizar decoradores en las vistas basadas en clase.

Los decoradores se utilizarán de una manera distinta si se hereda de una clase base, o si se utilizan en el *URLconf*. En cuanto al primer caso, mediante el decorador *@method_decorator*, decora un método de la clase. Dicho decorador hay que incluirlo en el método *dispatch()* de la clase para decorar la definición de la misma, ya que dicho método es el que se encarga de aceptar un objeto *request* y devolver un objeto *response*. Del segundo caso cabe comentar que se puede utilizar el decorador directamente en el *URLconf*, decorando la salida del método *as_view()*.

Un ejemplo de esta segunda forma es:

```
url(r'^cliente/login/$', auth_views.LoginView.as_view(template_name =
'login_cliente.html'), name = 'login'),
```

De esta manera un programador podrá personalizar las vistas tal y como desee aplicando los métodos expuestos anteriormente. Con ello logrará personalizar su aplicación sin necesidad de escribir muchas líneas de código que serían necesarias de no ser por dichas ventajas que aporta Django.

10.2.5 Vistas de error

Es posible llevar un control de los errores HTTP que se pueden producir en una aplicación web. Si el usuario intenta acceder a un recurso en la aplicación Web y se produce un error, se devuelve un código error HTTP, mediante una plantilla por defecto, indicando el motivo del fallo. Estos errores pueden ser por distintos motivos, tanto por parte del cliente como del servidor. Algunos ejemplos son: intentar acceder a un recurso que no está disponible, acceso no autorizado a un recurso, utilizar un método de solicitud no soportado por dicha URL, que el servidor no esté útil, entre muchos otros. Estos mensajes de error es posible manejarlos tanto a nivel de vista como a nivel de plantillas.

Django ofrece subclases de HttpResponse para los errores Http más comunes, como se puede ver en la siguiente tabla:

Subclase de HTTPResponse	Código de Error HTTP
HttpResponseBadRequest	400
HttpResponseNotFound	404
HttpResponseForbidden	403
HttpResponseNotAllowed	405
HttpResponseGone	410
HttpResponseServerError	500

Tabla 9 - Tabla de errores HTTP junto con subclases de HTTPResponse.

Por lo que para devolver un error en una función vista, es necesario retornar una instancia de alguna de estas subclases, dependiendo del tipo de error, en lugar de devolver una instancia de *HttpResponse*, como se hacía anteriormente. De esta manera, se manda el mensaje de error al usuario informándole de la incidencia producida.

Cabe comentar que el programador puede definir subclases de *HttpResponse* para controlar distintos errores no contemplados en la tabla anterior.

10.2.5.1 Manejo de errores HTTP comunes

Django aporta excepciones para los errores HTTP más comunes, como por ejemplo para el error 404 aporta una excepción *Http404*. Cuando es lanzada esa excepción, devuelve una página de error estándar para la aplicación. Es posible personalizar los datos mostrados en dicha página gracias al manejo de las vistas para los errores. Por otro lado, en cuanto a la presentación de los datos de esa página, también es posible modificar la plantilla.

Es necesario conocer, que Django aporta 4 variables que apuntarán a la vista correspondiente para cada error HTTP. Estas variables son: *handler400*, *handler403*, *handler404*, *handler500*. Estas variables tienen un valor, que será el de la vista creada por defecto para esos errores. Por ejemplo en el caso del error 404 apunta a la vista por defecto: *django.views.defaults.page_not_found*. Es posible modificar dicha vista, para ello, se debe de poner que esa variable contenga la nueva vista creada para ese tipo de error, en el *URLconf* de la aplicación en uso.

Esto que se acaba de comentar es válido para los diferentes errores HTTP comunes, como son el 400, 403, 404 y 500.

Para el error HTTP más común, el 404, Django nos aporta unas funciones a modo de atajo para las vistas: *get_object_or_404()* y *get_list_or_404()*. Estos métodos sirven para utilizarlos dentro de una vista, con el objetivo de recuperar un determinado objeto o lista ellos para mostrarlos como resultado de la vista, y en caso de ser la consulta vacía, lanzar la excepción 404.

- El método *get_object_or_404()* realiza una llamada al método *get()* del *Manager* correspondiente y en caso de no existir dicho objeto, devuelve la excepción *Http404*.
- El segundo método, get_list_or_404() funciona de la misma manera que el primero, pero retornando una lista que cumple las condiciones pasadas en el método *filter()*.

Ejemplo de uso:

```
no_usuario = get_object_or_404(Cliente, user = usuario)
```

Con ello se logra tener un acoplamiento débil en la aplicación, ya que, si no, se tendría que tratar este tipo de excepciones en la capa superior, lo que acoplaría fuertemente la capa del modelo y la capa de la vista.

Cabe comentar que las páginas de error personalizadas no se mostrarán si el modo *DEBUG* del fichero "*settings.py*" se encuentra activado. Por defecto, dicha configuración se encuentra activada con el fin de poder ayudar al programador en su tarea de depurar los errores que le vayan surgiendo. En la documentación oficial de Django, recomiendan que una vez terminado el desarrollo de la aplicación Web se entregue con el modo *DEBUG* desactivado, ya que si no por el contrario, los usuarios pueden extraer información de los metadatos del entorno de la aplicación, como el *traceback* de Python, que contiene información como nombre del archivo, de la función, entre otra información relevante.

Se muestra un ejemplo de la página que aparece cuando se comete un error durante el desarrollo:

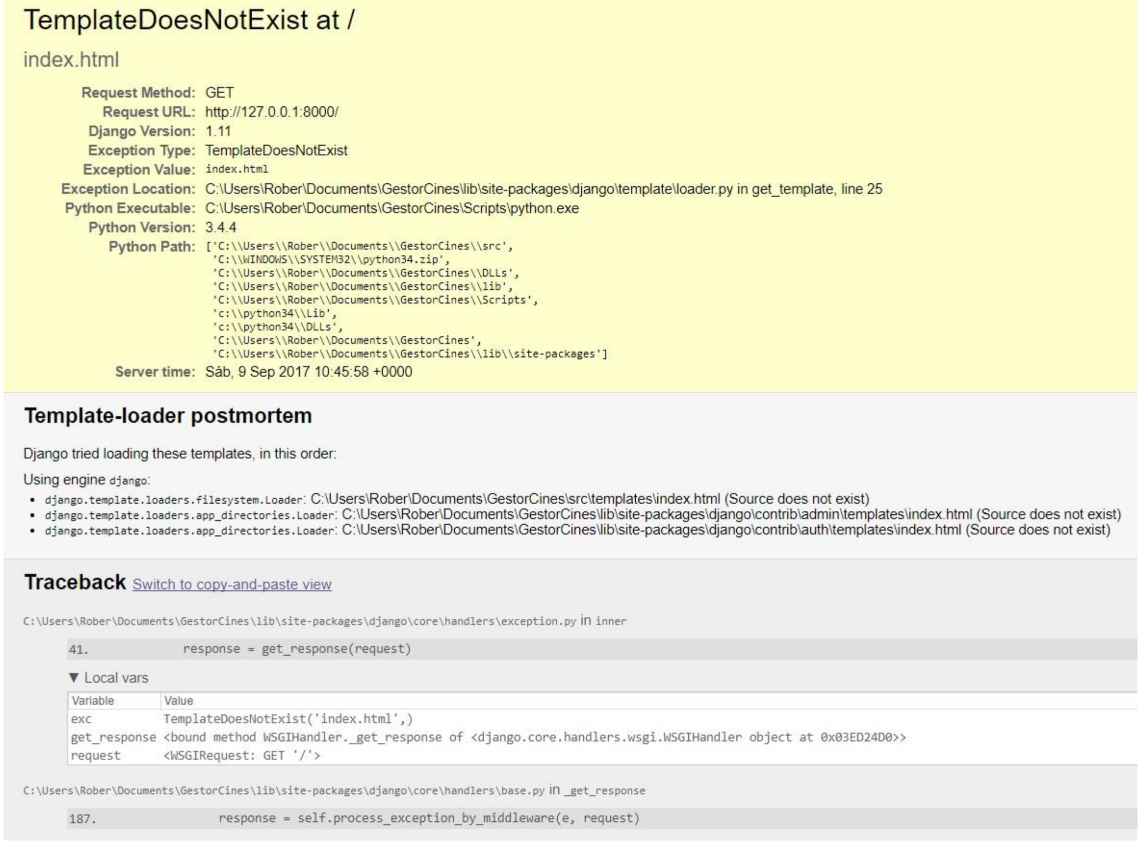


Ilustración 8 - Ejemplo de Traceback de un error en una aplicación.

10.2.6 Componentes Middleware de Django

Django incorpora el Framework *Middleware* de bajo nivel, con el objetivo de que el desarrollador pueda modificar las peticiones y respuestas, según sus requerimientos, que tienen lugar en la aplicación. Cada componente middleware se encarga de unas funciones específicas y dichos componentes son una clase Python.

En este framework vienen integrados distintos componentes *middleware*, pero el programador puede diseñar a su gusto otros aparte. Algunos ejemplos de grupos de componentes que vienen integrados en Django son:

- *Middleware* de excepciones, nuevo en 1.10.

- *Middleware* GZip, para navegadores que incluyen la compresión gzip.
- *Middleware* de mensajes.
- *Middleware* de soporte para seguridad.

Como nota importante, es necesario mencionar que para usar los distintos componentes es necesario instalarlo en la aplicación. Para ello, en el archivo de configuración *settings.py*, en la variable *MIDDLEWARE*, se añadirán siguiendo un determinado orden, la ruta de los distintos componentes a utilizar.

Es necesario tener en cuenta el orden que son declarados ya que la salida de un *middleware* puede influir en otro. Antes de llamar a la vista es cuando Django emplea el *middleware*. Para ello, mira en la variable *MIDDLEWARE* comentada anteriormente, de arriba abajo y en último lugar, si no se producen interrupciones, es cuando se produce la llamada a la vista. Estas interrupciones se producen si una capa bloquea a las demás, es decir, si no llama a *get_response()*. La respuesta volverá en el orden inverso, de abajo a arriba.

En cuanto a las excepciones que se pueden producir en los distintos componentes del *middleware*, cabe comentar que no es necesario llevar el control de las mismas ya que si no, devolverán un objeto *HttpResponse* con el valor de *status_code* a 404. Por lo tanto, no es necesario estar controlado las posibles excepciones ya que Django lo simplifica añadiendo esa respuesta de código de error HTTP.

Por defecto en settings.py se incluyen los siguientes:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

11 Capa de Plantillas

Esta última capa por comentar del modelo MTV de Django es la encargada de la presentación de los datos de la aplicación en el navegador web del usuario final. La capa de plantillas define, por lo tanto, cómo se va a visualizar la información en la página web correspondiente. Dichos datos que mostrarán las plantillas son proporcionados por la capa anterior, la capa de la vista que es la encargada de definir qué datos se van a mostrar. Por lo tanto, esta capa será la encargada de recibir los datos enviados por la capa de la vista, la cual recoge los datos específicos del modelo, y mostrarlos en una plantilla.

Gracias a la disponibilidad de las distintas capas de Django, se permite al programador disponer la lógica de programación separada de la lógica de presentación, lo que permite una mayor reusabilidad del código. Con esta última capa, quedaría completada la explicación de la relación Modelo – Vista – Plantilla.

Otro aspecto importante es que esta capa debe de ser capaz de generar tanto contenido estático como dinámico para las distintas páginas web que se puedan dar en la aplicación. De esta forma se pueden construir aplicaciones web mucho más amigables y prácticas para los usuarios finales.

11.1 Plantilla en Django

Una plantilla en Django define la forma en que se va a presentar la información al usuario de la aplicación. Puede estar compuesta de contenido estático y de partes dinámicas.

Una plantilla será una instancia de *Template*, que es comúnmente usada para generar páginas HTML de tal forma que el usuario vea el contenido a través de su navegador, pero no tiene que ser estrictamente utilizada para crear dicho tipo de código, si no que puede producir cualquier formato de tipo texto como XML, CSV, xHTML, entre otros.

Django tiene su propio motor de plantillas, mediante el cual se puede realizar herencia y posee una sintaxis particular para poder seleccionar objetos y diversos métodos para poder interactuar con el HTML de la página. A esta sintaxis se le conoce con el nombre de DTL, del inglés, "*Django Template Language*". Un proyecto Django, se puede configurar, usando varios motores de plantillas, o en su defecto ninguno, si la aplicación no utiliza plantillas.

La forma que tienen las plantillas de recibir datos que le proporcionan la vista es mediante los contextos, que se corresponderán con un diccionario Python. Con esos contextos y después del proceso de renderizado de las plantillas, éstas contarán con los datos recogidos por la vista de la base de datos con el objetivo de mostrarlo al cliente. Quedando en manos de la plantilla ordenarlo y mostrarlo correctamente.

11.2 Configuración y uso

Es necesario configurar los motores de plantillas que se vayan a utilizar en una aplicación en Django con el objetivo de indicar al framework como debe de realizar las distintas opciones de gestión de las mismas. Dicha configuración se debe de realizar en el archivo *settings.py*, generado al crear el proyecto. Dentro de él, en la configuración de *TEMPLATES*, se le añadirá una lista de configuraciones para cada motor de plantillas con la siguiente información:

- *BACKEND* → Indica el *backend* de plantillas a utilizar.
- *DIRS* → Contendrá una lista de directorios donde el motor de plantillas buscará, por orden del listado, los archivos fuentes de las plantillas.
- *APP_DIRS* → Valor booleano, por defecto a falso, que indica si el motor de plantillas tiene que buscar plantillas dentro de las aplicaciones instaladas.
- *OPTIONS* → Se definirán ajustes específicos para el *backend*.

Para cargar y renderizar las plantillas que son almacenadas en el sistema de archivos, Django aporta una API independientemente del *backend* utilizado. La carga consiste en localizar una plantilla para un identificador determinado y preprocesarla, compilándola generalmente a una representación en memoria.

El módulo de Django *django.template.loader*, aporta dos métodos:

- *get_template(plantilla, using = "motor")* → Se encarga de cargar la plantilla que se pasa como argumento y devuelve un objeto *Template*. Va recorriendo los distintos motores definidos, y si no encuentra la plantilla dada en ninguno de ellos, devuelve el error *TemplateDoesNotExist*. El argumento opcional *using*, sirve para indicar que la busque en un determinado motor de la aplicación.
- *select_template(lista_plantilla, using = "motor")* → De un modo semejante al anterior, devuelve un objeto *Template* de la plantilla primera encontrada existente de la lista que se le pasa como parámetro. El segundo argumento *using* tiene la misma funcionalidad que en el caso anterior.

Estos métodos de carga de plantillas serán utilizados por las vistas para coger la plantilla a renderizar. El proceso completo de cómo se realiza esto que se acaba de comentar se explicará más adelante.

Para cargar dichas plantillas del sistema de archivos es necesario crear una carpeta que por motivos de legibilidad se llamará "*templates*" en el directorio principal del proyecto, *src/templates*. A continuación, hay que indicar dicha configuración en *settings.py* en el apartado *TEMPLATES*. Quedando de la siguiente manera, para utilizar el motor de plantillas de Django y cargarlas de la carpeta *templates*:

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, "templates")],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Separadamente de cargar las plantillas desde el sistema de archivos, es posible hacerlo desde otros orígenes diferentes. Para ello, Django aporta unos cargadores específicos. Se pasa a generar una tabla con los que vienen incluidos en Django, *django.template.loaders*:

Cargador	Descripción
filesystem.Loader	Está activado por defecto, es el encargado de cargar las plantillas desde el sistema de archivos.
app_directories.Loader	Se encuentra activado por defecto y se encomienda a cargar plantillas de aplicaciones Django en el sistema de archivos.
eggs.Loader	Similar al anterior, pero carga las plantillas desde eggs Python. Se encuentra deshabilitado por defecto.
cached.Loader	Con el objetivo de reducir el tiempo que Django emplea en leer y compilar plantillas. Este cargador almacena la plantilla compilada en memoria y dicha instancia se devolverá en solicitudes posteriores.
locmem.Loader	Se encarga de cargar las plantillas de un diccionario de Python.

Tabla 10 - Tabla de cargadores de plantillas.

Aparte de cargar las plantillas de dichas fuentes, se pueden cargar desde otros orígenes implementando un cargador personalizado. Dichos cargadores personalizados, deberán de heredar de la clase base *django.template.loaders.base.Loader* e implementar los métodos *get_contents() y get_template_sources()*.

El motor de plantillas de Django va usando los cargadores que se le pasan en la configuración *TEMPLATES* en la opción "*loaders*" en orden, hasta que alguno de ellos la encuentre.

En cuanto a las excepciones que se pueden producir en este apartado, se pueden dar los siguientes casos:

I. Que no se encuentre la plantilla en los distintos motores de una aplicación. Se lanzará la excepción, *TemplateDoesNotExist.*
II. Otro caso es que se haya encontrado correctamente, pero la plantilla contenga errores de sintaxis en la definición de la misma. Resulta la excepción *TemplateSyntaxError.*

11.3 Backends

Django incorpora dos *Backends* predefinidos que son:

- *django.template.backends.django.DjangoTemplates*. Si se establece este *backend*, se configurará el motor de plantillas de Django, el cual se está describiendo en este apartado.
- *django.template.backends.jinja2.Jinja2*. Se permitirá configurar el motor de plantillas *Jinja2*. *Jinja2,* es un motor completo de plantillas para Python y es de los más utilizados. Está inspirado en el de Django, pero aportando otras herramientas diferentes a los programadores.

Es posible aparte de utilizar dichos motores de plantillas utilizar uno de terceros o incluso crear un *backend* personalizado con el objetivo de utilizar otros lenguajes de plantillas. Para el último caso, es necesario conocer que un *backend* es una clase que hereda de *django.template.backends.base.BaseEngine* y debe de implementar al menos el método *get_template().*

11.4 Django template language (DTL)

Como se ha comentado anteriormente, las plantillas de Django están marcadas mediante el lenguaje DTL, *Django Template Language*. No es estrictamente necesario utilizar el lenguaje de plantillas que nos aporta Django, sino que es posible utilizar uno de terceros, o incluso, escribir un lenguaje de plantillas propio, pero es recomendable debido a la gran potencia que lleva.

Las plantillas de Django utilizan 4 construcciones principales que vienen dadas por el Framework, que son:

I. Variables
II. Filtros
III. Etiquetas
IV. Comentarios

A continuación, se pasa a describir cada uno de ellos.

11.4.1 Variables

Una variable es representada entre "*{{*" y "*}}*". Estas variables tienen unos valores que son pasados a la plantilla mediante el contexto. Cuando una plantilla es renderizada, se reemplazan dichas variables por su contenido. Como nota importante cabe comentar que una plantilla no tiene la posibilidad de asignar una variable o modificar su contenido, solo muestra el mismo.

11.4.2 Filtros

Es posible modificar la salida del contenido de una variable para que se muestre en la plantilla según unos filtros. Para añadir estos filtros es necesario seguir la siguiente estructura: *"{{ mi_variable|filtro1|filtro2 }}"*. Como se puede apreciar es viable añadir tantos filtros como se deseen concatenándolos con "|" y el resultado de un filtro será pasado al siguiente. Algunos ejemplos de los filtros que Django tiene por defecto son:

Filtro	Descripción
add	Añade la cantidad que se le pasa al filtro al valor de la variable.
first	Devuelve el primer elemento de la lista.
lower	Sirve para poner en minúsculas el texto de la variable que tiene asociada.
random	Coge un valor aleatorio de la lista dada.
time	Sirve para dar formato a una fecha según el tipo de formato que se le pase al filtro.

Tabla 11- Tabla de filtros en DTL.

Se pueden escribir filtros personalizados si los que proporciona Django por defecto no cumplen con las expectativas del programador. Para ello, bastará con definir una función en Python que recibirá como argumentos, el valor de la variable y el valor del argumento opcionalmente. Dichas funciones devolverán el valor de la variable modificada según lo programado en la función y deberán de ser registradas en *django.template.Library*. Para registrarlas bastará con usar el decorador *register.filter()*.

11.4.3 Etiquetas de bloques

Las etiquetas de bloques permiten distintas opciones como: agregar estructuras de control, añadir contenido de la base de datos, añadir bucles entre otras opciones. Dichas etiquetas se muestran entre "*{%*" y "*%}*". Algunos ejemplos de etiquetas de bloques pueden ser:

Etiqueta de bloque	Descripción
if / else	Evalúa si se cumple una determinada condición o no. Dependiendo de ello, se ejecutará un fragmento de código u otro.
for	Es un bucle, sirve para recorrer los elementos de una lista.
url	Devuelve la ruta absoluta (URL) que coincide con una determinada vista y los parámetros opcionales.
block / endblock	Define un bloque que será reemplazado por la plantilla hija.
include	Sirve para incluir una plantilla en el contexto actual.

Tabla 12 - Tabla de etiquetas de bloques en DTL.

Como en muchas ocasiones, Django permite personalizar al programador muchas de sus opciones y, por lo tanto, ésta también será una de ellas. Se pueden tanto escribir etiquetas de bloques mediante unas facilidades que aporta Django, o en su defecto, escribirlas desde cero.

11.4.4 Comentarios

El lenguaje DTL permite añadir comentarios de una línea a las plantillas al igual que sucede en otros lenguajes de programación. Para añadir dichos comentarios se deben de situar entre "*{#*" y "*#}*". La cadena de caracteres introducida entre dichos bloques no se mostrará al usuario final.

Se muestra un ejemplo de su uso:

```
{# A continuación, se muestra el formulario #}
```

Como se ha podido observar por los diferentes puntos comentados del DTL de Django, una plantilla no puede ejecutar código Python crudo, ya que este lenguaje se encarga de la presentación del contenido y no de la gestión del mismo. Con las estructuras que se han comentado, se puede presentar el contenido perfectamente adaptado a las necesidades de los desarrolladores, de tal forma que queda separada la lógica de negocios de la lógica de presentación del contenido.

11.5 Escapar código HTML

Como se ha comentado en el apartado anterior, las plantillas de Django contienen variables que se sustituirán a la hora de realizar el renderizado de la plantilla. Por lo que dichos valores pueden tener código HTML, que, al sustituirse en la plantilla, se mostrará al usuario si no es controlado. Es necesario examinar que no se pueda introducir dicho código ya que por el contrario un usuario no legítimo, podría aprovechar dicha brecha de seguridad para realizar ataques *Cross Site Scripting*, XSS y comprometer el sistema. Por ejemplo, como el contenido de la variable puede ser el resultado de un formulario web, ahí es donde dicho usuario podrá aprovechar la brecha de seguridad para incrustar código maligno.

Para realizar dicho control, Django aporta por defecto un sistema de seguridad para escapar el código HTML, con el fin de ayudar al programador para que no tenga que centrarse en esta tarea.

En el caso contrario, a veces, puede resultar necesario que sí se quiera permitir la introducción de código HTML, por lo que será necesario desactivar dicho escape automático. Algunos ejemplos que sea necesario deshabilitar esta opción son: que el programador guarde en la base de datos algo de código de HTML y quiera mostrarlo, o si se utiliza el sistema de plantillas para generar texto distinto a HTML, por lo que no quiere que Django escape dicho código. Es posible deshabilitarlo tanto para variables

individuales como para bloques de plantillas. Para las variables individuales, es tan sencillo como añadir el filtro *{{ nombre_variable|safe }}*. Para los bloques de plantillas, es necesario añadir la etiqueta *{% autoescape off / on %}* tanto si se quiere habilitar o deshabilitar.

```
{% if no_peli.trailer %}
      <p> <b>Trailer de {{ no_peli.titulo }}</b></p>
      {% autoescape off %}{{ no_peli.trailer }}{% endautoescape %}
{% endif %}
```

11.6 Herencia de plantillas

Siguiendo su principal objetivo de evitar la duplicidad y redundancia de código, Django ofrece un potente sistema de herencia para las plantillas. Gracias a este sistema, no es necesario que se cree una plantilla personalizada para cada parte de la aplicación que se está desarrollando incluyendo el mismo fragmento de código en muchas de ellas.

Para aprovechar al máximo esta herramienta, lo más conveniente es crear una plantilla base, que será el padre de todas y tendrá el código y la estructura que tendrán sus hijas. De esa plantilla base heredarán distintas plantillas hijas que contendrá cada una su propio contenido siguiendo su estructura. Para dividir el contenido de una plantilla, se utilizarán bloques que serán definidos en la plantilla base con el objetivo de que puedan personalizarse por las hijas.

Para definir dichos bloques se utilizan las etiquetas "*{% block nombre_bloque %}*" y "*{% endblock %}*", donde entre ellas se situará el código a introducir o simplemente se declara vacío para que cada hija introduzca su parte. En cuanto a las plantillas hijas cabe comentar que deben de empezar incluyendo "*{% extends plantilla_padre.html %}*" para indicar a Django que tiene que cargar la plantilla base. Una vez que Django carga la plantilla padre, detecta las etiquetas "*block*" y procede a realizar la sustitución correspondiente de los bloques con el contenido de las plantillas hijas.

Las plantillas que heredan de la base no tienen necesariamente que definir todos los bloques de ella. En ese caso, se mostrará el contenido por defecto que tendrá la plantilla padre.

A continuación, se muestra un ejemplo de herencia de plantillas, junto con la definición de parte de los bloques definidos en la plantilla padre:

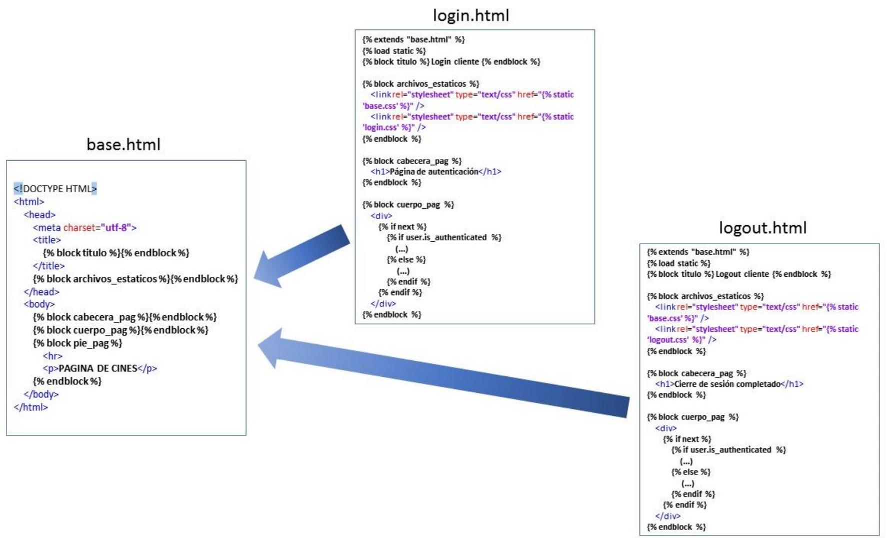


Ilustración 9 - Ejemplo de herencia de plantillas.

11.7 Incluyendo otras plantillas

Otra forma distinta que Django permite a los desarrolladores reducir notablemente las líneas del código es mediante la inclusión de una plantilla en otra. Esto se puede realizar mediante la introducción del siguiente código en la plantilla donde se desea añadir el contenido: *{% include "plantilla.html" %}* o bien, *{% include nombre_plantilla %}*, siendo *nombre_plantilla* una variable que contiene el nombre de la plantilla a incluir.

La plantilla que se indica en el *include* será evaluada en el mismo contexto que la principal, por defecto. Es posible, tanto añadir nuevos valores al contexto, mediante la opción *with*, o incluso que se evalúe con los parámetros que se le digan, incluyendo la *only* al llamar a la etiqueta *include* de la plantilla.

Como se puede imaginar esto permite un gran ahorro de código al programador ya que, si un determinado fragmento HTML se repite en numerosas páginas, solo será necesario que realice la inclusión de dicha plantilla.

11.8 Contexto y renderizado de plantillas

Como se ha comentado anteriormente, las plantillas tienen una serie de variables y etiquetas de bloques que son necesario resolver y evaluar para determinar la plantilla final que será mostrada en el navegador del cliente. Para realizar dichas operaciones se utilizan los contextos en las plantillas. Un contexto por lo tanto será una equivalencia entre las distintas variables y su valor correspondiente, que se corresponderá con un diccionario en Python. Se representan mediante la clase *Context* del módulo *django.template*. Es en las vistas donde se asignará un contexto a una plantilla para cuando se haga el render de la plantilla, se sustituya cada variable por su valor.

11.8.1 Procesadores de contexto

Aparte de que el contexto sea una instancia de *django.template.Context*, también puede ser de *django.template.RequestContext*, lo cual cambiará ligeramente la funcionalidad. Esta clase aporta diferentes variables al contexto tales como un objeto *HttpRequest* y algunos datos del usuario de la aplicación en ese momento, entre otras posibilidades.

Mediante el uso de *RequestContext* junto con los procesadores de contexto, el programador podrá evitar numerosas líneas de código repetidas ya que permitirán incluir un determinado número de variables en diferentes plantillas sin tener que volverlas a declarar en cada una. Un procesador de contexto será una función Python que tomará como argumento un objeto *HttpRequest* y retorna el diccionario con las variables que se van a incluir automáticamente en el contexto para cada plantilla que lo ejecute, según la opción de configuración del procesador de contextos del motor. Dicha opción será una lista de procesadores de contextos, que se encuentra en *settings.py*, el archivo de configuración que se crea por defecto con el proyecto Django. Vienen los siguientes procesadores de contexto cuando se crea una aplicación nueva con el motor de plantillas por defecto *DjangoTemplates*:

- '*django.template.context_processors.debug*'
- '*django.template.context_processors.request*,
- '*django.contrib.auth.context_processors.auth'*
- '*django.contrib.messages.context_processors.messages'*

Django viene con algunos procesadores de contextos incluidos con un determinado número de variables cada uno de ellos. Se pasa a crear una tabla para nombrar cada uno:

Procesador de Contexto	Variables
django.contrib.auth.context_processors.auth	user, perms
django.template.context_processors.debug	debug, sql_queries
django.template.context_processors.i18n	LANGUAGES, LANGUAGE_CODE
django.template.context_processors.media	MEDIA_URL
django.template.context_processors.static	STATIC_URL
django.template.context_processors.csrf	Facilita un token para la etiqueta csrf_token.
django.template.context_processors.request	request
django.template.context_processors.tz	TIME_ZONE
django.contrib.messages.context_processors.messa ges	messages, DEFAULT_MESSAGE_LEVE LS

Tabla 13 - Tabla de procesadores de contexto de Django.

Si se usan estos procesadores de contexto de Django, cada *RequestContext* tendrá las variables propias de cada uno de ellos

Django permite escribir procesadores de contextos personalizados con el objetivo de poder incluir otras variables no contempladas en los casos anteriores.

Por lo tanto, una vista, a la hora de definir el procesador a utilizar, deberá de indicar que se trata de *RequestContext*, al que pasará los siguientes argumentos:

- Un objeto *HttpRequest* en primer lugar.
- Un diccionario Python por si se desea integrar alguna variable adicional.
- Como parámetro opcional, recibirá una lista de funciones de procesadores de contexto adicionales que usará. Se guarda en la variable *processors*.

11.8.2 Renderizado de plantillas

Con el contexto creado y la plantilla correctamente implementada, es necesario relacionar una plantilla con un contexto para que se produzca la traducción de variables y obtener la plantilla final. Esto se realiza mediante el proceso de renderizado de plantillas. Por lo tanto, una vez que se tiene una instancia de la clase *Template*, que será la plantilla y un determinado contexto, es necesario llamar a la función *Template.render()* de la plantilla, pasándole como argumento el contexto dado. Con todo ello, quedaría la plantilla formada correctamente habiéndose traducido las variables y evaluado las etiquetas de los bloques.

Django a modo de atajo aporta el método *render()* del módulo *django.shortcuts* el cual permite realizar lo anteriormente comentado en una sola línea de código, lo que ayuda a reducir código al programador. Dicho atajo que aporta Django permite realizar en una sola llamada:

- La carga de la plantilla, es decir, evita tener que llamar *a loader.get_template().*
- La generación de un contexto para esa plantilla.
- Por último, evita tener que realizar el proceso de renderización de la plantilla por separado.

Por lo que los argumentos que recibirá dicha función serán: la plantilla a utilizar en primer lugar y un diccionario de forma opcional de segundo argumento. El contexto creado por este atajo por defecto será un *RequestContext*. Por último, cabe comentar que devuelve un *HttpResponse*, por lo que se puede retornar en la función vista.

A continuación se pasa a mostrar el código para comprobar el código que permite ahorrarse el método *render()*:

- Código sin atajo.

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse

def cliente(request):
	no_usuario = request.user
	id_usuario = get_object_or_404(Cliente, user = no_usuario)

	lista_entradas_futuras = Entrada.objects.filter(id_cliente =
id_usuario).filter(fechaHora__gte=datetime.datetime.now()).order_by(
'id_entrada')
	lista_comentarios = Comentario.objects.filter(usuario =
id_usuario).order_by('pelicula')

	template_cliente = get_template('usuario.html')
	html_res =
template_cliente.render(Context({'usuario':id_usuario,
'entradas_f':lista_entradas_futuras,
'comentarios':lista_comentarios}))

	return HttpResponse(html_res)
```

- Código con atajo *render()*.

```
from django.shortcuts import render, get_object_or_404
def cliente(request):

	no_usuario = request.user
	id_usuario = get_object_or_404(Cliente, user = no_usuario)

	lista_entradas_futuras = Entrada.objects.filter(id_cliente =
id_usuario).filter(fechaHora__gte=datetime.datetime.now()).order_by(
'id_entrada')
	lista_comentarios = Comentario.objects.filter(usuario =
id_usuario).order_by('pelicula')

	return render(request, 'usuario.html', {'usuario':id_usuario,
'entradas_f':lista_entradas_futuras,
'entradas_p':lista_entradas_pasadas,
'comentarios':lista_comentarios})
```

Otro atajo que ofrece Django es *django.shortcuts.render_to_response()*. Éste es el antecesor del anterior, *render()*. Funciona de la misma manera salvo que la solicitud, el objeto *request*, no está disponible en la respuesta. Por esta casuística que se acaba de comentar, la función *render_to_response()* tenderá a desaparecer en futuras versiones de Django.

Es necesario llevar a cabo este proceso de renderización de plantillas, debido ya que una de ellas por sí sola no puede asignar o modificar el valor de una variable, para ello se han creado los contextos.

Django permite que una misma plantilla se pueda utilizar con múltiples contextos llevando a cabo el renderizado con cada uno de ellos. Esta ventaja que aporta Django sirve para evitar tener una duplicidad de código, lo que concuerda con su principio básico DRY.

11.9 Incluir funcionalidad extra a las plantillas

Por un lado, es posible a las plantillas de Django darles estilo, es decir, usar una hoja de CSS3. Con ello se podrá personalizar al máximo una plantilla para mostrar de la forma más clara posible la información. Para ello, se creará una hoja CSS3 con los estilos que se deseen aplicar a una plantilla**.**

Para añadir las hojas de estilo a una determinada plantilla bastará con incluir:

```
    <link rel="stylesheet" type="text/css" href="{% static
'base.css' %}" />
    <link rel="stylesheet" type="text/css" href="{% static 'add-
cliente-ok.css' %}" />
```

Aparte de poder estilizar las plantillas de Django, es posible aumentar la funcionalidad de la página y hacerla más dinámica con la inclusión de JavaScript y Ajax. Django permite la inclusión de dichas tecnologías en sus plantillas de tal forma que el programador, pueda realizar páginas web lo más completas posibles.

Por ejemplo, para incluir JavaScript en ellas bastará con introducir el siguiente fragmento en la plantilla:

```
<script>
      window.onload = desactiva_ocupados();

      function desactiva_ocupados() {
            var lista = {{ lista }}
            //codigo de la función
      }
</script>
```

Como se puede ver, es posible introducir variables DTL en dicho código con la finalidad de poder extender la utilidad de JavaScript. Esto es posible ya que Django realiza la sustitución de la variable por dicho valor al renderizar la página. También es posible llamar a archivos "*.js*" al igual que con las hojas de estilo.

El código anterior es sólo un ejemplo de la integración con JavaScript, pero es posible crear código mucho más completo y con más funcionalidad

También es posible integrar Ajax (*Asynchronous JavaScript And XML*), que puede entenderse como una extensión de JavaScript, con la finalidad de intercambiar información con el servidor y no tener que recargar la página completa cada vez.

Ilustración 10 - Logotipo de Ajax

12 Formularios en aplicaciones web

Un apartado muy importante en el ámbito de la programación Web son los formularios, ya que permiten al usuario interactuar con la aplicación. Por ejemplo, el usuario podrá introducir texto, elegir entre distintas opciones, entre muchas más posibilidades. Con los formularios HTML, se podrán crear objetos en las tablas de la base de datos, así como solicitar cualquier tipo de información de la misma.

En este capítulo se pasa a comentar cómo se encarga Django de gestionar los formularios web.

12.1 Métodos del formulario

En un formulario web existen dos tipos de métodos que se podrán utilizar para trabajar con ellos, dependiendo su uso final. Dichos métodos HTTP son: GET y POST. Estos métodos serán los únicos que se pueden utilizar en los formularios, y cada uno de ellos tiene una manera de actuar diferente:

- GET: envía los datos de forma visible al usuario ya que se mandan los datos agrupados en una cadena en la URL. Suele ser utilizado para formularios de búsqueda en la web y no se debería de usar para formularios de inicio de sesión ya que la contraseña viajaría en claro por la URL.
- POST: Envía los datos de forma invisible al usuario y suele ser utilizado para solicitudes que realicen cambios en el sistema, como, por ejemplo, un *login* de usuario, un formulario de petición de una gran cantidad de datos, entre otros.

En cuanto a la seguridad de la aplicación, es muy importante tener en cuenta que método HTTP se utiliza, para que usuarios maliciosos no puedan comprometer el sistema ni suplantar la identidad de otros usuarios por errores en el manejo de los métodos de formularios.

De forma simplificada, se pasa a mostrar un diagrama de peticiones GET y POST entre el usuario y el servidor, con la finalidad de poder entender el intercambio de mensajes que se produce:

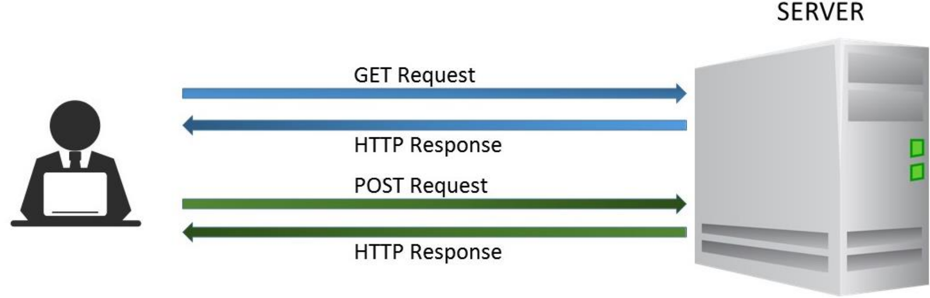


Ilustración 11 - Diagrama de petición – respuesta de métodos GET y POST de HTTP.

12.2 Formularios en Django

El manejo de formularios es una tarea compleja debido a la gran cantidad de casuísticas para tener en cuenta. Es por ello por lo que Django ofrece un conjunto de herramientas y librerías para trabajar con estos, de tal forma que se podrán realizar tareas como: generar formularios a partir de un determinado modelo o a partir de reglas específicas, validar dichos formularios y mostrar errores donde proceda y obtener en claro la información recibida por los mismos para realizar operaciones con ellos.

No es estrictamente necesario utilizar dichas herramientas para el manejo de formularios con este Framework de programación, pero si facilita mucho esta tarea. Por lo que, si un programador desea no utilizarlo, Django no pone ningún impedimento para ello.

En cuanto a los formularios en las aplicaciones Django, por defecto, en el sitio de administración, vienen incluidos formularios para dar de alta nuevos objetos, así como eliminarlos o modificarlos, siempre que se haya registrado el modelo en el sitio de adminsitración. Para el resto de la aplicación, se encargará el programador de utilizar las herramientas proporcionadas para generar los formularios según sus requisitos.

En la documentación oficial de Django, se indica que por convención se debe de situar el código relacionado con los formularios en un archivo llamado *forms.py* dentro de la carpeta de la aplicación, junto con *models.py* y *views.py* entre otros. No es una regla estricta, pero es recomendable para tener el código estructurado lo más limpio y estructurado posible. Con ello se podrá tener el código de los formularios separado del resto de clases de la aplicación. En ese archivo *forms.py* se debe de incluir la siguiente línea en primer lugar: "`from django import forms`" y a continuación importar los modelos que se vayan a utilizar en los formularios**.** Al igual que en el archivo de los formularios se importan los modelos para poderlos utilizar en los mismos, cuando sea necesario utilizar los formularios, habrá que importarlos, por ejemplo, en *views.py*.

12.2.1 Form y ModelForm

En cuanto a las clases de los formularios, existen dos que serán comúnmente utilizadas para crearlos y trabajar con ellos. Éstas clases son *django.forms.Form* y *django.forms.ModelForm*. La primera de ellas se puede considerar como el centro de la creación de formularios ya que será donde se implemente el propio formulario junto con su funcionamiento y otras opciones que se irán comentando.

Los formularios en Django serán subclases de *forms.Form*, en donde se definirán los *inputs* que tendrá el formulario mediante campos en dicha clase, los cuales realizarán una validación de los datos del formulario. Hay distintos tipos de campos en Django, los cuales se muestran a continuación:

Campo	Widget por defecto
BooleanField	CheckboxInput
CharField	TextInput
ChoiceField	Select
TypedChoiceField	Select
DateField	DateInput

DateTimeField	DateTimeInput
DecimalField	NumberInput
DurationField	TextInput
EmailField	EmailInput
FileField	ClearableFileInput
FilePathField	Select
FloatField	NumberInput
ImageField	ClearableFileInput
IntegerField	NumberInput
GenericIPAddressField	TextInput
MultipleChoiceField	SelectMultiple
TypedMultipleChoiceField	SelectMultiple
NullBooleanField	NullBooleanSelect
RegexField	TextInput
SlugField	TextInput
TimeField	TextInput
URLField	URLInput
UUIDField	TextInput

Tabla 14 - Tabla de campos para un formulario con sus widgets.

Si dichos tipos de campos, no satisfacen los requisitos del problema, es posible generar campos personalizados mediante subclases de *django.forms.Field* que implementen un método *clean()* y que *__init__()* reciba los siguientes argumentos: *required, label, initial, widget* y *help_text*.

Cada campo de la tabla anterior recibe unos argumentos específicos descritos en documentación oficial, pero existe una lista de argumentos que son comunes para todos ellos. Dichos argumentos son:

Argumentos	Descripción
required	Valor booleano que indica si el campo es obligatorio rellenarlo en el formulario.
label	Etiqueta que se mostrará para ese campo en el formulario.
label_suffix	Sufijo que se mostrará con la etiqueta
initial	Valor inicial para un campo en el formulario.
widget	Especifica el Widget que se usará al renderizar el campo.
help_text	Texto descriptivo para un campo.
error_messages	Sobrescribe los mensajes de error para el campo.
validators	Permite añadir una lista de validadores para ese campo.
localize	Permite la localización de la entrada de datos del formulario, así como la salida renderizada.
disabled	Permite deshabilitar o no un campo del formulario para que los usuarios no puedan editarlo.

Tabla 15 - Tabla de argumentos de campos.

Con estos argumentos comunes Django permite al programador personalizar tal y como desee el comportamiento de cada campo de los formularios que tendrá su aplicación.

En cuanto a la visualización de los campos del formulario en el HTML resultante, entran en lugar los *Widgets*. Cada campo comentado anteriormente tiene un *Widget* asociado por

defecto, el cual, determinará la forma en la que se verá el correspondiente *input*, ya que se identifica con un *widget* de formulario HTML. Es posible modificarlos si se requiere utilizar uno diferente al que viene por defecto. Para ello se debe de indicar en el atributo '*widget*' en el campo del formulario, como se ha comentado en la tabla anterior.

Se muestra a continuación un ejemplo de un formulario:

```
class ComentariosForm(forms.Form):
      comentario = forms.CharField(widget=forms.Textarea)
```

Otra opción muy potente que ofrece Django es la posibilidad de construir un formulario asociado a un modelo de la aplicación. Para ello se utilizará la clase *django.forms.ModelForm*. Es necesario indicar el modelo que tendrá asignado para una subclase de *ModelForm*, ya que de esta forma se encarga automáticamente de mapear los campos de ese modelo a los elementos *input* de HTML como un formulario. Por lo tanto, cuando se requiera generar un formulario dependiente de un modelo de la aplicación, *ModelForm* facilitará dicho trabajo a los programadores, ya que no será necesario replicar los campos del modelo en el formulario. Esta característica que se acaba de comentar confirma que Django sigue el principio DRY (*don't repeat yourself*) ya que no tendría el programador que repetir el mismo código. Este último tipo de formularios son los utilizados en el sitio del administrador de Django cuando trabaja con los modelos que tiene registrados.

En cambio, si se desea crear un formulario que no está relacionado directamente con un modelo, se deberá de generar una subclase de *Forms*.

Se muestra un ejemplo de un *ModelForm* completo.

```
class UserForm(forms.ModelForm):
      password          =         forms.CharField(label='Contraseña',
widget=forms.PasswordInput)
      password_confirm    =    forms.CharField(label='Introduzca    la
contraseña de nuevo', widget=forms.PasswordInput)

      class Meta:
            model = User
            fields = ('first_name', 'last_name', 'email', 'username',
'password')
```

Otra gran funcionalidad que ofrece el framework, tanto para *forms.Form*, como para *forms.ModelForm*, es la posibilidad de agrupar varios de ellos en un determinado *formset*. Un *formset* es una capa de abstracción que ofrece Django, la cual permite tener diferentes formularios en una misma página. De esta forma se podrá iterar por ellos e irlos mostrando como el programador estime oportuno.

12.2.2 Formularios en las vistas

Las vistas de Django, como se comentó en capítulos anteriores, son las encargadas de la lógica de qué datos se van a presentar al usuario de la aplicación. Por lo tanto, estas vistas

también serán las delegadas de la publicación y el procesamiento los formularios del archivo *forms.py* generados por los programadores.

Dependiendo del método que tenga un determinado formulario, ya sea GET o POST, habrá que tenerlo en cuenta la forma de instanciarlos correctamente. Hay dos posibilidades:

- Si es una solicitud GET, se debe de crear en la vista una instancia del formulario vacía y se debe de introducir en el contexto de la plantilla.
  ```
  Form = NameForm()
  ```
- Si es una solicitud POST, se crea una nueva instancia del formulario rellenándolo con los datos del formulario que se indica.
  ```
  form = NameForm(request.POST)
  ```

Una vez que se tiene el formulario recogido en la vista, si el método es POST, el siguiente paso que debe de realizar antes de operar con los datos recogidos en el formulario, es comprobar que los campos son válidos. Los formulario tienen un método que se denomina *is_valid()*, el cual sirve para comprobar si los datos introducidos por el usuario son correctos acorde a los requisitos introducidos por el programador en la declaración del formulario y de sus campos. Cuando se produce la llamada a este método y los datos son válidos, devuelve *true* y sus datos validados se almacenan en un diccionario de Python, *form.cleaned_data*. De ese diccionario el programador podrá extraer los datos en claro introducidos por el usuario en el formulario para procesarlos y poder realizar operaciones con ellos, como, por ejemplo, de inserción o de modificación de registros. Se puede incluir el método *is_valid()* en un bloque *if/else*, de tal manera que cuando no sea válido, vacíe el formulario o cualquier acción que el programador estime oportuno.

A continuación, para mostrar el uso de lo comentado anteriormente, se muestra un ejemplo de un formulario POST, del que se extraen los datos recibidos y se crea un objeto nuevo:

```
#formulario
      form = ComentariosForm(request.POST or None)
      if form.is_valid():
            form_data = form.cleaned_data

            comen = (form_data.get("comentario"))
            fx_comentario = datetime.datetime.now()
            usuario = None
            if request.user.is_authenticated():
                  usuario = request.user
                  no_usuario = get_object_or_404(Cliente, user =
usuario)
            new_comen = Comentario.objects.create(comentario = comen,
pelicula = item_peli, usuario = no_usuario, fecha_comentario =
fx_comentario)
```

12.2.3 Renderización de formularios

Una vez que se ha definido un formulario en el fichero *forms.py* y publicado en una plantilla por una función vista, el último paso es mostrarlo en un *template* para que el usuario final pueda utilizarlo.

Como nota importante, cabe comentar que todos los campos y atributos del formulario están recogidos en la variable que recibe de la vista. Comúnmente tendrá el siguiente nombre*: {{ form }}*. Django se encargará de procesar sus elementos *<label>* y *<input>* automáticamente, aunque, si el programador lo desea, puede procesarlos el a mano para llevar un mayor control. Esto último puede ser útil si se desea reordenar los campos del formulario.

Existen otras opciones para mostrar los formularios recibidos de las vitas. Estas opciones que ofrece Django son las siguientes:

Opción	Descripción
{{ form.as_table }}	Renderiza el formulario como celdas de una tabla, <tr>
{{ form.as_p }}	Renderiza el formulario como etiquetas de párrafo, <p>
{{ form.as_ul }}	Renderiza el formulario como elementos de una lista no ordenada.

Tabla 16 - Tabla de opciones de renderizado de formularios.

Por todo ello, se puede deducir que será necesario declarar el formulario en la plantilla con la etiqueta *<form>* así como el botón de tipo *submit* cuando sea necesario ya que Django no lo realiza de forma automática, dejando al programador realizar dicha tarea.

Si se declara un formulario de método POST, es necesario introducir "*{% csrf_token %}*" ya que si no Django devolverá un error de que falta añadir dicha sentencia. Esto es porque Django tiene una protección contra CSRF (*Cross Site Request Forgeries*) para proteger un sitio web frente a usuarios maliciosos que introducen comandos no autorizados y pueden provocar acciones no deseadas por los administradores de la aplicación. Se muestra un ejemplo de lo comentado en este apartado:

```
<form method="POST" action="">{% csrf_token %}
      {{ form.as_p }}
<input type='submit' value='Reservar'>
```

Como se ha comentario con anterioridad, también es posible renderizar un formulario manualmente con el objetivo de poder tener un mayor control sobre los campos y de los errores. Se muestra un ejemplo del formulario anterior renderizado de esta manera:

```
<form method="POST" action="">{% csrf_token %}
      {% for field in form %}
            <div class="campo_block">
                  {{ field.errors }}
                  {{ field.label_tag }} {{ field }}
                  {% if field.help_text %}
                        <p    class="help">{{    field.help_text|safe
}}</p>
                  {% endif %}
            </div>
            {% endfor %}
      <input type='submit' value='Reservar'>
</form>
```

Donde se puede ver que se podrá mostrar en el orden que se desee el campo junto con sus distintos atributos como son *field.errors, que* indicará los errores de validación para ese campo específico, *field.label_tag*, será la etiqueta del campo, *help_text*, el texto de ayuda, entre otros muchos atributos de *field* disponibles.

12.2.4 Validación de formularios

Otro aspecto importante en cuanto a los formularios es la validación de los mismos. Es decir, que cada campo contenga lo que el programador requiera y cumpla de esa manera con los requisitos de la aplicación.

Django en este tema sigue una validación a distintos niveles del formulario. En un primer lugar, comienza validando a nivel de campo. En este nivel, cabe comentar que cada campo viene incluido con un validador por defecto, que será el correspondiente al tipo de campo que sea. Aparte de dichos validadores por defecto, se pueden añadir otros adicionales, tanto de los que ofrece Django, como validadores personalizados para que sigan los campos con una determinada estructura no recogida en los que vienen por defecto. Para incluir dichos validadores adicionales, se utiliza el argumento *validators*. Este tipo de validación a nivel de campo devuelve un *ValidationError* si falla, en caso contrario, no devuelve nada.

Después de esa primera validación que realiza Django sobre los campos, es posible añadir otra a nivel de campo adicional. Para ello, dentro de la clase del formulario, es necesario declarar un método que se debe de llamar "*clean_"* seguido del nombre del campo asociado. Por ejemplo, *clean_miCampo(self).* Dicho método devolverá un *ValidationError* cuando se cumpla alguna condición que imponga el programador. Como norma general, aparte de devolver el mensaje de error anterior, debe de devolver siempre el valor resultante de la llamada a *self.cleaned_data* del campo correspondiente.

Por último, a nivel de formulario es posible sobrescribir el método *clean()*. Con ello, es posible mostrar mensajes de error al principio del formulario devolviendo un *raise forms.ValidationError*, teniendo en cuenta varios de los campos del formulario en cuestión. Aparte de dicho mensaje de error general, se pueden mostrar mensajes de error individuales para cada campo mediante *self.add_error(campo, mensaje)*.

De una forma reducida, este sería el proceso por el cual un formulario es validado. Cuando se produce una llamada al método *is_valid()* de una determinada vista para un formulario, surge lo siguiente:

I. En un primer lugar se validan los campos del formulario con sus validadores que tienen asociados. De esta manera, se llevan los datos parcialmente procesados para los siguientes pasos.
II. Django busca si se ha definido el método *clean()* para un determinado campo, y si es correcto lo analiza.
III. Por último, se realiza una llamada al método *clean()* a nivel de formulario.
IV. Si todos los pasos anteriores son satisfactorios, el formulario se considera correcto. En caso contrario, se mostrarían los mensajes de error y se llevan a cabo las acciones que el programador haya estimado oportunas.

Cuando se llama a *is_valid()* se produce una llamada a *clean()* de cada campo si existe y antes a los validadores de campos ya que necesariamente deben de existir para seguir con la validación, es decir, están parcialmente procesados.

De esta forma se asegura que no se almacena código incorrecto en la base de datos, que pueda desencadenar robo de información y otras situaciones no deseadas.

12.2.5 Resumen procesamiento formulario

En este apartado se pretende realizar un resumen de la creación de formularios para un sitio web, para conocer las relaciones de los distintos apartados anteriores.

En un primer lugar, es necesario escribir los formularios en el archivo *forms.py*. Estos formularios podrán estar asociados a un modelo, es decir, *ModelForm*, o ser formularios simples, *Form*. En dichos campos creados, se podrán incluir los validadores deseados o definir un método *clean* para un campo específico o en último caso un método *clean()* para el formulario. Una vez creado el formulario con sus validaciones en dicho archivo, se pasa a trabajar sobre la vista en la que publicará el formulario en una plantilla tras procesarlo. Para mandarlo a la plantilla, es necesario incluirlo en su contexto. Mediante la función vista, se comprobará la validez del formulario, y si todo es correcto, se realizan las operaciones oportunas. Por último, hay que recuperar el formulario enviado por la vista en la plantilla mediante variables, para posteriormente poder elegir la manera en la que se muestra al usuario final. Para ello, se recorrerán los campos del formulario o se usa alguno de los atajos definidos en los puntos anteriores.

De esta manera resumida, quedaría el formulario disponible para utilizarse en esa plantilla de la aplicación.

13 Usuarios en Django

En las aplicaciones Web un aspecto muy importante debido a su alto uso es la gestión de los usuarios que acceden a ellas. Es necesario tener un control de los mismos para evitar que usuarios no legítimos accedan a la parte de administración, que accedan a partes prohibidas por el administrador, entre otras muchas opciones de seguridad.

Django trae por defecto un sistema de autenticación de usuarios con el cual se pueden gestionar sus cuentas y accesos a determinadas partes de la aplicación junto con la de los administradores de la misma, pudiendo crear grupos de usuarios con un conjunto específico de permisos, y también maneja las sesiones de usuarios basadas en cookies. Además, proporciona un sistema configurable de *hashing* de contraseñas con el objetivo de mantenerlas almacenadas de forma segura en la base de datos de la aplicación.

Como se puede observar, el framework de autenticación de Django proporciona al programador de forma genérica herramientas para dicha gestión de los usuarios, quedando fuera de su alcance, implementaciones específicas de seguridad como pueden ser, conocer la fuerza de una contraseña, el número de intentos de inicio de sesión. Algunas de estas implementaciones se encuentran desarrolladas por terceros. Un ejemplo puede ser *django-axes*, que es un paquete con el que es posible controlar los intentos de inicio de sesión que ha realizado un usuario y su resultado ha sido negativo, por usuario o contraseña incorrecta. Otro ejemplo es *django-two-factor-auth*, con el cual es posible disponer de autenticación con doble factor en el login de Django.

Este sistema se encuentra en el módulo '*django.contrib.auth'*. No es necesario realizar ninguna configuración previa para usar dicho módulo, ya que al generar el proyecto con el comando *startproject*, se incluye tanto el *Middleware* de sesiones y de autenticación, *'SessionMiddleware'* y '*AuthenticationMiddleware'* respectivamente, así como los paquetes de '*django.contrib.auth'* y '*django.contrib.contenttypes'*. Este último es el sistema de contenido de Django, con el cual se pueden asignar permisos a los modelos creados en la aplicación. Todo ello se encuentra en el archivo de configuración *settings.py*.

Este sistema que Django ofrece por defecto a los programadores incluye una serie de modelos, vistas y plantillas con los que se podrá trabajar sin tener que preocuparse de su implementación interna, a no ser que se quiera extender su funcionalidad. Aparte de lo anterior, también incluye un sistema de gestión de permisos simple para gestionar los de cada usuario.

Como se ha comentado en el apartado anterior, es posible ir más allá y realizar un sistema de autenticación personalizado, ya que algunos puntos son extensibles y otros reemplazables. Todo ello se irá comentando a lo largo de este capítulo.

13.1 Tipos usuarios disponibles en Django

Es posible tener en una aplicación numerosos tipos de usuarios debido a los permisos que tengan asignados cada uno. Pero de una forma general, en Django, nos podemos encontrar con estos tres tipos de usuarios:

- **User**
  Este tipo de usuarios son una instancia de la clase *django.contrib.auth.models.User*. Esta clase se puede considerar como la clase de la que parten los otros tipos de usuarios, ya que se pueden entender como implementaciones especiales de la misma. En los próximos apartados se comentarán particularidades de esta clase, así como sus atributos y métodos.
- **AnonymousUser**
  Será una instancia de la clase *django.contrib.auth.models.AnonymousUser*. Dicha instancia será una implementación particular de la clase *django.contrib.auth.models.User*. Por ejemplo, algunas particularidades que posee dicha clase son:
  - No tiene nombre de usuario ni contraseña, por lo que el campo *username* será siempre nulo y el método *set_password()* y *check_password()* no los tendrá implementados.
  - No pertenece a ningún grupo y no posee ningún permiso especial. Por lo que los campos *groups* y *user_permissions* serán vacíos y *is_active*, *is_staff* y *is_superuser* estarán a False.
- **Superuser**
  Superuser es como el primer caso, *User*, pero tiene unas características especiales. Los campos *is_staff* y *is_superuser* los tiene a *True*, por lo que es lo que le marca la diferencia respecto a otro usuario de la aplicación. Este tipo de usuarios son los que tendrán permisos para acceder a la parte de administración de Django.

En el paquete *django.contrib.auth.models*, aparte de las clases anteriores, Django incluye también *AbstractUser*, *AbstractBaseUser*. El objetivo de dichas clases es poder extender el sistema de gestión y autenticación de usuarios que trae por defecto Django, ya que con esas clases se pueden modificar diversos comportamientos según las necesidades del programador. La principal diferencia entre ellas es que la primera se correspondería con la clase *User* abstracta completa, es decir, tiene sus correspondientes campos y métodos. De dicha clase se podrá heredar de ella y, por lo tanto, añadir nuevos campos para el perfil del usuario, así como, métodos adicionales. La clase *AbstractBaseUser* en cambio, solo posee la funcionalidad de autenticación, por lo que, mediante subclases, será necesario añadir los campos que tendrán los usuarios y los métodos que utilizarán.

Otra opción de extender la clase *User* que proporciona Django, es crear un campo en un modelo propio de la aplicación, que será el perfil del usuario, que sea una relación *OneToOne* a la clase *User*. Con ello existirá una correspondencia uno – a – uno entre el perfil y la instancia de dicha clase. De esta manera, se pueden agregar campos adicionales al modelo del perfil, como por ejemplo un DNI, fecha de nacimiento, imagen de perfil, entre otras. Con esta opción, es posible utilizar todos los métodos que aporta la clase *User* de Django, sin tener que preocuparse de implementarlos manualmente.

A continuación, se muestra este último caso:

```
class Cliente(models.Model):
      user = models.OneToOneField(User, on_delete=models.CASCADE)
      fecha_nacimiento = models.DateField('Fecha de nacimiento',
auto_now = False, auto_now_add = False, blank = True, null = True)
```

13.2 Sistema de autenticación en Django

Existe una clase, *models.User* del paquete '*django.contrib.auth*', comentada en el punto anterior, que se puede considerar como el centro de este sistema de autenticación. Los usuarios existentes en la aplicación son instancias de esa clase, pero con distintas configuraciones, según el tipo de usuario que sea, tal y como se ha comentado con anterioridad.

La clase *User* tiene una serie de campos, unos obligatorios y otros no, que permitirán definir un usuario para este Framework. Se pasa a comentar cada uno de ellos:

Campo	Descripción
username	Campo obligatorio que es el nombre de usuario con el que se identificará en la aplicación.
first_name	Nombre del usuario.
last_name	Apellidos del usuario.
email	Almacena la dirección de correo.
password	Campo obligatorio que es la contraseña del usuario. La usará para loguearse en la aplicación.
groups	Relación muchos a muchos con '*models.Group*'.
user_permissions	Relación muchos a muchos con '*models.Permission*'.
is_staff	Establece si el usuario pertenece al staff, por lo tanto, si puede acceder a la parte de administración.
is_active	Sirve para permitir a un usuario o no loguearse en la aplicación sin necesidad de darle de baja de la misma.
is_superuser	Indica si ese usuario es superusuario, es decir, si tiene todos los permisos.
last_login	Recoge la fecha del último acceso del usuario al sistema.
date_joined	Indica la fecha de creación de la cuenta del usuario.

Tabla 17 - Tabla de campos de la clase User.

Aparte de dichos campos que posee la clase *User*, también tiene una serie de atributos y de métodos, donde pasamos a comentar los más destacables. Comenzando por los atributos:

- *is_authenticated*. Es posible averiguar si un usuario se encuentra autenticado actualmente en la aplicación mediante este valor booleano.
- *is_anonymous*. Si el usuario actual es anónimo o no, es decir, si es una instancia de *User* o de *AnonymousUser*.
- *username_validator*. Apunta a una instancia del validador utilizado para los nombres de usuarios.

En cuanto a los métodos, hay un gran número de ellos en los que destacamos los siguientes:

- *get_username()*. Devolverá el nombre del usuario.
- *set_password()*. Establece la contraseña pasada por parámetro para el usuario.
- *check_password()*. Valor booleano que devuelve *True* si es la contraseña correcta para el usuario.
- *email_user()*. Envía un correo al usuario. Se le pasa como parámetro el asunto, mensaje y el correo desde el que se envía.

La clase *User* posee un *Manager*, *models.UserManager*, que tiene métodos de ayuda aparte de los que añade *BaseUserManager*, como pueden ser la creación de usuarios y superusuarios. Estos métodos son *create_user* y *create_superuser* y serán utilizados cuando se desee crear un usuario para la aplicación, así como, un superusuario de la misma.

13.2.1 Backends de autenticación en Django

Django, viene incorporado con una serie de *Backends* de autenticación predefinidos, como pueden ser: *ModelBackend*, *AllowAllUsersModelBackend* y *RemoteUserBackend*, entre otros. Todos ellos se encuentran en '*django.contrib.auth.backends'*. Estos *Backends* se encargan de autenticar al usuario de la aplicación para poder acceder a su perfil de la aplicación. Este proceso busca en la lista: *AUTHENTICATION_BACKENDS*, situada en el archivo de configuración *settings.py*, donde estarán los *Backends* que serán usados en la aplicación. Este proceso comenzará por el primero de la lista y si falla seguirá recorriéndola hasta que no se produzca un error. Inicialmente, esa lista solamente contendrá el *ModelBackend*, que se encargará de buscar en la tabla de usuarios si el usuario y contraseña son correctos, así como los permisos del mismo.

Es posible tanto extender de los *Backends* que Django trae por defecto, como el implementar uno personalizado, cuando se requiera que el usuario se autentique en otro servicio diferente al que viene por defecto. Un ejemplo sencillo por el cual se puede requerir un *Backend* personalizado es que el usuario pueda iniciar sesión con el correo en lugar de con el nombre de usuario. Un *Backend* personalizado será una clase Python que implementa dos métodos obligatorios: *get_user()* y *authenticate()* así como un conjunto opcional de métodos de autorización de permisos.

13.3 Permisos y grupos

En el tema de los permisos, Django viene con un sistema de gestión de los mismos por defecto, con el cual se podrán asignar y eliminar permisos tanto a usuarios individuales como a grupos de usuarios.

En cuanto al alcance del sistema de permisos, va desde el sitio de administración de Django, hasta poderlo usar en el propio código de la aplicación. A nivel de objetos se pueden gestionar permisos por según su tipo y por instancias de los mismos.

Para relacionar los permisos y los grupos a los que pertenece un determinado usuario, existe en la clase *User* dos relaciones muchos – a – muchos, *user_permissions* y *groups* respectivamente. Con ello quedará completamente definido el usuario tanto con sus datos como de los permisos y grupos a los que pertenece.

Si se tiene *django.contrib.auth* instalado en el proyecto, como viene por defecto, Django a cada modelo le asignará automáticamente 3 tipos de permisos, que serán el de crear, modificar y eliminar (*add*, *change*, *delete*).

Es posible añadir permisos adicionales para un modelo determinado de la aplicación aparte de los 3 que tienen por defecto. Para ello, se añaden en la clase *Meta* del modelo en el diccionario *permissions*. De esta forma, se agregarán dichos permisos extras a la tabla de permisos de la aplicación cuando se crea un objeto de dicha clase.

El principal objetivo de que existan grupos en Django es el poder agrupar a un número determinado de usuarios que tengan los mismos permisos. Con esos usuarios agrupados, se pueden dar permisos al grupo, donde automáticamente se extenderán a los usuarios que forman parte de él, por lo que no será necesario ir usuario a usuario modificando dichos permisos.

13.4 Proceso de autenticación en peticiones Web

Una parte muy utilizada en cuanto a la gestión de los usuarios en una web es la parte en la que ellos crean su cuenta para posteriormente loguearse y cerrar su sesión mediante formularios de la aplicación.

13.4.1 Vistas y plantillas

Para que un usuario pueda iniciar sesión y cerrar sesión, con el backend de autenticación por defecto, existen dos formas para ello. La primera es mediante una vista personalizada, recoger los parámetros *username* y *password* del correspondiente formulario y, posteriormente, llamar al método *authenticate*. A dicho método será necesario pasarle el objeto *request* junto con el usuario y la contraseña. Esto se realiza para comprobar que si no devuelve *None*, se pueda llamar al método *login()* pasándole *request* y el resultado de la llamada a *authenticate*. Ambos métodos pertenecen al paquete ‘*django.contrib*.auth*’*.

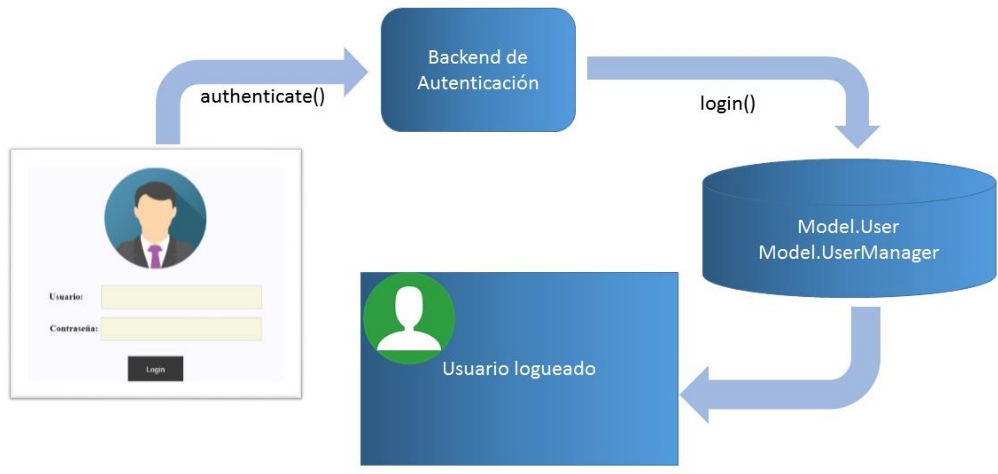


Ilustración 12 - Proceso de autenticación de usuarios en Django.

Al igual que se ha creado un método para el inicio de sesión de un usuario en el sistema, también es necesario crear un método para el cierre de la misma. En una vista de *logout* personalizada se realizará una llamada al método *django.contrib.auth.logout()* pasándole como parámetro el objeto *request*. De esta sencilla forma, se cerrará la sesión del usuario actual de la aplicación.

```
def login_cliente(request):
      form = login_form(request.POST or None)
      pag_redireccion = request.GET.get('next', '')
      if form.is_valid():
            form_data = form.cleaned_data
            usuario = (form_data.get("username"))
            password = (form_data.get("password"))
            in_lista_clientes =
Cliente.objects.filter(user__username = usuario).exists()
            if in_lista_clientes:
                  user = authenticate(request, username=usuario,
password=password)
                  if user is not None:
                        login(request, user)
                        if pag_redireccion is not '':
                              return
HttpResponseRedirect(pag_redireccion)
                        else:
                              return HttpResponseRedirect('/cliente')
                  else:
                        form.add_error("username", 'El usuario y
contraseña son incorrectos')
            else:
                  form.add_error("username", 'El usuario indicado no
se encuentra registrado como cliente en la aplicación.')
      return render(request, 'login.html', {'form': form})
```

Otra posible solución es utilizar las vistas que provee Django para realizar el inicio de sesión, cierre de sesión entre otras gestiones de autenticación en lugar de definirse los métodos como se comentaba anteriormente. La última versión de Django, la 1.11,

implanta un cambio importante en cuanto a estas vistas de autenticación se refiere, ya que pasan de ser vistas basadas en funciones a vistas basadas en clases.

Vista basada en función Desaparece en 1.11	**Vista basada en clase Novedad en 1.11**	**Descripción**
login()	LoginView	Vista que se encarga de realizar login del *User*.
logout()	LogoutView	Vista que se encarga de realizar el cierre de sesión del *User*.
password_change()	PasswordChangeView	Permite al *User* cambiar su contraseña.
password_change_done()	PasswordChangeDoneView	Se corresponde con la página que se muestra cuando un *User* cambia su contraseña correctamente.
password_reset()	PasswordResetView	Permite al *User* resetear su contraseña mediante la generación de un enlace que se le enviará por correo.
password_reset_done()	PasswordResetDoneView	Página que se mostrará después de que un usuario ha recibido el correo para resetear su contraseña.
password_reset_confirm()	PasswordResetConfirmView	Muestra un formulario para que el *User* introduzca su nueva contraseña.
password_reset_complete()	PasswordResetCompleteView	Vista que informa al usuario que su contraseña se ha reestablecido correctamente.

Tabla 18 - Tabla de vistas de autenticación.

Como nota importante cabe destacar que Django no proporciona ninguna plantilla por defecto para dichas vistas de autenticación. Pero en cambio, sí que introduce la ruta por defecto de donde deberán de almacenar dichas plantillas que implementará el usuario. Este directorio es */registration/*. Donde para cada clase tendrá la suya, por ejemplo, para *LoginView*, será */registration/login.html*. Esto que se acaba de comentar no es un requisito que haya que seguir estrictamente ya que es posible utilizar una plantilla situada en otro lugar, indicándolo mediante el atributo *template_name*. Dicho atributo recibirá el nombre de la plantilla asignada por el usuario.

Aparte del atributo *template_name* que tienen dichas clases, cada una tendrá otros tantos particulares, recogidos en la documentación oficial. A modo de ejemplo, un atributo que tienen en común las clases anteriores puede ser, *extra_context*, donde se le podrá pasar un contexto extra para añadir al contexto por defecto a la plantilla.

Se muestra un ejemplo de uso de estas vistas basadas en clase, para un proceso de *login* y *logout* de un usuario en una aplicación:

```
url(r'^cliente/login/$', auth_views.LoginView.as_view(template_name =
'login_cliente.html'), name = 'login'),
url(r'^cliente/cambiar-pass-done/$',
auth_views.PasswordChangeDoneView.as_view(template_name  =  'cambiar-
pass-done.html'), name="password_change_done"),
```

13.4.2 Formularios

Aparte de las vistas y plantillas que se han comentado anteriormente, Django, siguiendo su principal objetivo de facilitar la tarea a los programadores, incluye una serie de formularios que se utilizarán para dichas tareas de autenticación de usuarios. Estos formularios que incluye Django por defecto son los siguientes:

Clase del formulario	Descripción
AdminPasswordChangeForm	Formulario para cambiar la contraseña de un administrador.
AuthenticationForm	Formulario para el inicio de sesión de los usuarios.
PasswordChangeForm	Este formulario permite al usuario cambiar su contraseña.
PasswordResetForm	Sirve para generar y enviar por correo el link para el restablecimiento de la contraseña
SetPasswordForm	Permite al usuario cambiar su contraseña sin meter la antigua.
UserChangeForm	Formulario de la parte de administración para cambiar tanto la información del usuario como sus permisos
UserCreationForm	Formulario para crear un nuevo usuario de la aplicación.

Tabla 19 - Formularios predefinidos en Django.

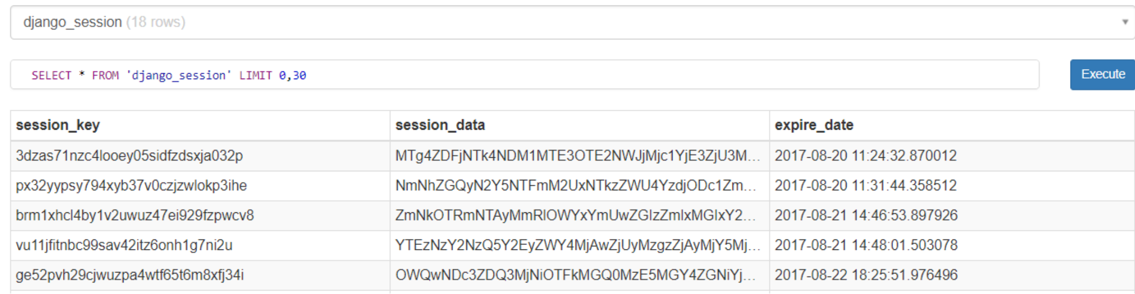
El usuario puede utilizar sus propios formularios de autenticación si no desea utilizar los que provee Django por defecto. Implementar formularios de autenticación personalizados puede resultar muy útil cuando el programador desea incluir apartados extras que no se contienen en dichos formularios predefinidos. Para crearlos, se debe de realizar como cualquier formulario estándar, pero teniendo en cuenta la clase *User*, si se usa para los usuarios.

13.5 Sesiones de los usuarios

Un apartado muy importante en cuanto a la gestión de usuarios es el uso de *cookies* y de las sesiones de los usuarios de la aplicación. Una *cookie* es información que envía un determinado sitio web y que almacena el navegador del cliente con el objetivo de tener información del usuario. Gracias al uso de ellas, un determinado usuario no tendrá que introducir en cada página de la aplicación sus credenciales de acceso, ya que cuando inicia sesión, se almacena una *cookie*. Django, es consciente de la complejidad de dicho tema y propone un entorno de sesiones al programador para aliviar dicha dificultad. Para utilizar dicho entorno, es necesario agregar al proyecto tanto el middleware '*django.contrib.sessions.middleware.SessionMiddleware*' como tener en

INSTALLED_APPS: *'django.contrib.sessions'*. Esta configuración se incluye por defecto al crear un proyecto Django.

La forma que tiene Django de almacenar los datos de la sesión es en el servidor, en una tabla en la base de datos llamada "*django_session*", donde se almacenará una versión codificada del identificador de la sesión, es decir, el hash.


django_session (18 rows)

```
SELECT * FROM 'django_session' LIMIT 0,30
```

Execute

session_key	session_data	expire_date
3dzas71nzc4looey05sidfzdsxja032p	MTg4ZDFjNTk4NDM1MTE3OTE2NWJjMjc1YjE3ZjU3M...	2017-08-20 11:24:32.870012
px32yypsy794xyb37v0czjzwlokp3ihe	NmNhZGQyN2Y5NTFmM2UxNTkzZWU4YzdjODc1Zm...	2017-08-20 11:31:44.358512
brm1xhcl4by1v2uwuz47ei929fzpwcv8	ZmNkOTRmNTAyMmRlOWYxYmUwZGIzZmlxMGIxY2...	2017-08-21 14:46:53.897926
vu11jfitnbc99sav42itz6onh1g7ni2u	YTEzNzY2NzQ5Y2EyZWY4MjAwZjUyMzgzZjAyMjY5Mj...	2017-08-21 14:48:01.503078
ge52pvh29cjwuzpa4wtf65t6m8xfj34i	OWQwNDc3ZDQ3MjNiOTFkMGQ0MzE5MGY4ZGNiYj...	2017-08-22 18:25:51.976496

Ilustración 13 - Ejemplo de contenido tabla 'session'

En cuanto a la creación de las sesiones de los usuarios, cabe comentar que Django se encarga de dicha gestión. Una vez que en usuario es autenticado, es decir, se ha comprobado que sus credenciales son correctas, cuando se produce la llamada al método *login()*, almacena automáticamente el identificador del usuario en la sesión. Y cuando se cierra la sesión de un usuario y por lo tanto se realiza una llamada al método *logout(),* los datos de la sesión de la solicitud actual quedan eliminados por completo. Es un asunto muy importante en cuanto a la seguridad de la aplicación ya que, si no se realiza dicha acción, otro usuario distinto, aunque aparentemente el anterior se haya deslogueado, seguiría con su sesión. Esto que se acaba de comentar queda transparente al programador si utiliza las vistas *LoginView* y *LogoutView*, ya que Django tiene implementado dicho comportamiento por detrás.

Django, con el fin de aportar la máxima seguridad posible a sus aplicaciones, no almacena información de la sesión en la URL, si no que utiliza las *cookies*. Ya que si no el sistema sería más vulnerable porque se podría robar el identificador de sesión por un usuario no legítimo.

13.6 Limitar accesos a usuarios registrados.

Django es consciente de que es muy utilizado en la programación web el limitar el acceso a ciertas páginas a usuarios que hayan sido previamente registrados. Por ello, aporta un decorador para realizar dicha comprobación. Este decorador es *login_required()*, que reside dentro de *'django.contrib.auth.decorators'* y se usará en las vistas donde se requiera dicha autenticación previa. Se encarga de redirigir a un usuario no identificado a la página de inicio de sesión predefinida y en cambio, si el usuario está identificado, ejecuta la vista normalmente. Puede recibir dos parámetros opcionales:

- *redirect_field_name*: si el campo de redirección no es el predeterminado, es decir, no usa "*next*", se debe de indicar en este parámetro.

- *login_url*: sirve para indicar la página de *login* personalizada si no se usa la que viene por defecto, */accounts/login/*.

Si por el contrario se usan vistas basadas en clases, Django proporciona el *Mixin* '*django.contrib.auth.mixins.LoginRequiredMixin*'. Cuando una vista de este tipo utiliza dicho *Mixin* y un usuario no registrado intenta acceder, son redirigidos a la página de inicio de sesión o se muestra un error HTTP 403 si valor de *raise_exception* de *AccessMixin* está puesto a *True*.

Se muestra un ejemplo de una vista que está restringida para que puedan acceder exclusivamente usuarios registrados en la aplicación:

```
from django.contrib.auth.decorators import login_required

@login_required
def sesion_detalle(request, identificador):
      # resto de código de la vista
```

Aparte de poder limitar el acceso a usuarios registrados a distintas partes de la aplicación, también es posible limitar el acceso si un determinado usuario no cumple un requisito específico, o basándose en los permisos del mismo. Para el primer caso que se acaba de comentar, Django propone el decorador *user_passes_test*, o en su defecto el *Mixin* *UserPassesTestMixin*, si se usan vistas basadas en clase. Con ello se podrá poner un determinado requisito que el usuario deberá de cumplir para poder ejecutar la vista correctamente, o por el contrario su acceso será denegado. De igual manera, para el segundo caso comentado, Django ofrece el decorador *permission_required* y el *Mixin* *PermissionRequiredMixin*, mediante los cuales, se deberá de incluir los permisos que debe de cumplir el usuario para acceder a dicha parte de la aplicación.

Otra forma posible de hacer una diferencia entre usuarios registrados o no registrados es a la hora de mostrar información en las plantillas de la aplicación. Es posible saber si un usuario ha sido registrado o no mediante *request.user.is_authenticated*.

```
{% if user.is_authenticated %}
      <p>Ya estás autenticado en la web.</p>
{% else %}
      <p>Introduce tus credenciales para loguearte como cliente</p>
{% endif %}
```

14 Sitio de administración de Django

14.1 Introducción

Como se ha comentado al inicio de este trabajo, Django fue diseñado en una empresa del ámbito de publicación de noticias en la web. Por lo que, es muy común para ese tipo de aplicaciones disponer de un sitio de administración en el cual se lleven las labores propias de dicha tarea. Como puede ser la gestión de usuarios, añadir, eliminar o modificar entradas de un determinado sitio, entre otras.

Siguiendo con el objetivo principal de Django que es el desarrollo ágil de aplicaciones web, la forma que tiene este Framework de simplificar la tarea de administración es mediante la inclusión de una interfaz HTML para los administradores de la misma, que mediante la lectura de los metadatos de los modelos permitirá modificar su contenido. Cabe comentar que dichos administradores no tienen por qué ser técnicos, por lo que el sitio será limpio y claro para ellos. Esta interfaz que proporciona Django evitará las tareas comunes de desarrollo de estos sitios administrativos, como podrán ser el control del *login* de los usuarios, mostrar formularios y validar sus datos, entre muchos otros, lo cual implica un gran ahorro de tiempo para los programadores.

Esta interfaz también se encargará de que no pueda ser accedida por usuarios que no tengan los permisos para tal acción, lo cual, evitará numerosos problemas.

Esta interfaz de administración, está incluido en el paquete '*django.contrib'*, exactamente en '*django.contrib.admin'* y cabe comentar que es una aplicación Django, por lo que contendrá sus propios modelos, plantillas y vistas. Para acceder a la misma desde una aplicación, es necesario anclarla en el *URLConf*. Toda la configuración necesaria se pasa a describir en este capítulo.

14.2 Preparación entorno administrador

La puesta en marcha del sitio del administrador es muy sencilla, solo es necesario tener unas casuísticas en cuenta:

- Lo primero que es necesario realizar, es añadir '*django.contrib.amdin'* dentro de *settings.py* en el apartado de aplicaciones instaladas: *INSTALLED_APPS*. Con ello se podrá utilizar dicha aplicación.
- Otra configuración necesaria, es definir una expresión regular para determinar cuándo se debe de dirigir a la interfaz de los administradores. Para ello se deberá de incluir en el *URLConf* una instancia de *AdminSite*.
- Para poder acceder a ella por primera vez es un requisito tener una cuenta de un usuario que sea *Superuser*. Para ello hay que crear un usuario con dichas características.
- Como se ha comentado en la introducción, en la interfaz de administración es posible realizar cambios en algunos modelos con el fin de tener un mayor control

de los mismos, pudiendo crear modificar y eliminar objetos en las tablas. Es posible elegir que modelos se desean modificar registrándolos en la página de administración. Ya que cada modelo tiene una clase *ModelAdmin* con las funcionalidades propias que tendrán en el sitio.

14.3 Creación de un superusuario

Para entrar en el sitio de administración, un requisito primordial es disponer de una cuenta de superusuario, es decir, permisos *is_superuser* o disponer de permisos *is_staff* de la clase *User*, que indicará que es miembro del cuerpo de administración. Para crear super usuario desde la consola, será necesario introducir lo siguiente:

```
(GestorCines)        C:\Users\Rober\Documents\GestorCines\src>python
manage.py createsuperuser
Username (leave blank to use 'rober'): roberto
Email address: roberto@correo.com
Password:
Password (again):
Superuser created successfully.
```

Una vez completado los pasos anteriores, se habrá generado un usuario con las siguientes características:

- *is_staff: True*
- *is_superuser: True*
- *is_active: True*

Como se puede apreciar tendrá permisos para realizar cualquier acción dentro de dicho sitio de administración.

14.4 Registrar modelos en el sitio

Como se ha comentado anteriormente, es necesario registrar en el sitio de administración los modelos que se vayan a utilizar en él para llevar a cabo su gestión. Para ello es necesario dirigirse al fichero *admin.py* e incluir la siguiente línea para cada uno de ellos:

```
admin.site.register(Comentario)
```

Con ello, es posible realizar operaciones por defecto de alta, modificaciones y baja de objetos para ese modelo.

14.5 Clases ModelAdmin

Como se ha comentado previamente, es posible decir a Django que parte de los modelos creados en la aplicación se quieren mostrar en el sitio de administración, pudiendo personalizar tanto la información que se muestra de ellos, como los formularios de creación de los mismos.

Para ello Django ofrece las clases *ModelAdmin*, que encapsulará la configuración de ese modelo para el sitio administrativo, es decir, la representación que un modelo tendrá en él. Estas clases se pueden crear opcionalmente, por lo que no es necesario crear para cada modelo registrado en el sitio, una de ellas. Si no se especifica una *ModelAdmin* para un determinado modelo, Django usará las opciones por defecto de administración de dicho modelo.

Estas clases *ModelAdmin* deben de ser implementadas en el fichero, inicialmente vacío, *admin.py* y serán subclases de *django.contrib.admin.ModelAdmin*.

Al igual que se registran los modelos para poderlos utilizar en la interfaz administrativa, también será necesario registrar las clases *ModelAdmin*. Django pone a servicio de los programadores el decorador *register* para tal función.

Como se ha ido observando a lo largo de esta documentación, Django ofrece muchas opciones de personalización y las *ModelAdmin* también están incluidas en ese conjunto. Ofrece una serie de opciones para personalizar que partes del modelo visualizar, los formularios de los mismos, así como una serie de widget para interactuar con ellos, como, por ejemplo, agregar una caja de búsqueda, las opciones de guardado, opciones de filtrado, lista de campos a mostrar para los objetos, entre muchas otras. Esto es posible gracias a las diferentes opciones de configuración que dispone dicha clase. Algunas de dichas opciones más relevantes son:

Opción	Descripción
actions	Lista de acciones para la lista de cambios.
list_display	Lista de campos a mostrar para una instancia en la lista de cambios.
list_filter	Lista de filtros a añadir para la lista de cambios.
exclude	Campos a excluir para el formulario de edición.
fields	Campos a incluir en el formulario de edición.
form	Para indicar el ModelForm a indicar si no se desea utilizar el creado automáticamente.
readonly_fields	Campos que no se permite su edición en los formularios.

Tabla 20 - Tabla de opciones de ModelAdmin.

Otra potente funcionalidad que agrega Django a este sitio es la posibilidad de realizar acciones sencillas para poderlas aplicar a una serie de objetos sin tener que ir aplicándolos uno a uno. Para ello será necesario crear una función que realice los pasos deseados. Esta función Python será implementada en *admin.py* y tomará tres argumentos: En primer lugar, el modelo, a continuación, un objeto *HttpRequest* que representará la solicitud y, por último, un *Queryset* que contendrá el subconjunto de elementos seleccionados por el administrador para aplicar los cambios.

A continuación, se pone un ejemplo con varias opciones comentadas anteriormente:

```
class AdminComentario(admin.ModelAdmin):
      readonly_fields = ["comentario","usuario", "pelicula"]
      list_display = ["__str__", "usuario", "pelicula"]
      list_filter = ["usuario", "pelicula"]
      search_fields = ["comentario"]

      # Permisos de añadir objetos
      def has_add_permission(self, request, obj=None):
            return False

      class Meta:
            model = Comentario

admin.site.register(Comentario, AdminComentario)
```

14.6 Cambiar apariencia del sitio de administración

Django, para el sitio administrativo, como se ha comentado anteriormente trae un modelo, vistas y plantillas por defecto. Es posible modificar la visualización de este sitio manejando dichas capas.

14.6.1 Personalización de Plantillas

Se va a comenzar con la capa de plantillas, que como se ha descrito en esta documentación es la encargada de ver cómo se muestran los datos, en este caso, los del sitio del administrador.

Para personalizar dichas plantillas, existen dos opciones, se pueden sobrescribir las existentes o heredar de esas plantillas base.

Para realizarlo de la primera forma, lo primero que hay que realizar es crear un directorio *\admin* dentro de una ruta que se tenga incluida en "*TEMPLATE*" en "*DIRS*", que normalmente se llamará "*templates*". Dicha subcarpeta necesariamente se debe de denominar "*admin*", ya que Django para el sitio de administración busca las plantillas en un subdirectorio con ese nombre por defecto. Una vez creado dicho subdirectorio (*src/templates/admin*), es necesario copiar la plantilla que se quiera reescribir a él y modificarla aquí. Todas las plantillas que trae el sitio de administración por defecto se encontrarán en: *django\contrib\admin\templates\admin*, donde Django haya sido instalado.

Como nota importante, cabe mencionar que no se deberían de modificar en la carpeta raíz de Django ya que si no se realizarían dichas modificaciones para todas las aplicaciones generadas.

Algunos ejemplos de plantillas que trae por defecto, entre muchas otras, son:

- base.html
- base_site.html
- change_form.html

- index.html
- login.html

Otra forma de tener plantillas personalizadas de Django para el sitio de administrador es mediante la herencia de las plantillas dadas por defecto. Para ello, las plantillas creadas heredarán de ellas incluyendo en las etiquetas de bloques el contenido que se estime oportuno por el diseñador. Como la plantilla nueva generada extiende del modelo de plantilla base de Django para el sitio, tendrá el mismo aspecto que ellas.

Otra opción de personalización de las plantillas es la integración en un subdirectorio *\admin*, como en el primer caso, unas plantillas predefinidas de terceros. Estas incluyen una personalización diferente lo que permitirá tener un nuevo sitio de administrador. Aparte de integrarlas solamente, es posible, reescribirlas personalizándolas a gusto del diseñador del sitio.

14.6.2 Personalización de vistas

Aparte de modificar las plantillas de un sitio web, es posible modificar las vistas y crear vistas personalizadas.

Crear vistas personalizadas para el sitio de administración es muy parecido a crear vistas para cualquier modelo en Django. Dichas vistas no es necesario incluirlas en un directorio predeterminado como sucede en el caso de las plantillas anteriormente comentado, basta con definirlas en *views.py*. Para ello es necesario escribir una función que realice lo que el programador determine oportuno y, posteriormente, incluirla en el *URLConf*. Una nota importante, es que se debe de incluir dicho patrón URL antes que las vistas que incluye del administrador. Debido a que como se procesa en orden, si se pone a la inversa, nunca entraría en esa vista personalizada.

14.6.3 Personalizar formularios

Como se ha comentado en el apartado 5 de este capítulo, las subclases de *ModelAdmin* incluyen distintas opciones de personalización de formularios para el sitio de administración. Con ellas un programador podrá manejar el formulario según sus requisitos. Algunas de esas opciones son: *exclude*, *fields*, *form*, entre muchas otras.

14.7 Gestión de usuarios en el sitio de administración

Es posible que aparte del superusuario creado para el sitio de administración de Django, también lo utilicen otros usuarios administradores, los cuales podrán tener distintos perfiles restringiéndoles el acceso a las partes que se estimen oportunas. Como cabe esperar el superusuario de la aplicación podrá realizar todo tipo de acciones en el sitio.

En el sitio de administración de Django, por defecto, se muestra un apartado para la gestión de usuarios y grupos de la aplicación.

Ilustración 14 - Sección ‘Autenticación y autorización’ en el sitio del administrador.

Dentro de la sección de grupos de esta parte de “*Autenticación y autorización*”, es posible que el administrador de la página estime oportuno generar grupos de usuarios para agruparles según unos permisos determinados.

Para crear un grupo de usuarios es necesario darle a “*añadir*” dentro del apartado “*Grupos*” y a continuación introducirle un nombre y unos permisos que tendrán los usuarios pertenecientes a él.

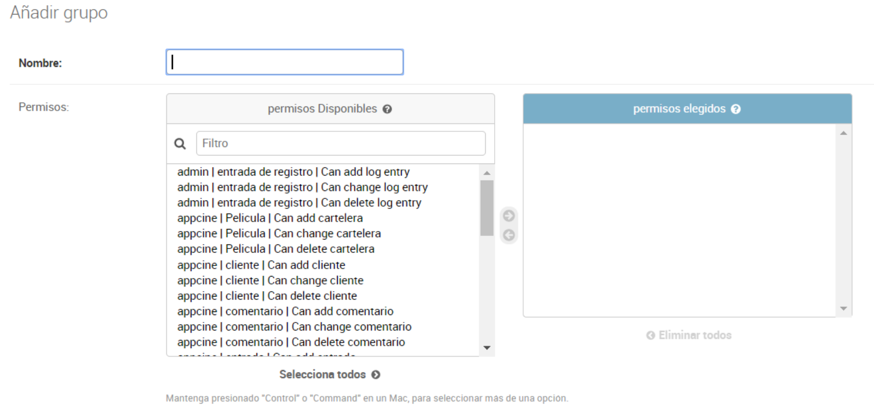


Ilustración 15 - Añadir grupo en el sitio del administrador.

Una vez creado el grupo, no se asigna a ningún usuario ya que será en la parte de gestión de los mismos donde se realizará dicha acción.

En la sección de gestión de usuarios, es posible tanto crear uno nuevo como modificar las características de uno existente. Para crear nuevos usuarios será necesario darle a la opción “*Añadir usuario*” y posteriormente introducir un nombre y contraseña. Posteriormente, para modificar los atributos de este, es posible pinchando sobre el nombre de usuario de la lista de cambios de usuarios. Donde saldrán distintos apartados como el nombre de usuario junto con la contraseña indicando el algoritmo elegido, información personal, apartado de permisos y de fechas importantes.

- **Información personal**
  Donde se podrá rellenar un nombre junto con los apellidos y el correo electrónico de un usuario de forma opcional.
- **Permisos**
  En este apartado y más importante, primero habrá que decidir que tipo de usuario será y si tiene permisos para loguearse tanto en la parte del cliente como en la de administración.

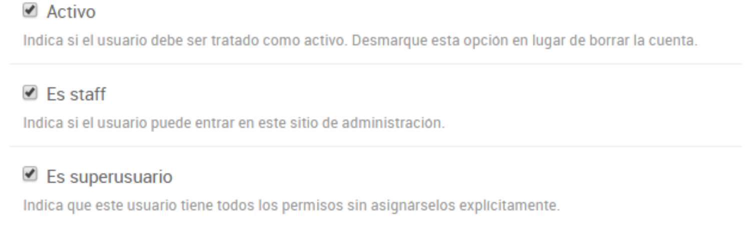


Ilustración 16 - Permisos de usuario en el sitio del administrador.

Estos campos que se aprecian en la imagen anterior se corresponden con los campos de la clase *User*, *is_active*, *is_staff* y *is_superuser*.
A continuación, se muestran los grupos existentes y la opción de añadirle a uno de ellos.

Ilustración 17 - Asignación de grupos a usuarios en el sitio del administrador

Por último, se podrán asignar permisos particulares para el usuario a modificar, mostrándose en una lista todas las opciones de permisos para cada modelo junto con opciones de añadir modificar o eliminar permisos, usuarios, tipos de contenido y sesiones.

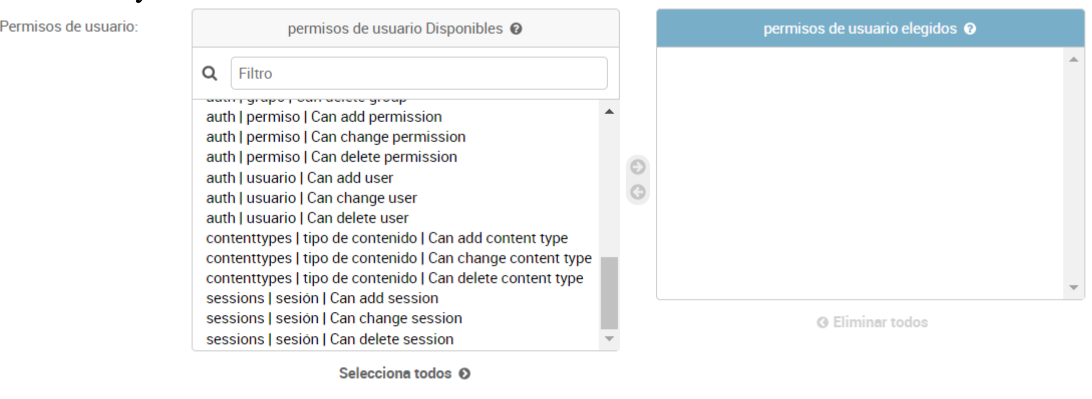


Ilustración 18 - Gestión permisos de usuarios en el sitio del administrador

- **Fechas importantes**
  En este último apartado se muestra la fecha en la que se ha dado de alta el usuario y la de su último acceso.

15 Requisitos aplicación a desarrollar

En este capítulo se pretenden definir los requisitos que tendrá la aplicación web desarrollada a modo de estudio del Framework de programación web Django.

Se da el supuesto caso que una empresa de gestión de cines quiere implantar un sistema de gestión del mismo en la Web, mediante el cual se permita llevar a cabo las principales gestiones de un cine, como, por ejemplo, gestión de cartelera, sistema de venta de entradas, entre otras.

La aplicación constará de dos partes claramente separadas, una para clientes y otra para administradores del cine.

En cuanto a la parte del **cliente** se pueden destacar las siguientes partes:

I. **Sistema de gestión de usuarios**
   En este apartado se pueden realizar distintas acciones características de cualquier sistema de gestión de usuarios. Se pasa a comentar cada una de ellas:
   - **Registro de cliente**
     Se requerirán los siguientes campos para el registro de un nuevo cliente en la aplicación:
     - Nombre y apellidos.
     - Correo electrónico.
     - Fecha de nacimiento.
     - Foto de perfil.
     - Nombre de usuario y contraseña.

   - **Iniciar sesión**
     Para el inicio de sesión de un cliente se requerirá el nombre de usuario junto con su contraseña. En este apartado no se permitirá el login de los administradores del cine.

   - **Cerrar sesión**
     Al igual que se inicia sesión, es posible cerrarla cuando el usuario lo desee con el fin de poder entrar con otra cuenta o dejarlo libre para que acceda otro usuario desde su misma máquina.

   - **Cambio de contraseña**
     Es posible que un usuario, previamente logueado en la aplicación, quiera cambiar la contraseña por motivos de seguridad, lo cual es posible mediante un formulario en el que deberá de introducir la antigua contraseña y a continuación la nueva a almacenar.

   - **Página de perfil de cliente**
     En la página de perfil del cliente se presentará diferente información relevante para el cliente como puede ser:
     - Foto de perfil del usuario si la subió en el registro.
     - Datos personales del usuario.

- Listado de entradas compradas separadas por las futuras y las que ya ha asistido al cine.
- Visión del conjunto total de los comentarios que ha realizado sobre las películas en cartelera ordenados por la fecha de publicación. En este apartado estará disponible la acción de eliminar comentarios.

II. **Visión de películas en cartelera.**
Sin diferenciar si un cliente se ha registrado o no en la aplicación, es posible visualizar las distintas películas que se encuentran en cartelera para el cine. Las películas que se mostrarán en este apartado son las que tienen al menos una sesión a partir de la fecha del día de visualización de la página en una sala del cine. Quedando excluidas del mismo las que no tienen ninguna sesión futura a pesar de que hayan estado en cartelera anteriormente.
En esta página, estará disponible un filtro de los géneros de las películas en cartelera, pudiendo elegir un determinado género y visualizar los resultados correspondientes. Por último, hay disponible un buscador el cual permitirá buscar por nombre de película o por nombre del director.

III. **Visualización de información de una película en cartelera.**
De las distintas películas mostradas en cartelera, es posible seleccionar una de ellas para ver datos como: el título, director, género, datos de interés, sinopsis, entre otros. De forma opcional y si el administrador ha registrado un tráiler para esta película, se podrá visualizar el vídeo dentro de la propia página. A continuación, se mostrará un listado con las sesiones futuras y una lista con los últimos comentarios de la película. Pudiendo en esta última zona realizar su propio comentario si se encuentra logueado en la aplicación.

IV. **Sistema de compra de entradas**
Por motivos de seguridad de la empresa del cine, esta parte quedará exclusivamente para usuarios que se hayan registrado y logueado en la aplicación. Los cuales, podrán después de elegir una determinada sesión de una película, comprar varias entradas. Se mostrará el cine de forma visual para facilitar al cliente elegir su sitio preferente, si este no se encuentra deshabilitado porque se haya reservado con anterioridad. Después de elegir los asientos deberá introducir su información de la cuenta bancaria para realizar el pago de la compra.
Como nota en este apartado, cabe mencionar que las sesiones de los miércoles serán más baratas que las de cualquier otro día de la semana. Quedando los precios de la siguiente manera:
   - Entrada sesión **miércoles**: 3.4€
   - Entrada sesión cualquier día de la semana menos miércoles: 6.2€

En cuanto a la parte de **administración** cabe mencionar que no todos los usuarios registrados en la aplicación tendrán acceso a esta parte. Solo tendrán permisos los que cree el superusuario de la aplicación con permisos especiales. Estos usuarios administradores podrán:

I. **Añadir, modificar y eliminar películas.**
Tienen la posibilidad de gestionar las películas pudiendo crearlas, modificarlas o incluso eliminarlas. Para dar de alta una película, se pedirán los siguientes campos en el formulario:
   - Título.
   - Sinopsis.
   - Página oficial.
   - Género.
   - Nacionalidad.
   - Duración.
   - Año.
   - Distribuidora.
   - Director.
   - Actores.
   - Clasificación.
   - Datos de interés.
   - Portada.
   - Tráiler.

   Para modificarlas y eliminarlas, las películas serán elegidas de una lista.

II. **Añadir, modificar o eliminar salas del cine.**
Los administradores podrán crear nuevas salas del cine, así como eliminarlas o modificarlas dependiendo de la situación de cada sala real. Para ello una sala tendrá asociada la siguiente información:
   - Nombre de sala
   - Número totales de filas
   - Número totales de columnas

III. **Añadir o eliminar una película existente en la cartelera.**
También llevarán a cabo la gestión de la cartelera del cine. Para ello, será necesario crear una sesión en donde se relacionará una película con una sala y se le asignará una fecha de proyección.

IV. **Visualizar y eliminar los comentarios de las películas.**
Otra opción disponible será la de visualizar todos los comentarios registrados en la aplicación sobre las distintas películas junto con su autor. El objetivo principal de dicho apartado será de poder eliminar alguno de ellos si lo considera oportuno. No pudiendo en ningún caso modificarlos.

16 Implementación de la aplicación

En este apartado se va a describir el proceso de implementación de la aplicación desarrollada siguiendo con los requisitos recogidos en el punto anterior.

Se va a seguir para el desarrollo de este capítulo, el mismo orden que se ha seguido para la investigación del Framework anterior. Por lo tanto, los apartados serán los siguientes:

I. Desarrollo del modelado de la aplicación.
II. Implementación de las funciones vistas que se van a utilizar y configuración del *URLconf*. En este apartado se incluirá el desarrollo de los formularios que se usarán las vistas para recoger los datos.
III. Implementación de las plantillas finales que se mostrarán al usuario de la aplicación.
IV. Levantar el sitio de administración de Django y personalización del mismo.

16.1 Notas iniciales

Para comenzar con el desarrollo de la aplicación, como se ha comentado en el apartado 3 del Capítulo 8, se ha creado un entorno virtual de Python y posteriormente se ha instalado Django. Por lo tanto, se pasa a comprobar que se ha instalado todo correctamente:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines>pip freeze

Django==1.11

pytz==2017.2
```

Como se puede ver la versión de Django que se va a emplear es la 1.11. La versión de Python instalada es la 3.4.4.

Cabe comentar, que se va a mostrar todas las ventajas que Django aporta a los desarrolladores con el servidor de desarrollo incluido en él y con el modo de *Debug* activado, es decir, *Debug = True* en el archivo de configuración, *settings.py*. Se ha determinado tal solución ya que este trabajo de investigación se centra en las ventajas que Django ofrece al programador, y con ello, se observa de una forma más clara.

Como trabajo futuro de esta misma aplicación, se puede seguir implementando nuevas funcionalidades aparte de montarla en otro servidor más potente de cara a la producción.

Por lo tanto, para poner en marcha el servidor de desarrollo que ofrece Django, se deberá de incluir lo siguiente en la línea de comandos:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py runserver
Performing system checks...
System check identified no issues (0 silenced).
September 01, 2017 - 18:34:13
Django version 1.11, using settings 'GestorCines.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Con ello, introduciendo "`http://127.0.0.1:8000/`" en el navegador, se podrá interactuar con la aplicación desarrollada.

16.2 Flujo de la aplicación

En este apartado se pretende resumir de manera que sea legible para el lector, el flujo de la aplicación de gestión de cines que se ha desarrollado. Para ello se muestra a continuación un diagrama del flujo de la misma.

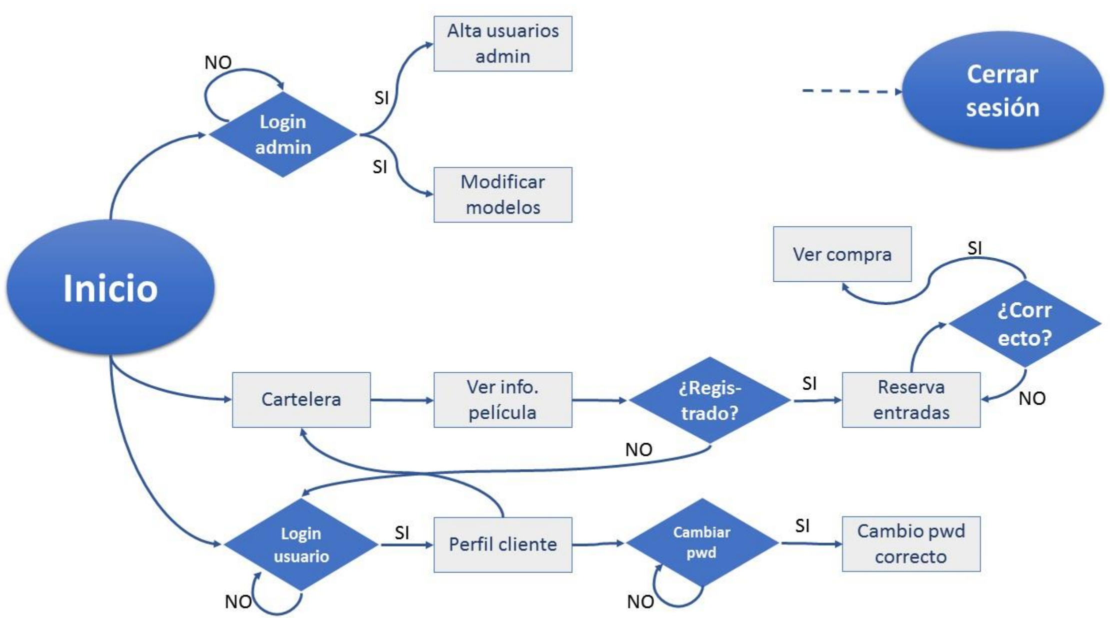


Ilustración 19 - Flujo de la aplicación desarrollada.

En el diagrama anterior se puede apreciar de una forma reducida el flujo de la aplicación que puede seguir un determinado usuario entre las distintas páginas de la misma. Por motivos de legibilidad del mismo, no se ha incluido una unión a '*Cerrar sesión*' desde cada pantalla. Pero se da por hecho de que el usuario en cualquier momento puede cerrar la sesión en la aplicación (siempre que la haya iniciado anteriormente). Es posible distinguir claramente las dos partes de la aplicación: la parte del cliente y la parte de administración.

16.3 Desarrollo del modelo

En esta primera parte, se va a desarrollar el modelo de la aplicación. Cabe mencionar en primera instancia que se ha usado el motor de base de datos de SQLite3, que viene por defecto en Django. En este apartado, se van a crear las distintas clases que se corresponderán con las tablas en la base de datos. Para ello es necesario realizar un estudio previo de los distintos tipos de objetos que existirán en la aplicación junto con los campos que éstos tendrán. Los modelos que existirán son los siguientes:

I. Película.
II. Sala.
III. Cliente.
IV. Entrada.
V. Cartelera.
VI. Comentario.

Llegados a este punto es hora de definir los campos que tendrá cada modelo, así como las opciones de metadatos de los mismos. Para ver todas las relaciones existentes con el modelo, se presenta a continuación un modelado del mismo:

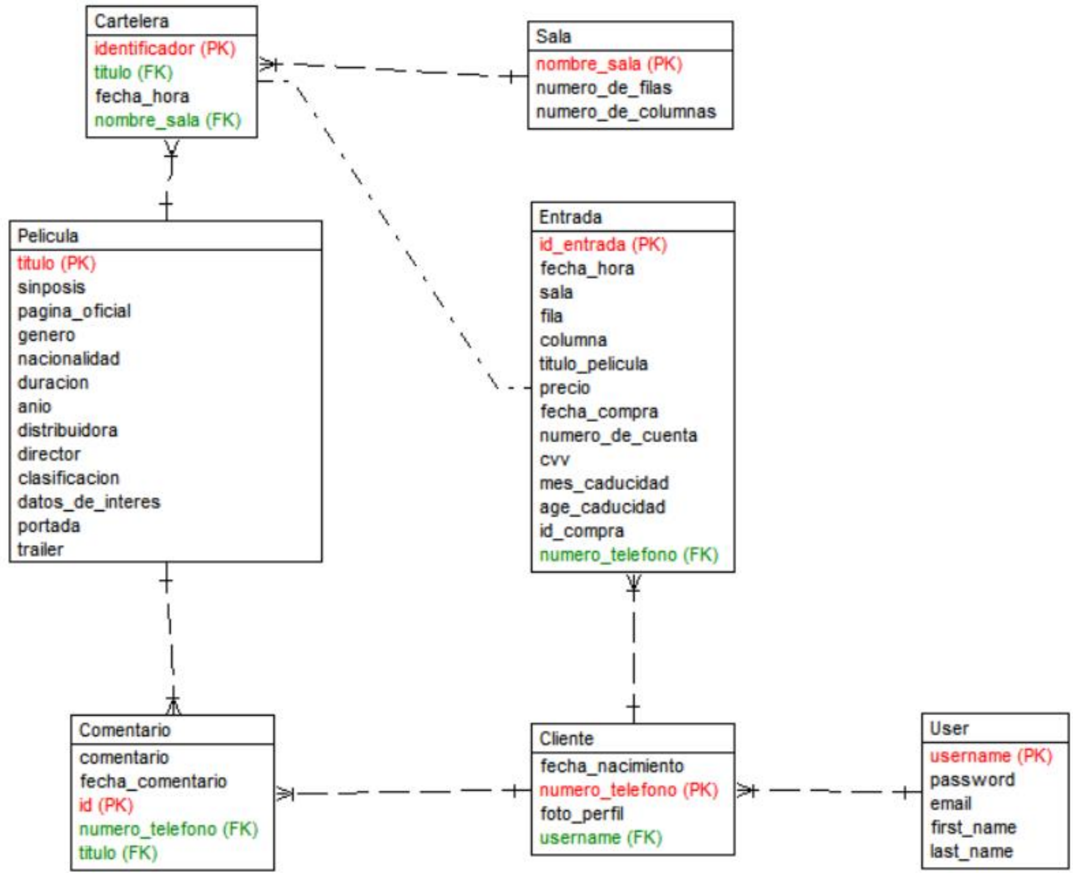


Ilustración 20 - Modelado de la aplicación desarrollada.

Una vez que se tiene claro los modelos a utilizar junto con sus campos y relaciones, es hora de implementarlo en el archivo *models.py*.

Como consideraciones previas, hay que comentar que se van a realizar dos importaciones:

```
from django.db import models
from django.contrib.auth.models import User
```

La primera es necesario realizarla ya que cada modelo es una subclase de *models.Model*. y la segunda es debido a que se va a utilizar el modelo *User* definido en *django.contrib.auth.models*, como una relación en el modelo '*Cliente*'.

Los objetos que se han comentado antes, se van a representar como clases Python, que a su vez es una subclase de '*django.bd.models.Model'*. Cada uno de los campos se corresponden con una variable de dicha clase que representará un campo en la tabla de la base de datos correspondiente. Y estos campos serán una instancia de una clase *Field*, con el objetivo de indicar que tipo de dato contendrá. Cada instancia recibirá una serie de parámetros con el objetivo de satisfacer los requisitos expuestos anteriormente.

Junto con la declaración de las clases que se corresponden con los modelos, se va a definir para cada una de ellas el método *__str__()*, que devolverá una representación Unicode del objeto. En cada método *__str__()*, se devolverá la representación que se quiere obtener para cada modelo.

Así mismo, se definirá cuando sea necesario una clase interna *Meta* para el modelo con el objetivo de indicar metadatos propios. Por ejemplo, se podrá elegir el campo para ordenar los objetos del modelo cuando se extraiga una lista, definir un nombre tanto en singular como en plural para el modelo, entre otras. Las opciones *Meta* utilizadas son:

- *ordering*
- *verbose_name*
- *verbose_name_plural*

Se pasa a describir los tipos de relaciones que se han utilizado entre los modelos:

- **ForeignKey.**
  Este tipo de relación se ha utilizado para vincular los siguientes modelos:
    - *Cartelera* con *Sala* y con *Película*.
    - *Comentario* con *Cliente* y *Película*.
    - *Entrada* con *Película*, *Sala* y *Cliente*.

  Se ha empleado este tipo de relación, muchos – a – uno ya que, por ejemplo, una determinada película puede tener muchos comentarios asociados. La consecuencia de haber utilizado *ForeignKey* sobre dichos campos es que Django crea un índice sobre ellos de forma automática en su tabla de la base de datos.

- **Uno – a – uno.**
  En el modelo de la aplicación actual, existe únicamente una relación uno – a – uno.
    - Cliente con User.

  Se ha empleado dicho tipo de relación con la clase *User* de '*django.contrib.auth.models'*, con el objetivo de poder extender dicha clase para

añadirla nuevos campos adicionales que son requeridos por los requisitos del problema.

Para llevar a cabo que los clientes puedan subir su foto de perfil y que los administradores puedan subir a la aplicación la portada de una película, es necesario realizar la siguiente configuración:

- En el modelo, al indicar que el campo es *ImageField*, será necesario indicar mediante *upload_to*, la subcarpeta donde se almacenará.
- En el archivo *settings.py*, es necesario incluir:

```
MEDIA_ROOT = os.path.join(os.path.dirname(BASE_DIR), "media")
MEDIA_URL = '/media/'
```

Con ello queda definido dónde se almacenarán los archivos subidos por los clientes y administradores, en la carpeta base del proyecto y en el directorio "*/media*". Dentro de este último directorio, se crearán las subcarpetas indicadas por el valor de *upload_to* del correspondiente campo del modelo. Todo ello teniendo en cuenta que se encuentra en el modo de desarrollo.

Se pasa a mostrar un ejemplo completo de una de las clases creadas:

```
class Cliente(models.Model):
      user = models.OneToOneField(User, on_delete=models.CASCADE)
      fecha_nacimiento  =  models.DateField('Fecha  de  nacimiento',
auto_now = False, auto_now_add = False, blank = True, null = True)
      numero_telefono  =  models.IntegerField('Número  de  teléfono',
primary_key=True)
      foto_perfil   =    models.ImageField('Foto    de    Perfil',
upload_to='imgperfil/', blank = True, null = True)

      def __str__(self):
            return self.user.username
```

En cuanto a las migraciones, cabe comentar que cada vez que se ha realizado un cambio en el modelo se han ejecutado los comando *makemigrations* y *migrate* en ese orden. Previamente para comprobar que no hay ningún error en el modelo, se puede ejecutar el comando *check* pasándole la aplicación a comprobar. También se ha empleado la opción *sqlmigrate* para ver el script SQL generado por Django para la creación o modificación de las tablas. Se pasa a mostrar en orden los comandos y sus resultados de una de las primeras migraciones de la aplicación:

```
(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py check appcine
System check identified no issues (0 silenced).

(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py makemigrations
Migrations for 'appcine':
  appcine\migrations\0001_initial.py
    - Create model Cliente
    - Create model Pelicula
    - Create model Sala

(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py sqlmigrate appcine 0001
BEGIN;
--
-- Create model Cliente
--
CREATE TABLE "appcine_cliente" ("usuario" varchar(100) NOT NULL
PRIMARY KEY, "password" varchar(40) NOT NULL, "email" varchar(254)
NOT NULL);
--
-- Create model Pelicula
--
(...)
COMMIT;

(GestorCines) C:\Users\Rober\Documents\GestorCines\src>python
manage.py migrate
Operations to perform:
  Apply all migrations: admin, appcine, auth, contenttypes, sessions
Running migrations:
  Applying appcine.0001_initial... OK
```

16.4 Implementación de vistas y formularios para la parte del cliente

En este apartado se van a describir las vistas y los formularios que se han creado para el desarrollo de la aplicación.

Como se ha comentado a lo largo del desarrollo del Trabajo de Fin de Grado, las vistas serán las encargadas de qué datos mostrar a los clientes. Es por ello, que va a existir una vista para cada una de las partes que tiene la aplicación, con el objetivo de que cada una de ellas cumpla con los requisitos iniciales. Dichas funciones Python se han implementado en el archivo *views.py*. Lo primero que es necesario importar a dicho archivo son los modelos que se han declarado anteriormente, junto con los formularios que se crearán a continuación, entre otras clases necesarias.

Cabe comentar que se han definido funciones para las vistas, las cuales, recibe como parámetro obligatorio un objeto *HttpRequest*. En dicho objeto viaja la información referente a la solicitud realizada por el usuario, por ejemplo, si el método empleado es GET o POST, entre otra información posible. Pueden recibir opcionalmente más

parámetros, que generalmente, será recogido de la URL, el cual es la misión del *URLConf*. Estos parámetros opcionales se comentarán en la descripción posterior de las vistas. Las funciones creadas devolverán un objeto *HttpResponse* con la información de la plantilla que retornará o en caso contrario, una excepción.

Algunas de las vistas definidas usan un atajo proporcionado por Django: *get_object_or_404()*. Dicho atajo permite al programador indicar exclusivamente objeto del modelo a recuperar junto con una condición para extraerlo de forma exclusiva. El atajo será el encargado de devolver el objeto requerido, o si por el contrario no existiera, devuelve un error 404.

En cuanto a los decoradores posibles que pueden tener las vistas, se ha utilizado el *@login_required*, la cual realiza lo siguiente:

- Si el usuario está logueado en el sistema, ejecuta la vista normal.
- Si el usuario no se encuentra actualmente logueado en la aplicación, redirigirá a la página de inicio de sesión y pasa como argumento la ruta a la cual el usuario se dirigía, para que una vez que inicie correctamente sesión, sea redirigido a ella.

En el archivo de configuración settings.py del proyecto, se va a definir lo siguiente:

```
LOGIN_URL = 'login'
```

Con ello se establece que la URL de login para la aplicación, es la que se le pasa a continuación mediante una URL con nombre. Con ellos correctamente señalado Django sabrá a que URL se debe de dirigir cuando un inicio de sesión sea requerido.

Vistas más importantes implementadas en la aplicación:

- **cartelera**
    - **Objetivo**: Mostrar las películas distintas que están en cartelera con una sesión para un día mayor que el actual.
    - **Notas que destacar**:
        - Empleo de código SQL a la hora de realizar una determinada consulta sobre la base de datos.
        - Utiliza objetos *django.db.models.Q* para realizar consultas más complejas.
        - *Querysets* con numerosos filtros encadenados.

- **cartelera_genero**
    - **Objetivo**: Como la anterior, pero devuelve una lista con las películas en cartelera para un determinado filtro de género. No es necesario que la película tenga una sesión futura, contrario a la función anterior.
    - **Nota que destacar**:
        - Mismas que la función *cartelera*.

- **vista_pelicula**
  - **Objetivo**: Mostrar el detalle de la película elegida por el usuario de las que están en cartelera, así como, los 5 últimos comentarios introducidos.
  - **Parámetro adicional**: Título de la película. Este parámetro que recoge de la URL lo utiliza para filtrar por la película escogida por el usuario.
  - **Notas que destacar**:
    - Utiliza el atajo *get_object_or_404*, para seleccionar la película pasada por parámetro, o, por el contrario, muestra un error 404 si la película a buscar no existiera.
    - Emplea el formulario *ComentariosForm*, mediante el cual, recoge el comentario introducido por el usuario, el identificador del usuario y lo almacena en el modelo *Comentario*.

- **película_comentarios**
  - **Objetivo**: Mostrar el total de comentarios para una película.
  - **Parámetro adicional**: Título de la película, utilizado con la misma finalidad que en la función anterior.
  - **Notas que destacar**:
    - Utiliza objetos *django.db.models.Q* para realizar consultas más complejas.

- **registrar_cliente**
  - **Objetivo**: Dar de alta un cliente en la aplicación y recoger los datos del formulario de solicitud de información del mismo.
  - **Notas que destacar:**
    - Utiliza dos comentarios seguidos, uno a continuación del otro, *UserForm* y *ClienteForm*. De esta manera, recogerá los datos necesarios para primero crear una instancia de la clase *User*, para después asignarla al cliente recogido del otro formulario, debido a la relación existente entre *Cliente* y *User*.
    - Comprueba que las dos contraseñas introducidas en el formulario coincidan, tanto la inicial como la de confirmación. Si no coinciden, añade un error al formulario indicando dicha incidencia.

- **cliente**
  - **Objetivo**: Mostrar la información del cliente, junto con la de sus entradas compradas en la aplicación, comentarios realizados en las distintas películas y la opción de cambiar la contraseña de su inicio de sesión.
  - **Notas que destacar**:
    - Utiliza el decorador *@login_required*, para obligar al usuario a loguearse.
    - No recibe el identificador del usuario por parámetro ya que lo recoge del usuario logueado actualmente en la aplicación. Se realiza mediante *request.user*.

- **delete_comentario**
    - **Objetivo**: Eliminar el comentario recogido por parámetro de la tabla correspondiente.
    - **Parámetro adicional**: El identificador del comentario único.
    - **Notas que destacar:**
        - Utiliza la opción *pk* para referirse a la clave primaria de la tabla con la finalidad de eliminar el comentario.
        - Emplea el método *delete()* para eliminar la instancia del modelo *Comentario*.

- **sesión_detalle**
    - **Objetivo**: Mostrar toda la información de la sesión para que el cliente pueda conocerla con la finalidad de comprar sus entradas del cine. Adicionalmente, se dispondrá de una imagen de la sala del cine para elegir sus asientos.
    - **Parámetro adicional**: El identificador de la sesión, para poder extraer toda su información de *Cartelera*.
    - **Notas que destacar:**
        - Realiza el cálculo del precio de la entrada dependiendo de si el día de la sesión cae en un miércoles o no.
        - Mostrará los asientos ocupados y libres de la sala para esa sesión específica.
        - Podrá almacenar varias entradas con una misma solicitud de datos en el formulario.

- **login_cliente**
    - **Objetivo**: Realizar un login para la aplicación personalizado ya que los administradores no tienen permisos con sus cuentas para acceder a la parte de los clientes.
    - **Notas que destacar:**
        - Se ha decidido realizar un login personalizado en lugar de usar el de la vista basada en clase por defecto que trae Django para tal efecto(LoginView) ya que, si no, los administradores con sus cuentas también tendrían permiso para entrar.
        - Realiza la comprobación de que el nombre de usuario introducido en el formulario exista en la tabla de clientes, en cambio, añade al formulario un error de inicio de sesión.
        - Recoge de la URL, mediante el método *request.GET.get()*, el valor de la siguiente página a mostrar al usuario, por si se recibe de la URL como resultado de la ejecución del decorador *@login_required*, definido en cualquier otra vista.

16.5 Desarrollo del URLConf

Partiendo de las vistas creadas en el apartado anterior, se pasa a definir el archivo *urls.py* con la finalidad de tener una correspondencia entre una dirección URL y una vista, además de otra información extra.

Aparte de las importaciones que realiza por defecto Django, es necesario añadir las vistas definidas en *views.py*, así como '*django.conf.urls.static'* para el manejo de los archivos estáticos.

Para cada instancia de *url()* de la variable *patterns*, se le va a asignar un nombre con el objetivo de poder redirigirse a ellas desde otros puntos de la aplicación, entre otras opciones.

Quedando la variable *urlpatterns* final de la siguiente manera:

Expresión regular	Vista	Nombre
`r'^$'`	views.index	index
`r'^admin/'`	admin.site.urls	admin
`r'^cliente/$'`	views.cliente	cliente
`r'^cliente/registrar/$'`	views.registrar_cliente	cliente_registrar
`r'^cliente/registro_correcto/$'`	views.registrar_cliente_ok	registrar_cliente_ok
`r'^cliente/login/$'`	views.login_cliente	login
`r'^cliente/logout/$'`	auth_views.LogoutView	logout
`r'^cliente/logout/correcto$'`	views.despedida	despedida
`r'^cliente/cambiar-pass/$'`	auth_views. PasswordChangeView	cambiar_pass
`r'^cliente/cambiar-pass-done/$'`	auth_views. PasswordChangeDoneView	password_change_done
`r'^cliente/delete/comentario/(?P<id>[^/]*)/$'`	views.delete_comentario	delete_comentario
`r'^cartelera/$'`	views.cartelera	cartelera
`r'^cartelera/(?P<genero>[^/]+)/$'`	views.cartelera_genero	cartelera_genero
`r'^pelicula/(?P<titulo>[^/]+)/$'`	views.vista_pelicula	película_detalle
`r'^pelicula/(?P<titulo>[^/]+)/comentarios/$'`	views.pelicula_comentarios	pelicula_comentarios
`r'^reserva/(?P<identificador>[^/]+)/$'`	views.sesion_detalle	sesión_detalle
`r'^compra-realizada-correctamente/(?P<id_compra>[^/]+)/$'`	views.compra_ok	compra_ok

Tabla 21 - URLConf de la aplicación desarrollada.

Como se puede apreciar en la tabla anterior, se han utilizado vistas basadas en clase proporcionadas por defecto en Django.

Dichas clases son:

- *LogoutView*
- *PasswordChangeView*
- *PasswordChangeDoneView*

Para la utilización de dichas clases como vistas, es necesario llamarlas de la siguiente manera: *LogoutView.as_view()*. De esta forma se pueden pasar parámetros adicionales como la plantilla a usar, entre otros.

Ejemplo de uso para la primera de ellas:

```
url(r'^cliente/logout/$', auth_views.LogoutView.as_view(next_page =
'despedida'), name='logout'),
```

Gracias a la utilización de ellas, se ha permitido un gran ahorro de tiempo a la hora de programar la aplicación. Éste es el objetivo principal de Django, ofrecer al programador soluciones a las tareas comunes en el desarrollo web, como por ejemplo son esas tres funcionalidades: cierre de sesión, cambiar contraseña y vista de contraseña correcta.

Como nota a destacar, estas vistas basadas en clase son nuevas para la versión 1.11 de Django (la utilizada en este proyecto), ya que antes ofrecía más o menos la misma funcionalidad, pero mediante funciones Python en lugar de clases.

Por último, se ha incluido en la variable *urlpatterns* lo siguiente:

```
from django.conf import settings
from django.conf.urls.static import static
urlpatterns = [
      (..)
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Con ello, como se encuentra en el modo de desarrollo la aplicación, se permite utilizar las imágenes subidas tanto por los clientes, como por los administradores.

16.6 Implementación de las plantillas de la parte del cliente

Para la implementación de las plantillas, se ha generado un nuevo directorio en la carpeta raíz del proyecto: *src/templates*. En este directorio será donde se incluyan las distintas plantillas generadas para la aplicación.

Para indicar a Django que tiene que cargar las plantillas del sistema de archivos, es necesario mostrarlo en el archivo *settings.py.*

Se ha utilizado la herencia de plantillas persiguiendo el objetivo de reducir al máximo el número de líneas de código. Para ello, se ha creado una plantilla base, que es el padre de todas, la cual definirá la parte común que será la barra de navegación superior y la parte de debajo de la página, así como distintos bloques para que las plantillas hijas lo rellenen según sus necesidades. Se muestra un diagrama con el uso de las plantillas que se han utilizado en la aplicación actual:

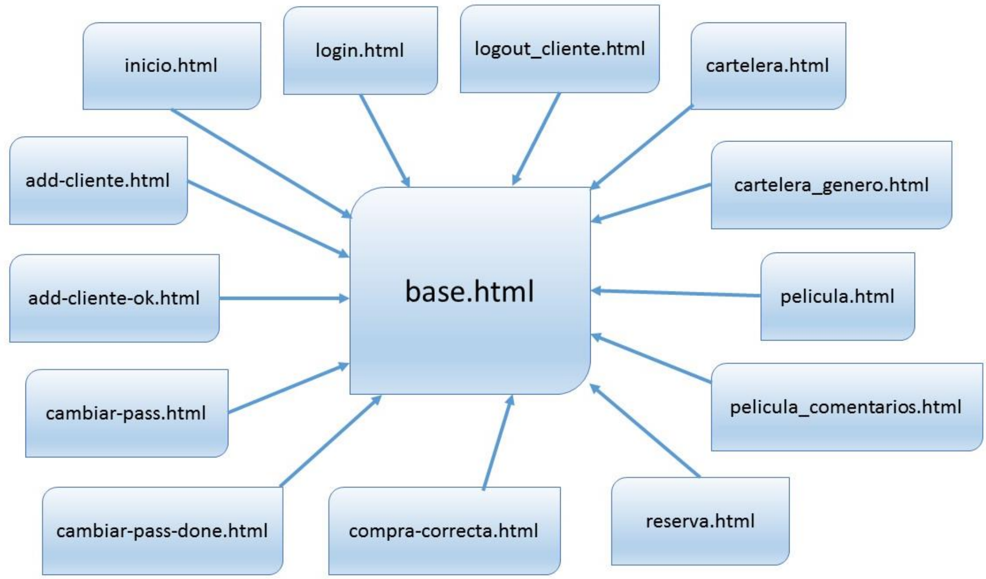


Ilustración 21 - Diagrama de herencia de plantillas.

Para poder realizar dicha herencia de plantillas, es necesario en cada una de las plantillas hijas, introducir la siguiente línea en primer lugar:

```
{% extends "base.html" %}
```

Con este código anterior se indica al motor de plantillas de Django que la plantilla actual hereda de *base.html*, y por lo tanto podrá sobrescribir los bloques definidos en ella, aparte de implementar los suyos propios.

Las plantillas que se muestran en el diagrama anterior han sido definidas mediante el DTL de Django, ya que se han usado variables, filtros y etiquetas sobre el código HTML. Gracias al DTL, es posible hacer más dinámicas las plantillas de Django, así como, mostrar los valores de las variables que recibe la plantilla del contexto pasado por la vista, entre otras opciones.

Cabe destacar como nota importante, que Django escapa automáticamente el código HTML, ya que si no se podrían producir situaciones no deseadas en la aplicación. Pero hay un momento en la aplicación que es necesario desactivarlo, ya que se quiere mostrar el resultado del código HTML introducido en el campo *Trailer* de *Pelicula*. Ya que, de esta manera, se mostrará el video correctamente, y no el código HTML como un texto. Quedando de la siguiente manera:

```
{% autoescape off %}{{ no_peli.trailer }}{% endautoescape %}
```

Para las plantillas que se han creado en este apartado, se han incluido las tecnologías CSS3 y JavaScript. La primera de ellas se ha introducido con la finalidad de poder estilizar las páginas HTML finales generadas tras el renderizado de las plantillas y que se muestren de una forma más agradable al cliente.

Para realizar dicha inclusión de dichos archivos estáticos, se ha creado una carpeta '*src/static/appcine*', y en ella una subcarpeta '*src/static/appcine/imagenes*'. Por lo tanto, es en '*src/static/appcine*' donde se almacenarán los archivos estáticos de la aplicación, como las hojas de estilo, imágenes, *.js*, entre otros. Con el objetivo de poder utilizar dichos archivos estáticos, se ha de incluir la siguiente información en *settings.py*:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static", "appcine"),
]
```

Con todo ello configurado, para utilizar dichos archivos estáticos, será necesario emplear la etiqueta "*{% static %}*" junto con el archivo a introducir. Cabe comentar que es necesario incluir el código "*{% load static %}*" previamente para realizar la carga.

Quedando de la siguiente manera:

```
{% extends "base.html" %}
{% load static %}

{% block archivos_estaticos %}
    <link rel="stylesheet" type="text/css" href="{% static
'base.css' %}" />
    <link rel="stylesheet" type="text/css" href="{% static
'pelicula.css' %}" />
{% endblock %}

      (...)
      <img src="{% static 'imagenes/cartelera.jpg' %}" alt="cartelera"
class="cartelera" WIDTH=400 HEIGHT=250>
```

Se puede ver como se asignan las hojas de estilo a utilizar, junto con una imagen almacenada en "*src/static/appcine/imagenes*".

Con todo este sistema de plantillas comentado, se puede observar la gran flexibilidad que Django aporta a los programadores, pudiendo reducir notablemente las líneas de código gracias a la herencia y también se puede destacar que permite decidir con bastantes detalles como se mostrará cada variable recibida de la vista. Con todo ello, es posible generar un sistema de plantillas para una aplicación bastante completo.

16.7 Configuración del sitio de administración de Django

Como se ha descrito a lo largo de esta investigación, Django viene incorporado con una interfaz de administración cargada por defecto. En la aplicación que se ha desarrollado, se ha utilizado dicha interfaz como base, a través de la cual posteriormente se ha modificado su funcionalidad y apariencia. Todo ello se pasa a comentar en este apartado.

Para utilizar el sitio de administración, lo primero que hay que introducir es la siguiente línea en el *URLConf*:

```
url(r'^admin/', admin.site.urls, name = 'admin'),
```

Con esa línea se introducen todas las URLs del sitio de administración que trae por defecto Django.

Posteriormente, siguiendo con los pasos del apartado 3 del Capítulo 14, se crea un superusuario que tendrá permisos para acceder a esta parte. Cabe comentar que solo los usuarios administradores podrán loguearse en la aplicación. Los usuarios creados en la sección del cliente no tendrán permisos para acceder a ésta. Esta comprobación la realiza de forma interna Django comprobando si el usuario es miembro del equipo de administración o no.

Introduciendo las credenciales del superusuario creado anteriormente, se muestra la siguiente pantalla:

Ilustración 22 - Inicio del sitio de administración de Django.

En ella es donde los usuarios administradores desarrollarán su trabajo, una vez que haya sido completamente implementada, ya que inicialmente sale vacía como se puede observar.

Si he pincha sobre '*Usuario*', se podrá ver que aparece correctamente el superusuario creado:

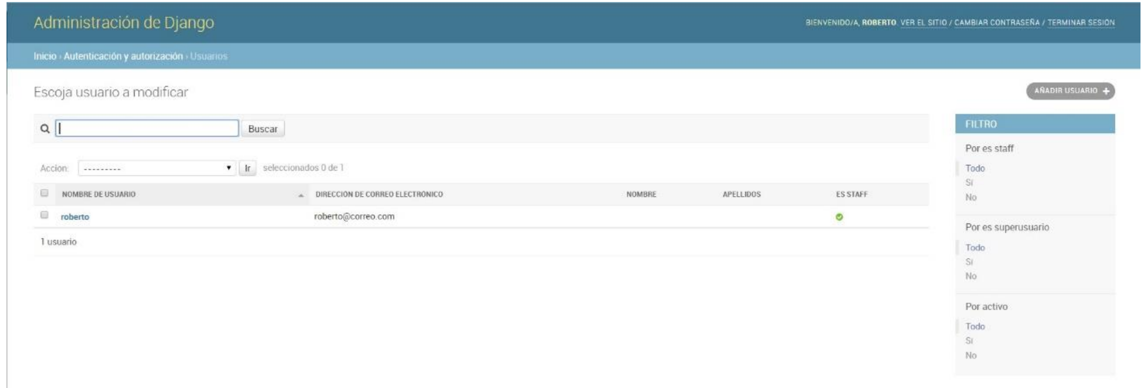


Ilustración 23 - Comprobación superusuario en '*Usuarios*' del sitio de administración

Una vez que se tiene levantado correctamente el sitio de administración, se va a trabajar sobre el archivo *admin.py*, en el cual, se incluirá toda la configuración que se desee añadir al mismo. En este archivo, se va indicar los modelos que se desee manipular desde el sitio de administración y se crearán subclases de *ModelAdmin,* mediante las cuales se permiten configurar diversas opciones para un determinado modelo.

Con las subclases de *ModelAdmin* generadas, es posible manejar opciones de configuración para el sitio, como, por ejemplo: la lista de objetos que se muestra de un determinado modelo, los formularios de creación de instancias para un modelo, los filtros de búsqueda, acciones a realizar con los objetos, entre muchas otras.

Los modelos que se van a registrar para el sitio de administración son los siguientes:

I. Película.
II. Sala.
III. Entrada.
IV. Cartelera.
V. Comentario.
VI. Cliente.

Para cada una de ellas se ha implementado una subclase de *ModelAdmin*, con la finalidad de editar las opciones facilitadas por Django.

Se muestra la configuración necesaria para realizar lo comentado en este punto del trabajo para un determinado modelo:

```
class AdminPelicula(admin.ModelAdmin):
	list_display = ["__str__", "pagina_oficial", "genero", "anio", 
"duracion"]
	list_filter = ["genero", "anio", "duracion"]
	search_fields = ["titulo"]

	class Meta:
		model = Pelicula


admin.site.register(Pelicula, AdminPelicula)
```

17 Manual de usuario de la aplicación

En este capítulo se va a mostrar al lector un manual de usuario de la aplicación. Una vez finalizada la lectura del mismo, podrá utilizar la misma conociendo exactamente las diferentes opciones expuestas en cada página.

En primer lugar, al iniciar la aplicación, se muestra al usuario una pantalla de bienvenida:

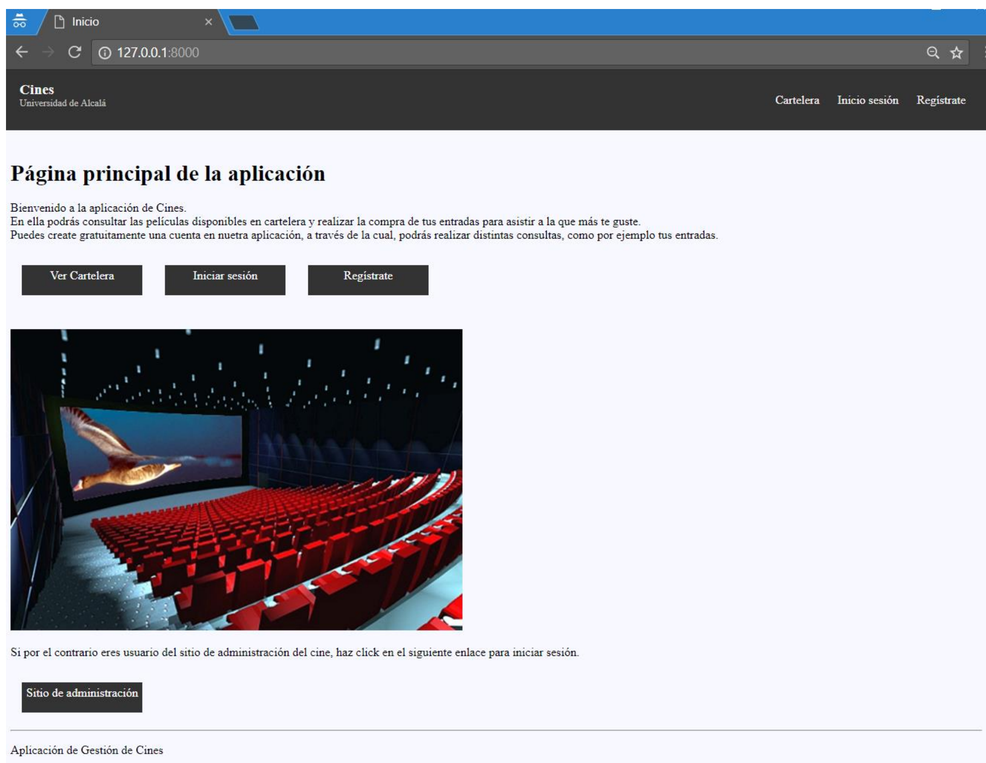


Ilustración 24 - Página principal del gestor de cines.

Como se puede apreciar, es una página de presentación de la aplicación de gestión de cines que se ha desarrollado. En ella, se pueden diferenciar tres partes:

- Una barra fija en la parte superior que aparecerá en el resto de la aplicación. Esta barra de navegación tiene distintos enlaces que se irán modificando de acuerdo a la pantalla en la que se encuentre el usuario con el fin de facilitarle el acceso a otras páginas.
- Descripción y enlaces para los usuarios de la aplicación del lado del cliente.
- Enlace al sitio de administración al que tendrán que acudir los usuarios de la misma.

Comenzando por la **parte del cliente**, se van a describir las distintas páginas a las que tendrá acceso.

Si el usuario en la página anterior hace click en '*Regístrate*', le aparecerá la siguiente pantalla:

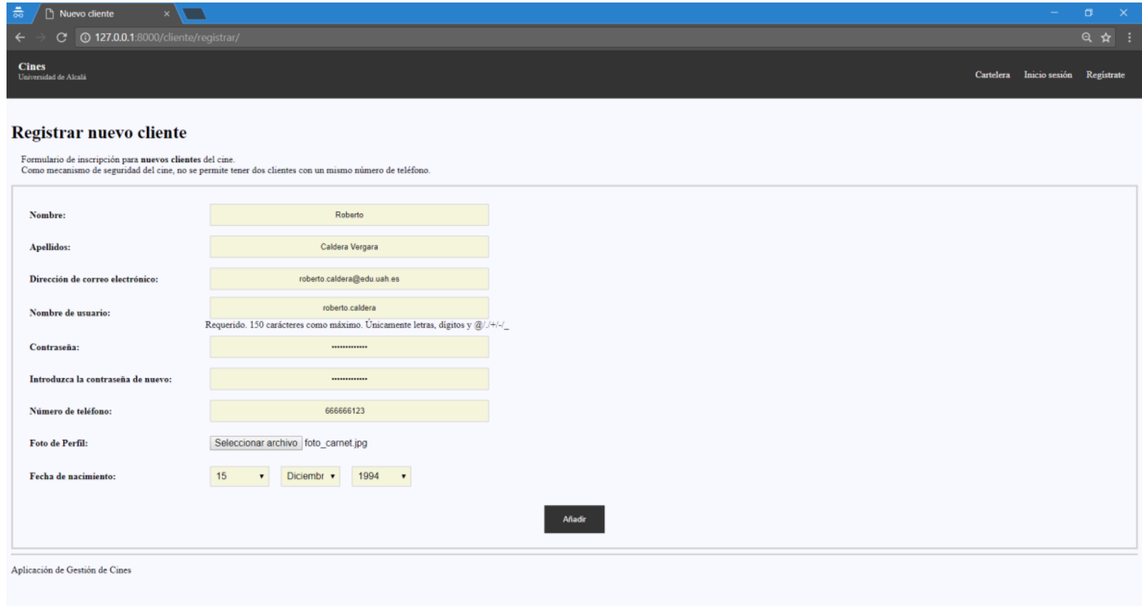


Ilustración 25 - Formulario de registro para nuevos clientes de la aplicación.

En esta pantalla se permite al usuario registrarse en la aplicación como cliente del cine. Dicha acción le permitirá tanto la compra de entradas, como realizar comentarios de películas.

Deberá de rellenar correctamente el formulario anterior con sus datos personales, los datos para el inicio de sesión y por último, puede insertar una fotografía para su perfil si lo desea.

Se pueden producir diferentes errores en el formulario:

- Si el nombre de usuario introducido ya está registrado previamente en la aplicación.
- Si el número de teléfono indicado ya está dado de alta en la aplicación.
- Si la contraseña y la de confirmación no son iguales.
- Si no se rellena algún campo obligatorio. Los campos obligatorios son los requeridos para el inicio de sesión y el número de teléfono: Nombre de usuario, contraseña y teléfono.

Se muestra un ejemplo de un error:

Registrar nuevo cliente

Formulario de inscripción para **nuevos clientes** del cine.
Como mecanismo de seguridad del cine, no se permite tener dos clientes con un mismo número de teléfono.

Por favor corrige los siguientes errores:

Nombre: Roberto

Apellidos: Caldera Vergara

Dirección de correo electrónico: roberto.caldera@edu.uah.es

Ya existe un usuario con este nombre.

Nombre de usuario: roberto.caldera

Requerido. 150 carácteres como máximo. Únicamente letras, dígitos y @/./+/-/_

Ilustración 26 - Error de usuario existente en la aplicación.

Si introduce todos los datos correctamente, le saldrá la siguiente página:

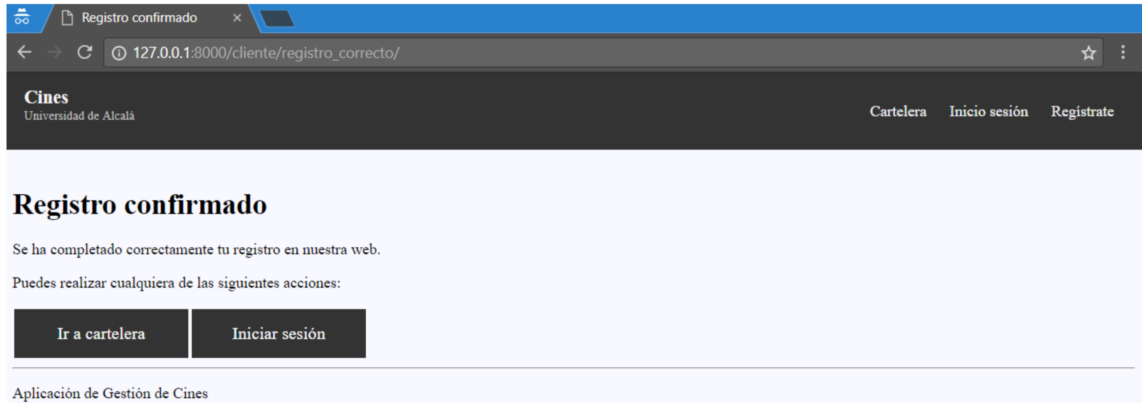


Ilustración 27 - Registro confirmado para un cliente en la aplicación.

En ella se confirma al usuario que ha sido dado de alta correctamente en el sistema. Se le permite dirigirse tanto a visualizar la cartelera actual del cine, como iniciar sesión en la aplicación.

Si pincha en '*Iniciar sesión*', le aparecerá el siguiente formulario:

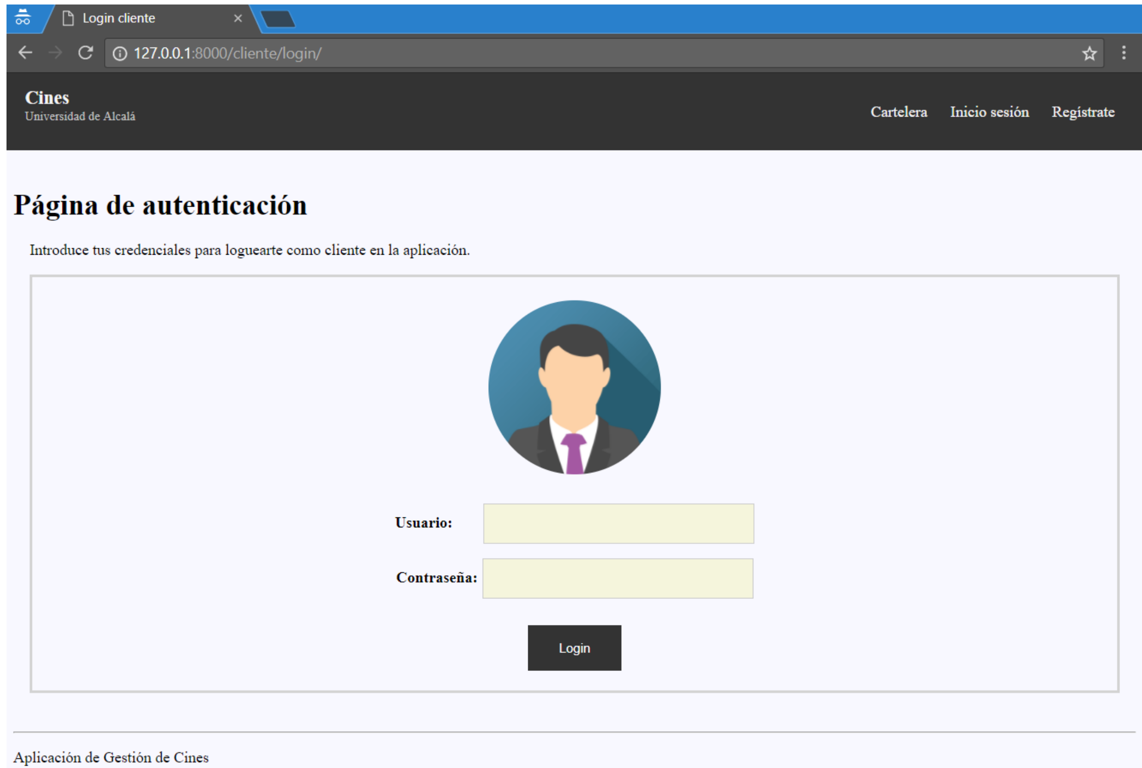


Ilustración 28 - Página de inicio de sesión de clientes en la aplicación.

En este formulario deberá de introducir el nombre de usuario que ha elegido en el registro realizado previamente, junto con la contraseña.

Si el cliente introduce erróneamente su identificador o la contraseña, le saldrá el siguiente mensaje de error:

Ilustración 29 - Error de usuario y contraseña incorrectos para un cliente en el login de la aplicación de gestión de cines.

Como nota, cabe mencionar que este inicio de sesión solo es válido para clientes del cine, no siendo válido para administradores del mismo. Se pasa a comprobar qué mensaje de error sale al intentar loguearse con una cuenta de administración:

Ilustración 30 - Error si un administrador intenta acceder al apartado del cliente en la aplicación.

Si, por el contrario, introduce el cliente sus datos correctamente, es redirigido a la siguiente pantalla:

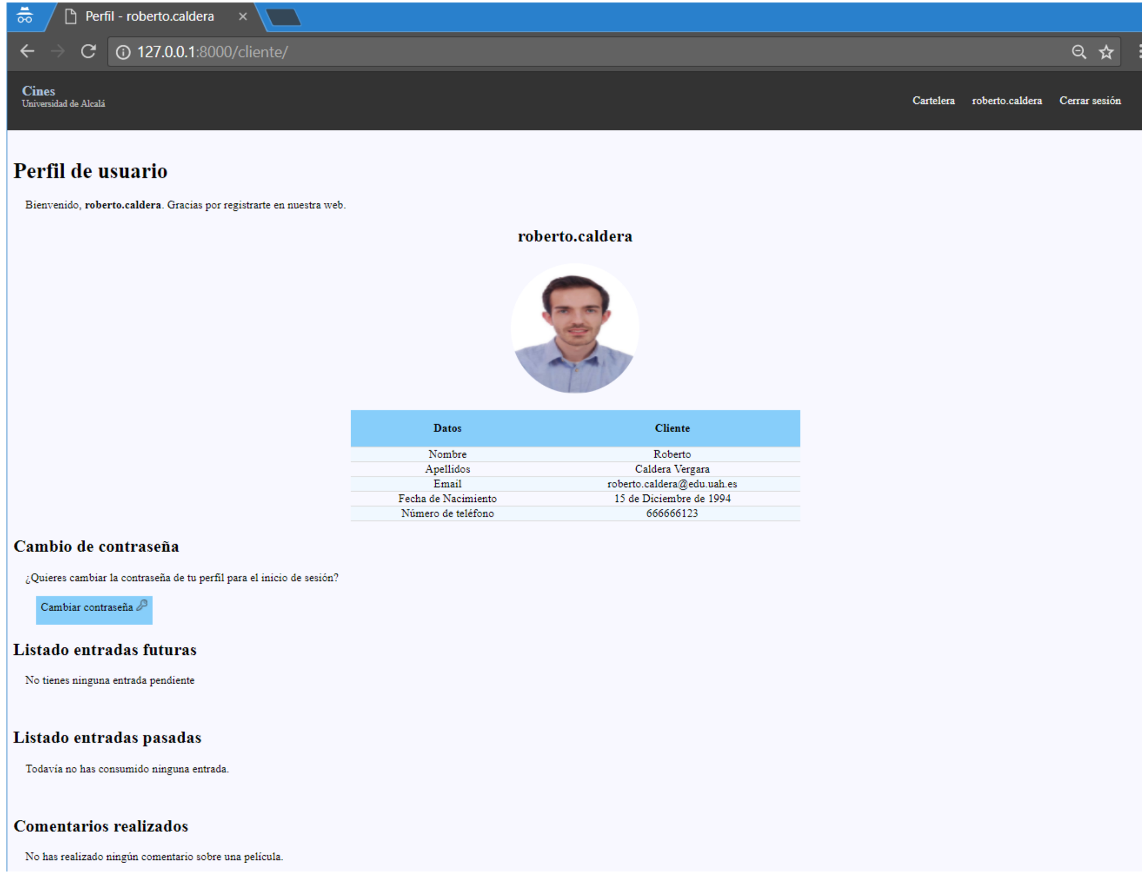


Ilustración 31 - Página del cliente en la aplicación de gestión de cines.

Esta página, es el apartado del cliente en la web, en ella tendrá disponible las siguientes opciones:

I. Visualizar su información personal introducida en el formulario de registro.
II. Ir a la página de cambio de contraseña.
III. Ver el listado de las entradas futuras compradas en el cine.
IV. Ver el listado de las entradas que ya ha consumido.
V. Ver y eliminar si desea todos los comentarios realizados sobre las películas.

Si el cliente decide cambiar su contraseña, puede hacer click en el enlace de la página de perfil comentada anteriormente. Se le redirigirá a la siguiente ventana:

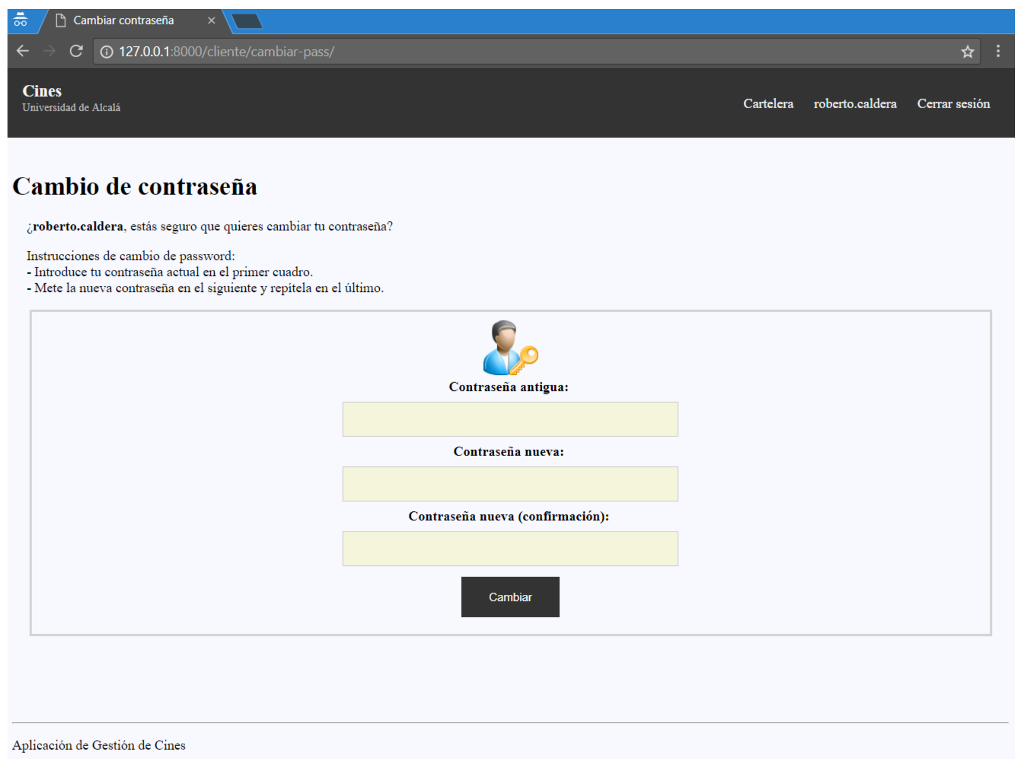


Ilustración 32 - Formulario de cambio de contraseña para un usuario de la aplicación.

En ella, deberá de introducir la contraseña antigua, junto con la nueva y su confirmación para asegurarse de que no se ha cometido ningún error.

Es posible que el cliente se confunda al confirmar la nueva contraseña, pero el sistema le advertirá con un mensaje de error:

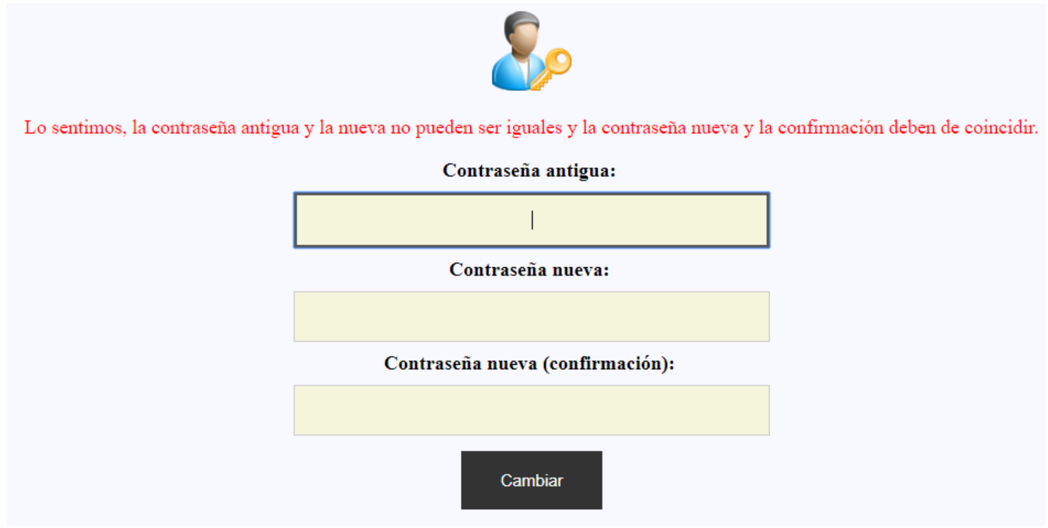


Ilustración 33 - Error en el formulario de cambio de contraseña de la aplicación.

Si realiza el cambio de contraseña correctamente, se le enviará a la siguiente pantalla para que el cliente verifique que se ha realizado correctamente.

Ilustración 34 - Mensaje de cambio de contraseña correcto en el formulario de la aplicación.

Por otro lado, cualquier usuario de la aplicación, registrado o no, pueden ver el contenido de la cartelera del cine. Para ello podrán dirigirse a ella pinchando en el enlace de la barra superior de navegación.

Ilustración 35 - Cartelera de la aplicación de gestión de cines.

Como se puede observar en la captura anterior, el cliente tiene en la misma página información de la película, un filtro de géneros y un buscador de películas y directores.

Por lo tanto, las acciones posibles que el cliente tiene en esta página serán las siguientes:

I. Filtrar por género las películas actuales en cartelera. Para ello, desde el filtro situado en la parte derecha, puede seleccionar el género deseado y se mostrarán las películas para dicho género, si las hay.

II. Buscar por nombre de película o directores. Mediante este buscador, el usuario podrá ver las películas actualmente en cartelera con ese nombre de película o director.

III. Visualizar toda la información posible de la película elegida. Para ello deberá de pinchar sobre la película que se quiera ver su contenido.

Si se realiza la última acción comentada anteriormente, se puede observar lo siguiente:

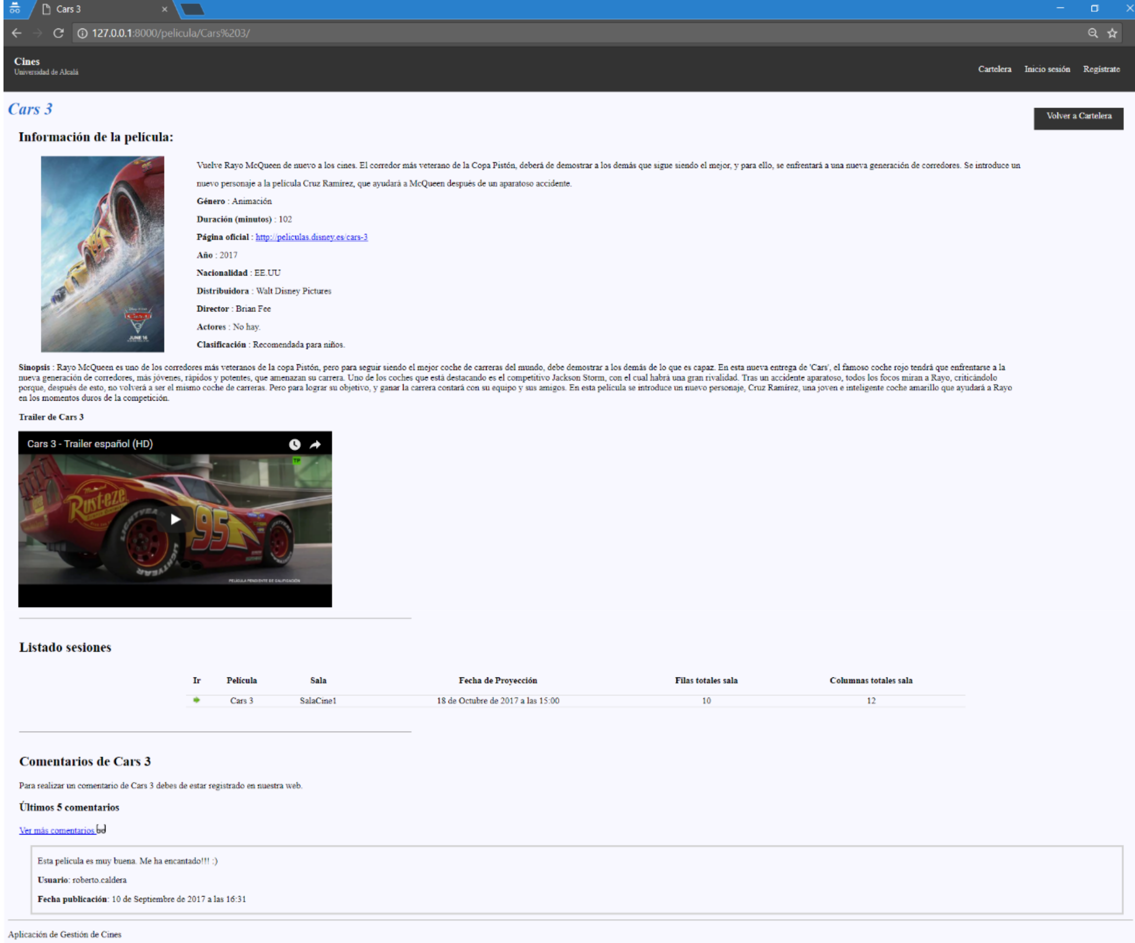


Ilustración 36 - Información de una película en la aplicación.

En esta página, se muestra la siguiente información sobre la película seleccionada de la cartelera.

- Información de la película: Título, género, duración, página oficial, año de rodaje, nacionalidad, distribuidora, director, actores, clasificación, sinopsis y tráiler (si existe).
- Listado de sesiones futuras para la película elegida, junto con distinta información de la misma.
- Posibilidad de realizar un comentario si se encuentra logueado actualmente como cliente.
- Últimos 5 comentarios registrados sobre esta película.
- Enlace para visualizar todos los comentarios de la misma.

Si hace click para ver la información de la sesión elegida, será llevado a la página de reserva de entradas, si se encuentra registrado. Si no está registrado, será redirigido a la página de login y automáticamente al iniciar sesión va a la página solicitada:

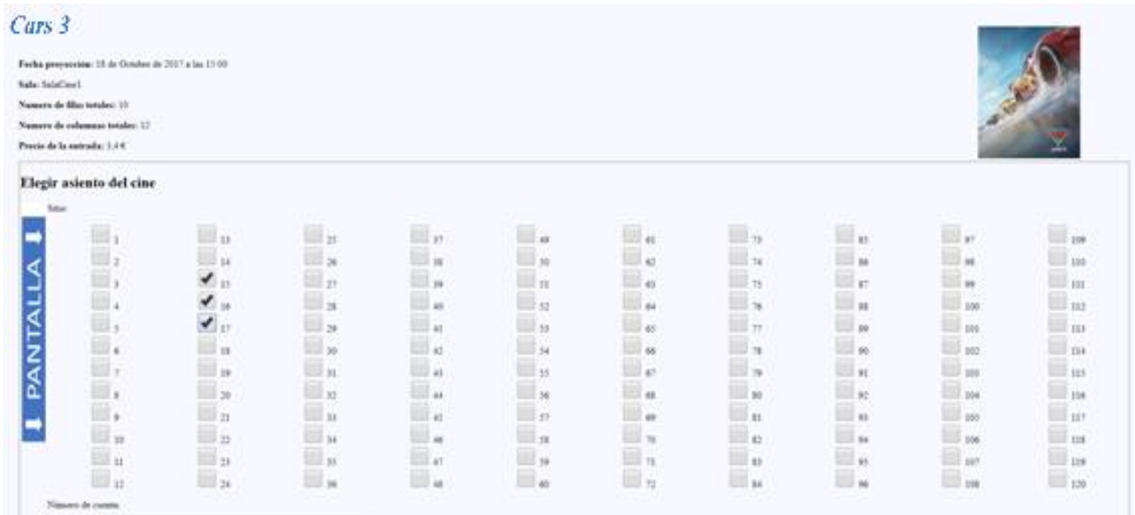


Ilustración 37 - Selección de asientos de una sala en la aplicación de gestión de cines.

En ella podrá elegir los asientos deseados de la sala para la sesión actual. Como restricción en cuanto a los asientos, cabe comentar que los se encontrarán deshabilitados, no pudiendo elegirlos en el formulario.

Una vez seleccionados los asientos, deberá de rellenar la información siguiente de su cuenta bancaria:

Ilustración 38 - Formulario de cuenta bancaria en la aplicación de gestión de cines.

Con todo ello, tras confirmar la compra en una ventana emergente, le saldrá el resumen de la misma:

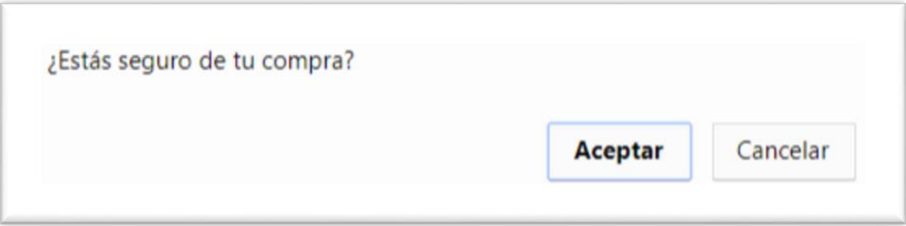


Ilustración 39 - Mensaje de confirmación de compra en la aplicación.

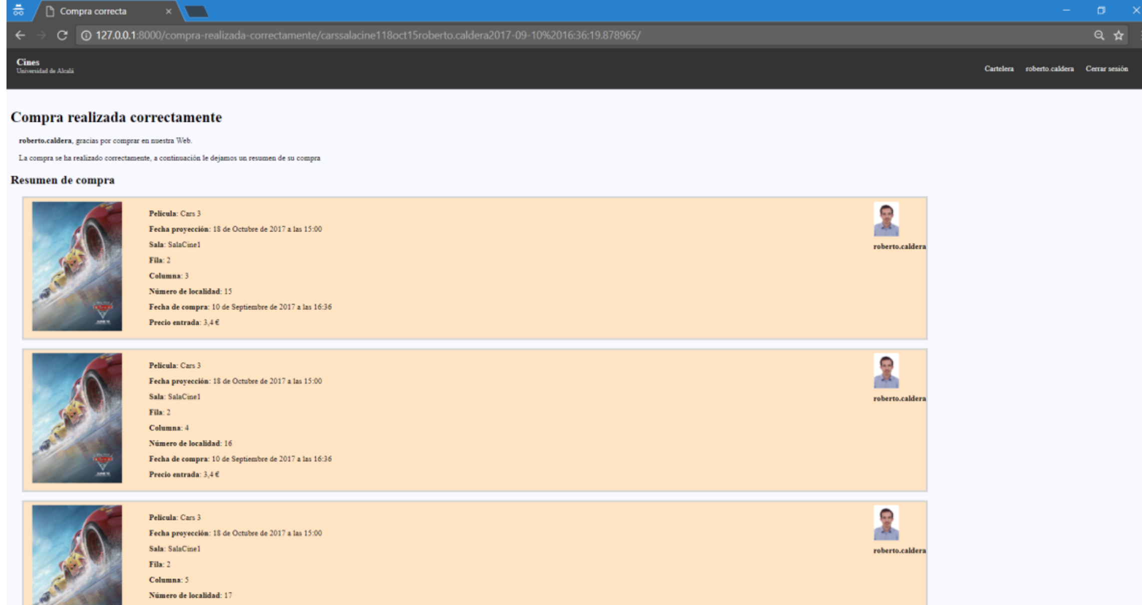


Ilustración 40 - Resumen de compra de la aplicación.

Se podrán visualizar las entradas compradas para la sesión elegida, en las cuales, se muestra un resumen de la misma. Al final de la página se muestra el importe total de la compra.

Si el cliente a continuación se dirige a su página de perfil, podrá comprobar que le aparecen las entradas compradas:

Listado entradas futuras

Película	Fecha proyección	Sala	Fila	Columna
Cars 3	18 de Octubre de 2017 a las 15:00	SalaCine1	2	3
Cars 3	18 de Octubre de 2017 a las 15:00	SalaCine1	2	4
Cars 3	18 de Octubre de 2017 a las 15:00	SalaCine1	2	5

Ilustración 41 - Listado de entradas futuras en perfil de cliente en la aplicación.

El cliente en cualquier momento tiene la posibilidad de cerrar la sesión. Para ello bastará con hacer click en '*Cerrar sesión*' en la barra de navegación superior.

Si se produce esta acción, le saldrá el siguiente mensaje:

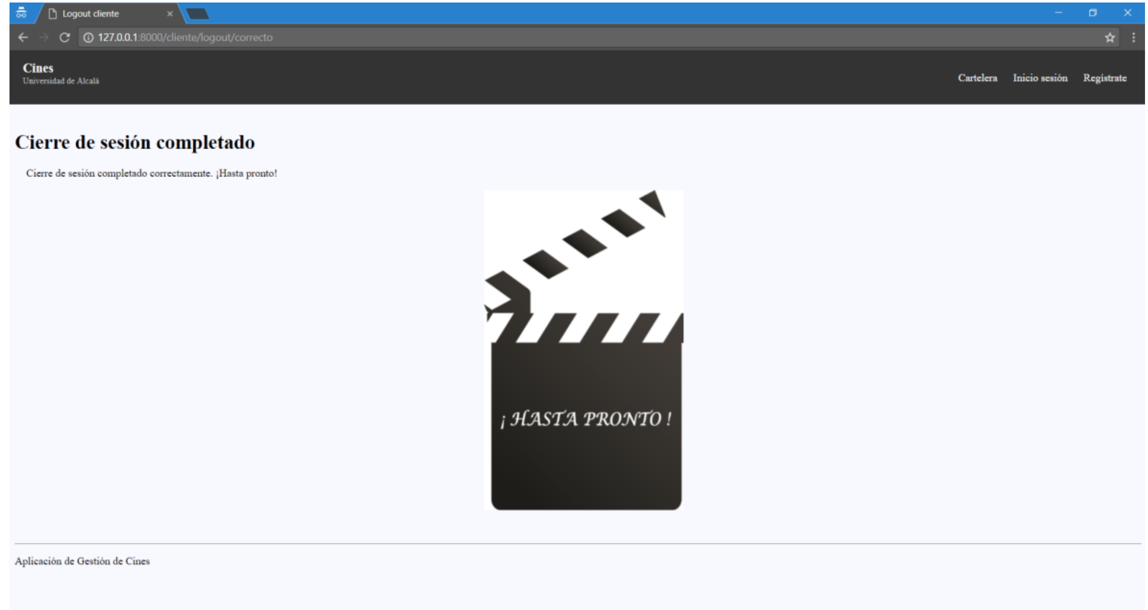


Ilustración 42 - Cierre de sesión en la aplicación.

Con todo lo anteriormente comentado, queda explicada la parte del cliente.

En cuanto a la **parte de administración**, para acceder a ella, desde la página principal, es necesario pinchar en el enlace '*Sitio de administración*'. Aparecerá una ventana de login para este apartado.

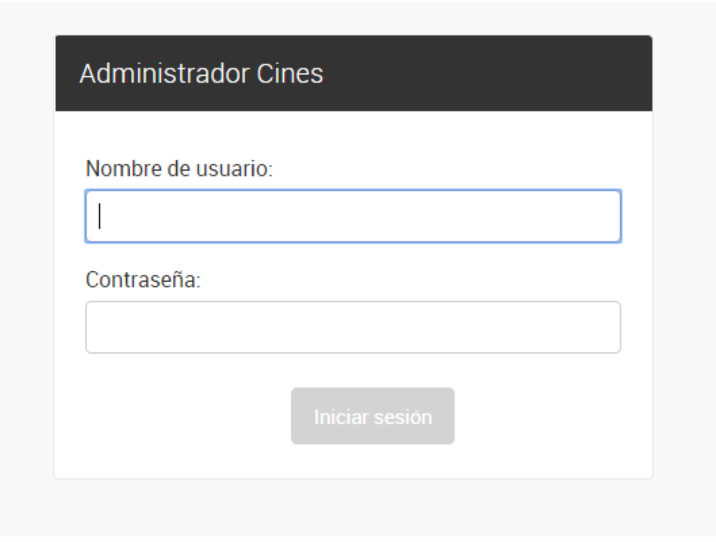


Ilustración 43 - Login del sitio de administración de la aplicación.

Si un cliente, que no tienen permisos de administración, intenta acceder a esta parte, le saldrá el siguiente mensaje:

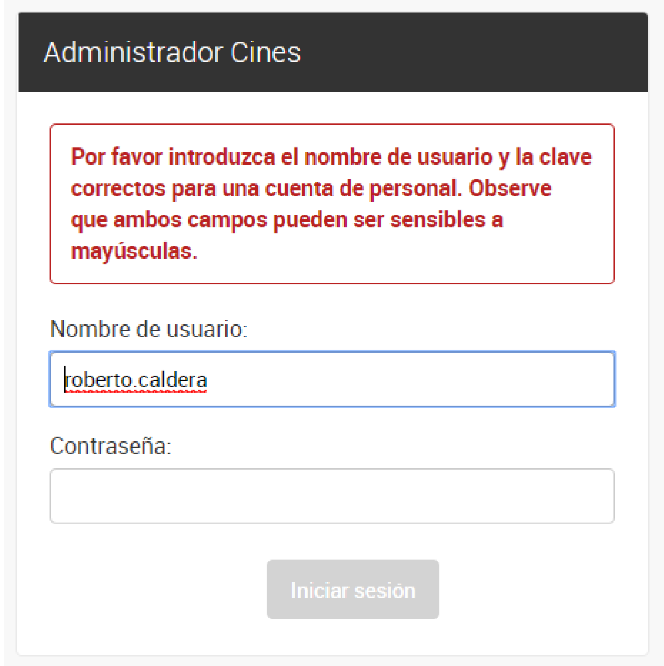


Ilustración 44 - Error de login de cliente en formulario del sitio de administración de la aplicación.

Si se introduce correctamente los datos del usuario administrador del sistema, será redirigido a la siguiente página de administración:

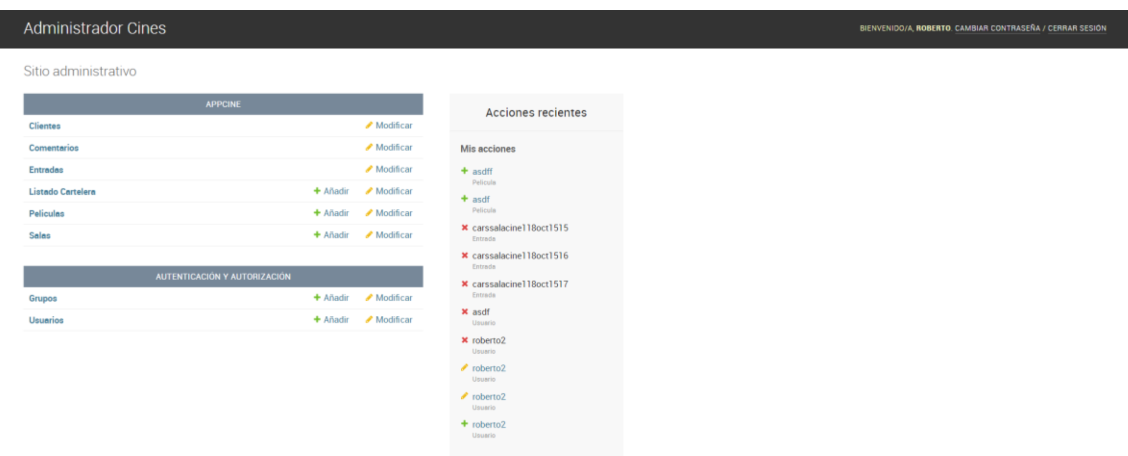


Ilustración 45 - Vista de la página de administración de la aplicación.

En ella puede realizar las siguientes gestiones:

I. Añadir, eliminar o modificar películas del sistema.
II. Añadir, eliminar o modificar salas del cine.
III. Gestionar la cartelera del cine.
IV. Visualizar y eliminar comentarios de películas.
V. Gestionar los usuarios de la aplicación para el sitio de administración.
VI. Observar los datos de los usuarios introducidos en la aplicación para la parte del cliente.

En cuanto a la gestión de películas, dispone de la siguiente información:

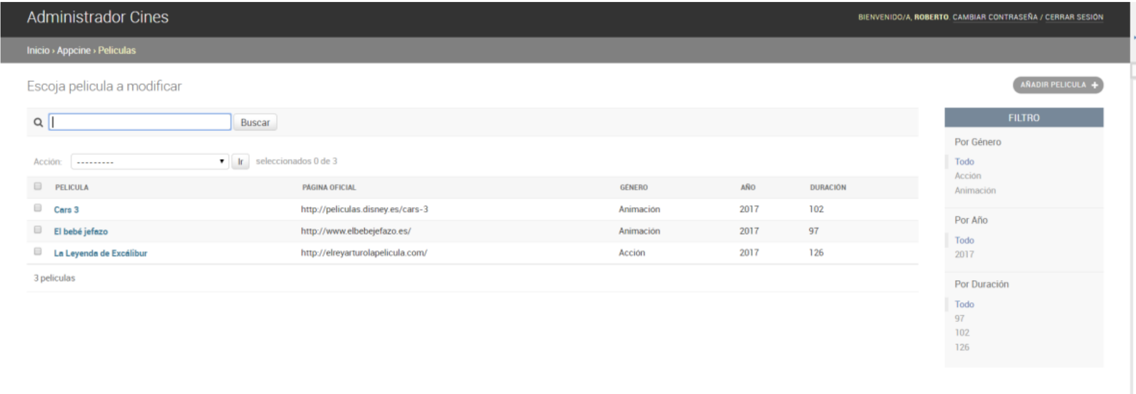


Ilustración 46 - Visión de las películas de la aplicación en el sitio del administrador.

Adicionalmente, si quiere dar de alta una nueva película, es posible realizarlo desde el enlace situado en la esquina superior. Le saldrá un formulario como el siguiente:

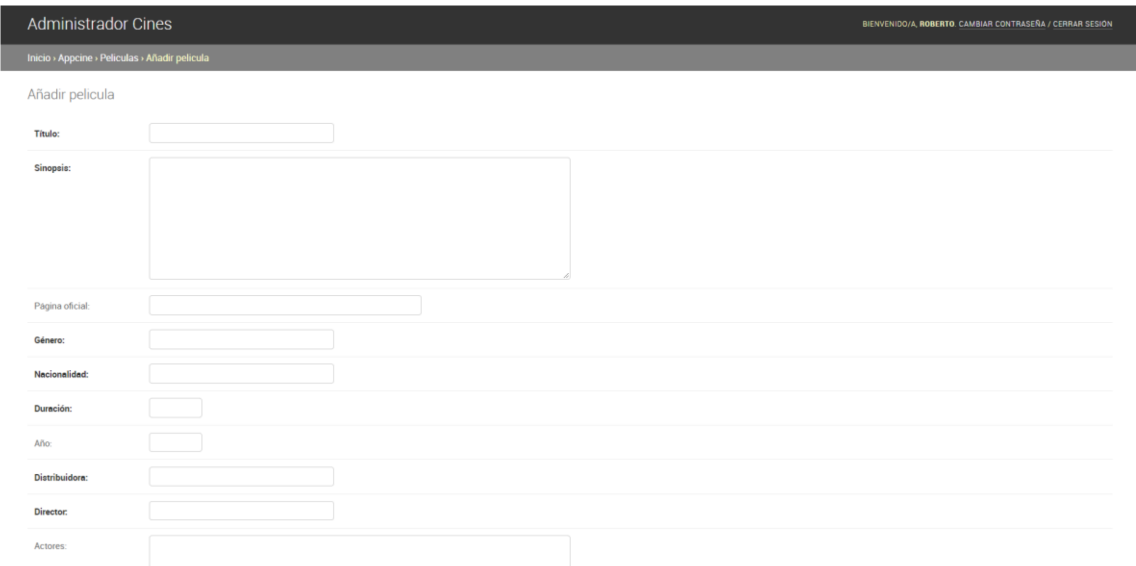


Ilustración 47 - Formulario registro película del administrador en la aplicación.

En este formulario deberá de introducir toda la información de la película solicitada.

La gestión de salas es similar a la que se acaba de exponer, pero con la información de las distintas salas disponibles del cine.

En cuanto a la cartelera, se puede visualizar al igual que en la sección anterior toda la información de las películas junto con la sala y fecha de proyección. Si se desea añadir una película a la cartelera, es posible mediante el enlace de la esquina superior derecha, con el siguiente formulario:

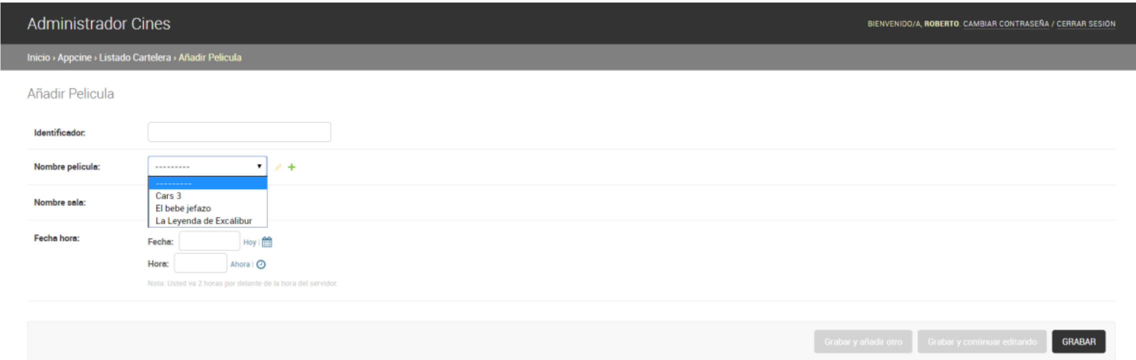


Ilustración 48 - Inserción de nueva sala en el sitio del administrador de la aplicación.

En el cual, deberá de introducir un identificador único de la sesión, junto con la película, sala, fecha y hora de la misma. Para elegir la película, aparecerá un desplegable con las que actualmente se encuentran registradas en la aplicación, al igual que sucede con las salas.

El apartado más importante del sitio del administrador es el de la gestión de los usuarios, en el cual se mostrará un listado de los usuarios registrados:

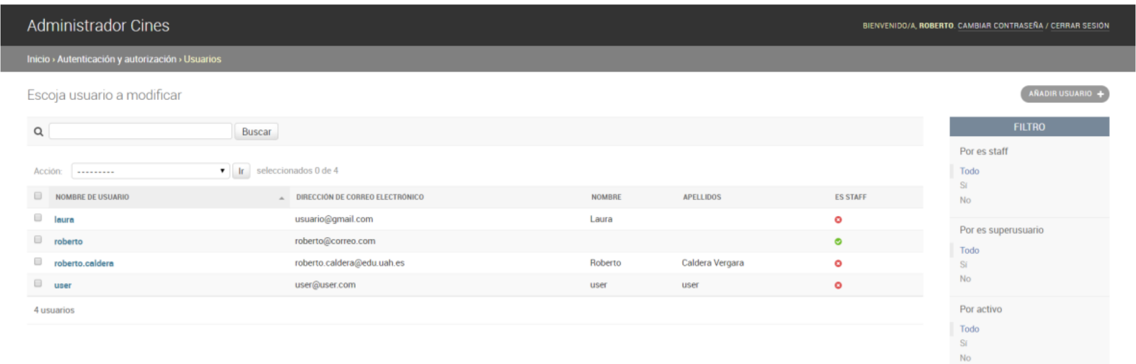


Ilustración 49 - Visión de los usuarios de la aplicación en el sitio del administrador.

Como se puede apreciar, en la columna '*ES STAFF*', hay dos opciones:

I. El tick verde significa que ese usuario tiene permisos para acceder al sitio de administración.
II. La cruz roja significa que ese usuario no dispone de permisos para entrar en este sitio.

Los superusuarios tienen la opción de crear nuevos usuarios para el sitio de administración. Para ello, en primer lugar, es necesario insertar un identificador junto con una contraseña. A continuación, se puede agregar más información como nombre, apellidos y correo electrónico, además de los permisos que tendrá. Generalmente, se seleccionarán como superusuarios, a no ser que se desee crear un usuario administrador con permisos especiales de crear solo películas u otros objetos del cine.

Se adjuntan los pasos para realizar dicha acción:

- Paso 1:

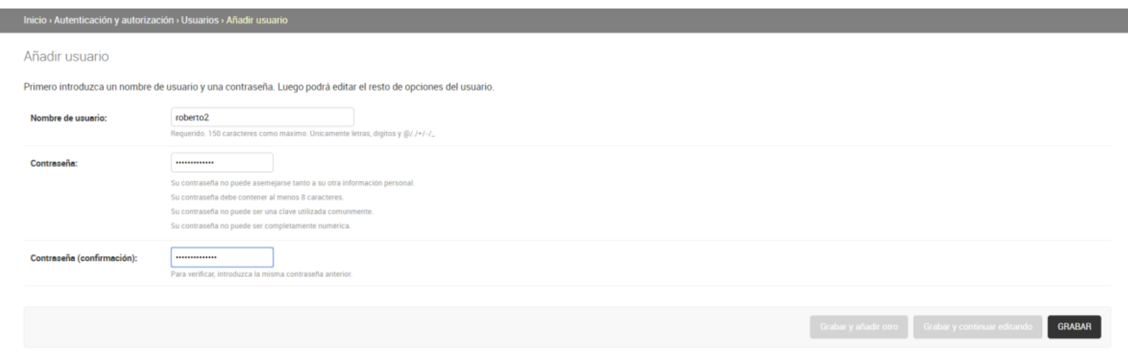


Ilustración 50 - Formulario de alta de usuario administrador en la aplicación.

- Paso 2:

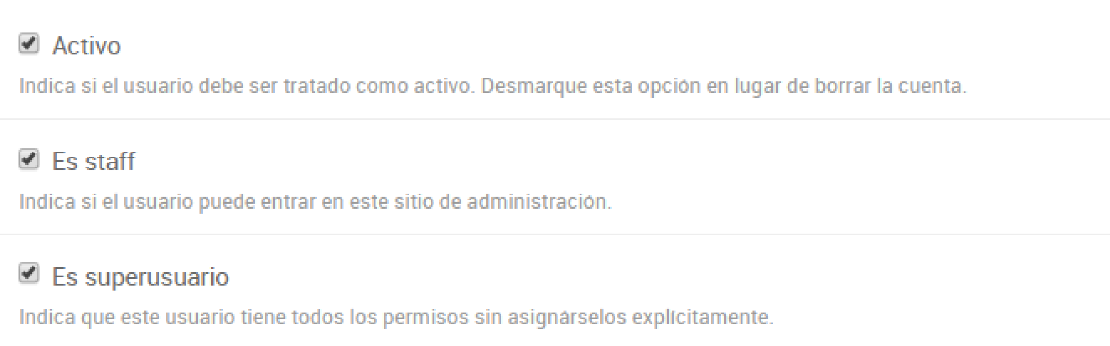


Ilustración 51 - Opciones de permisos para un usuario de la aplicación.

Es necesario activar las tres opciones para que el usuario creado pueda realizar todas las opciones que se han comentado del gestor de cines.

Si por el contrario se desea realizar un usuario administrador, con unos determinados permisos, se deben de elegir en el siguiente cuadro:

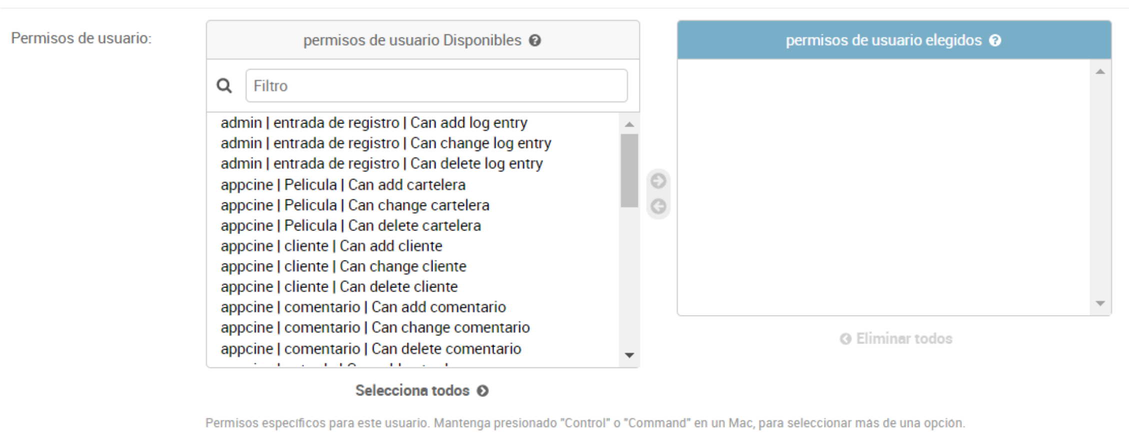


Ilustración 52 - Gestión de permisos para un usuario de la aplicación.

18 Conclusiones

Una vez que se ha dado por finalizado el Trabajo de Fin de Grado "*Estudio del Framework de Programación Web Django*", se extraen conclusiones sobre los distintos puntos aprendidos durante el desarrollo del mismo.

I. El empleo de un Framework a la hora del desarrollo Web facilita muchas ventajas a los programadores de este ámbito, ya que se les ofrece un esqueleto de la aplicación y ellos tienen que personalizarla y desarrollarla según sus necesidades partiendo de dicha base dada.

II. La utilización de un Framework Web aporta a los desarrolladores una mayor facilidad de entender la programación de la aplicación debido a que se conocen la estructura del Framework y, por lo tanto, pueden saber con desenvoltura donde se implementa cada una de las partes.

III. Relacionado con el punto anterior, cabe mencionar que dichos conocimientos sobre un Framework no es fácil adquirirlos ya que se necesita una cierta dedicación. También cabe comentar que la comprensión de un determinado Framework no implica conocer los demás.

IV. Tras realizar el estudio de Django, se puede deducir que debido a la utilización del patrón MTV, aporta una mayor reutilización del código, ofrece una mejor seguridad a la aplicación y facilidad del mantenimiento.

V. En cuanto a la capa del modelo, facilita mucho la tarea al programador debido al empleo del ORM del que dispone. A pesar de incorporar numerosas ventajas, a veces es necesario recurrir a incrustar código SQL puro cuando se requiere realizar una consulta más compleja. También permite incluir el motor de base de datos subyacente de la aplicación que el programador desee, sin ceñirse a un motor específico.

VI. De la capa de la vista, cabe resaltar su importancia ya que es la que decide que información ven los usuarios finales. En esta capa hay que mencionar la gran aportación que Django ofrece a los programadores incluyendo vistas por defecto para las acciones más repetidas en la programación Web.

VII. Después del estudio del sitio de administración que Django ofrece, se destaca que es muy sencillo de incluir en la aplicación, personalizarlo superficialmente y heredar de él. Gracias a dicha incorporación, permite ahorrar a los programadores la tarea tediosa de tener que desarrollar desde cero un sitio de administración. En cambio, si se desea manipular configuraciones más específicas de dicho sitio de administración, puede resultar complicado para programadores que no conozcan su estructura de interna exactamente.

VIII. En cuanto a la capa de las plantillas, que es la que determina la visibilidad de los datos de la aplicación, se destaca el uso del lenguaje DTL que Django ofrece para el desarrollo de plantillas de la aplicación y los distintos lugares posibles para realizar la carga de plantillas a una aplicación. También es necesario recalcar en este punto el uso de la herencia de plantillas, lo que aporta un mayor ahorro de código.

IX. De la seguridad del Framework, Django es consciente de la importancia que tiene este punto en la actualidad, y aporta diversos mecanismos de seguridad por defecto para distintas partes de la aplicación, lo cual permite al programador no tener que fijarse expresamente de esos detalles.

X. Como conclusión final, Django es un framework que facilita el desarrollo ágil de aplicaciones web, por lo que, con un previo estudio del mismo, se puede sacar mucho partido a las distintas herramientas que éste ofrece para construir una aplicación Web robusta.

19 Bibliografía

[1] "*Arquitectura y Diseño de Sistemas Web y C/S*", notas de clase 780041, Departamento Ciencias de la Computación, Universidad de Alcalá de Henares, invierno 2016.

[2] "*Protocolo de transferencia de hipertexto*", Wikipedia, 2017. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto. [Accedido: 25-Ago-2017]

[3] Gutierrez J. J., **"***¿Qué es un Framework Web?"*, [En línea] Disponible en: http://www.cssblog.es/guias/Framework.pdf.

[4] "*Patrones Software*", notas de clase 780042, Departamento Ciencias de la Computación, Universidad de Alcalá, invierno 2016.

[5] Holovaty A. y otros., **"***El Libro de Django*"., Segunda Edición., Apres., 2007., 381p.

[6] Django Software Foundation**,** 2017, Django (Versión 1.11). Disponible en: https://djangoproject.com

[7] Documentación oficial de Python, versión 3. Disponible en: https://docs.python.org/3/.

[8] "*Framework para aplicaciones web*", Wikipedia, 2017. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Framework_para_aplicaciones_web. [Accedido: 25-Jun-2017]

[9] González R.**, "***Python para Todos*"., Segunda Edición., Madrid España., 2010., 108p.

[10] García S**.**, "*La guía definitiva de Django"*, Amazon Web Services, 2017., 598p.

[11] B. A. B. P. Paul Davil Cumba Armijos, "*Análisis de python con django frente a ruby on rails para desarrollo ágil de aplicaciones web. Caso práctica: Dech."*, tesis de grado, Riobamba, 2012. [Accedido: 5-Ago-2017]

[12] "*The Python Tutorial"*. Marzo 2017. [En línea]. Disponible en: https://docs.python.org/3.6/tutorial/index.html [Accedido: 10-Ju-2017]

[13] **"***Django (framework)***",** Wikipedia, 2017. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Django_(framework). [Accedido: 30-Jun-2017].

[14] **"***Python***",** Wikipedia, 2017. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Python. [Accedido: 25-Jun-2017].