

## TEXT CLASS REVIEW

### TEMAS A TRATAR EN EL CUE:

- `django.contrib.admin`
- `django.contrib.auth`
- `django.contrib.contenttypes`
- `django.contrib.sessions`
- `django.contrib.messages`
- `django.contrib.staticfiles`

### REVISANDO LAS APLICACIONES PREINSTALADAS DJANGO

Por defecto, `INSTALLED_APPS` contiene las siguientes apps, todas provistas por Django:

- `django.contrib.admin` – El sitio de administración.
- `django.contrib.auth` – Sistema de autenticación.
- `django.contrib.contenttypes` – Un framework para tipos de contenido.
- `django.contrib.sessions` – Un framework para manejo de sesiones.
- `django.contrib.messages` – Un framework de mensajes.
- `django.contrib.staticfiles` – Un framework para manejar los archivos estáticos.

### DJANGO.CONTRIB.ADMIN

La característica más popular de Django para marco de Python es su sitio de administración integrado, que permite a los usuarios internos administrar datos, sin necesidad de crear una utilidad especial.

Django Admin es un panel de control que permite administrar todos los modelos. Es decir, el código ya está hecho por nosotros, los formularios también, y lo mejor es que cada uno está dispuesto para cada tipo de dato.

## CREAR SUPERUSUARIO

Para administrar toda la información, necesitamos crear un usuario con permisos elevados. Esto se consigue creando un superusuario con el siguiente comando (dentro de nuestro proyecto):

```
1 python manage.py createsuperuser
```

Ingresamos el nombre usuario que queramos, y presionamos enter:

```
1 Username: admin
```

Debemos ingresar la dirección de email:

```
1 Email address: admin@example.com
```

El último paso es ingresar la contraseña. Debemos hacerlo dos veces, la segunda como una confirmación de la primera:

```
1 Password: *****
2 Password (again): *****
3 Superuser created successfully.
```

## AGREGAR MODELO A DJANGO ADMIN

Por defecto, nuestro modelo (y todos los que vayamos a crear) no se puede administrar desde el panel de control, pues debemos especificarlo. Para ello vamos a un ejemplo `gastos/admin.py`, y lo registramos. Primero, importamos al modelo:

```
1 from .models import Gasto
```

Después, lo registramos:

```
1 admin.site.register(Gasto)
```

De manera que `gastos/admin.py` se ve así:

```
1 from django.contrib import admin
```

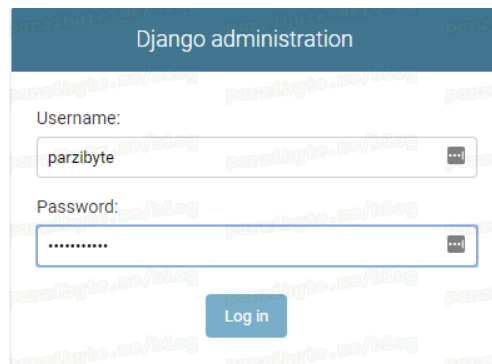
```
2 from .models import Gasto # Importar nuestro modelo
3 # Register your models here.
4 # Registrarlo dentro de admin:
5 admin.site.register(Gasto)
```

Ahora ejecutamos nuestra app:

```
1 python manage.py runserver
```

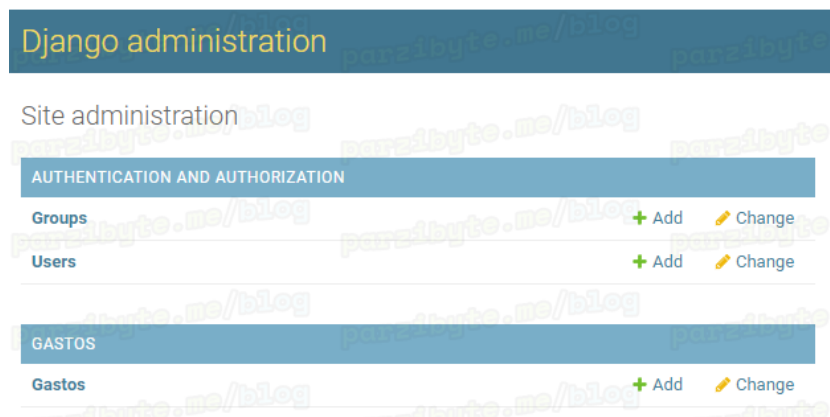
## ENTRAR A DJANGO ADMIN

Navegamos a `/admin` dentro de la URL de nuestra app (si no has movido nada, es `localhost:8000/admin`). Se procede a iniciar sesión con las credenciales creadas al generar el superusuario:



Inicio de sesión en Django Admin

Al inicio se muestran los usuarios y grupos que pueden acceder al panel de control. Abajo, nuestros modelos:



Finalmente, se pueden hacer todas las operaciones de la base de datos: insertar, mostrar, editar y eliminar. Esto no está pensado para el usuario final, sino para administradores, pero proporciona una manera muy fácil e intuitiva de administrar todos nuestros modelos.

## DJANGO.CONTRIB.AUTH

Django cuenta con un sistema de autenticación de usuarios predeterminado, el cual permite implementar de forma sencilla las funcionalidades básicas de inicio de sesión, recuperación y actualización de contraseñas, y cierre de sesión.

El módulo `django.contrib.auth` que agregamos previamente a los `INSTALLED_APPS` de nuestra aplicación en el archivo `app/settings.py` habilita el modelo de usuario predeterminado de Django, el cual podemos importar de la siguiente forma:

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

Este modelo cuenta los campos básicos para representar un Usuario en una plataforma web como `username`, `first_name`, `last_name`, `email`, `password`, entre otros. Lo utilizaremos para crear nuestro sistema de usuarios.

En el sitio de administración de Django ya fue creado el superusuario, el cual ya está autenticado y tiene todos los permisos, así que el siguiente paso es crear un usuario de prueba que represente un usuario normal del sitio.

```
1 from django.contrib.auth.models import User
2 # Create user and save to the database
3 user = User.objects.create_user('myusername', 'myemail@crazymail.com',
4 'mypassword')
5 # Update fields and then save again
6 user.first_name = 'John'
7 user.last_name = 'Citizen'
8 user.save()
```

## DJANGO.CONTRIB.CONTENTTYPES

Django cuenta con un framework para tipos de contenido. `ContentType` sirve para relacionar modelos con otros modelos, como si fuera una foreign key (llave foránea), pero con la ventaja de que el tipo de modelo con el cual lo relacionemos puede ser diferente para cada entrada de la tabla.

Imagina una sencilla red social, donde tenemos diferentes tipos de contenido; un modelo para videos, uno para imágenes, y uno para textos. `ContentType` permite crear un modelo que haga referenciar a cualquiera de nuestros tres modelos de una forma rápida.

En el corazón de la aplicación `contenttypes` se encuentra el modelo `ContentType`, en `django.contrib.contenttypes.models.ContentType`. Las instancias de `ContentType` representan y almacenan información sobre los modelos instalados en su proyecto, y las nuevas instancias de `ContentType` se crean automáticamente cada vez que se instalan nuevos modelos.

Las instancias de `ContentType` tienen métodos para devolver las clases de modelos que representan, y para consultar objetos de esos modelos. `ContentType` también tiene un administrador personalizado que agrega métodos para trabajar con `ContentType`, y para obtener instancias de `ContentType` para un modelo en particular.

Las relaciones entre sus modelos y `ContentType` también se pueden utilizar para habilitar relaciones "genéricas" entre una instancia de uno de sus modelos, e instancias de cualquier modelo que haya instalado.

## INSTALACIÓN DEL MARCO DE TIPOS DE CONTENIDO

Generalmente, es recomendable tener instalado el framework `contenttypes`; varias de las otras aplicaciones incluidas de Django lo requieren, la aplicación de administración lo utiliza para registrar el historial de cada objeto añadido o cambiado a través de la interfaz de administración, y el `authentication` framework de Django lo utiliza para vincular los permisos de usuario a modelos específicos.

## MODELO CONTENTTYPE

`class ContentType`: cada instancia de `ContentType` tiene dos campos que, en conjunto, describen de forma exclusiva un modelo instalado:

- **app\_label**: el nombre de la aplicación de la que forma parte el modelo. Esto se toma del atributo **app\_label** del modelo, e incluye solo la última parte de la ruta de importación de Python de la aplicación; **django.contrib.contenttypes**, por ejemplo, se convierte en una **app\_label** de tipos de **contenttypes**.
- **Model**: el nombre de la clase modelo.

Además, se dispone de la propiedad **name**: el nombre legible para humanos del tipo de contenido. Esto se toma del atributo **verbose\_name** del modelo.

Revisemos un ejemplo para ver cómo funciona esto. Si ya tiene instalada la aplicación **contenttypes**, y luego agrega el lugar de aplicación a su configuración **INSTALLED\_APPS** y ejecuta **manage.py migrate** para instalarla, el modelo **django.contrib.sites.models.Site** se instalará en su base de datos. Junto con él, se creará una nueva instancia de **ContentType** con los siguientes valores:

**app\_label** se establecerá en **'sites'** (la última parte de la ruta de Python **django.contrib.sites**).

**model** se establecerá en **'site'**.

## DJANGO.CONTRIB.SESSIONS

Las sesiones son el mecanismo que usa Django (y la mayor parte de Internet) para guardar registro del estado entre el sitio y un navegador en particular. Éstas permiten almacenar información arbitraria por navegador, y tenerla disponible para el sitio cuando el navegador se conecta. Cada pieza individual de información asociada con una sesión se conoce como "clave", que se usa tanto para guardar como para recuperar la información.

Django usa una cookie que contiene un id de sesión específica para identificar cada navegador y su sesión asociada con el sitio. La información real de la sesión se guarda por defecto en la base de datos del sitio (esto es más seguro que guardar la información en una cookie, donde es más vulnerable para los usuarios maliciosos). Puedes configurar Django para guardar la información de

sesión en otros lugares (caché, archivos, cookies "seguras"), pero la opción por defecto es una buena opción y relativamente segura.

La configuración está establecida en las secciones `INSTALLED_APPS` y `MIDDLEWARE` del archivo del proyecto (`locallibrary/locallibrary/settings.py`), tal como se muestra abajo:

```
1 INSTALLED_APPS = [  
2     ...  
3     'django.contrib.sessions',  
4     ...  
5  
6 MIDDLEWARE = [  
7     ...  
8     'django.contrib.sessions.middleware.SessionMiddleware',  
9     ...
```

Hay tres reglas muy sencillas para usar eficazmente las sesiones en Django:

- Debes usar sólo cadenas de texto normales como valores de clave en `request.session`, en vez de, por ejemplo, enteros, objetos, entre otros. Esto es más un convenio que una regla en el sentido estricto, pero merece la pena seguirla.
- Los valores de las claves de una sesión que empiecen con el carácter subrayado están reservadas para uso interno de Django. En la práctica, sólo hay unas pocas variables así, pero, a no ser que sepas lo que estás haciendo (y estés dispuesto a mantenerte al día en los cambios internos de Django), lo mejor es evitar usar el carácter subrayado como prefijo en tus propias variables; eso impedirá que Django pueda interferir con tu aplicación,
- Nunca reemplaces `request.session` por otro objeto, ni accedas o modifiques sus atributos. Utilízalo sólo como si fuera un diccionario.

Revisemos un ejemplo rápido.

Esta vista simplificada define una variable `has_commented` como True después de que el usuario haya publicado un comentario. Es una forma sencilla (aunque no particularmente segura) de impedir que el usuario publique dos veces el mismo comentario:

```
1 def post_comment(request, new_comment):  
2     if request.session.get('has_commented', False):
```

```
3     return HttpResponse("You've already commented.")
4     c = comments.Comment(comment=new_comment)
5     c.save()
6     request.session['has_commented'] = True
7     return HttpResponse('Thanks for your comment!')
```

Esta vista simplificada permite que un usuario se identifique como tal en nuestras páginas:

```
1 def login(request):
2     try:
3         m =
4 Member.objects.get(username__exact=request.POST['username'])
5         if m.password == request.POST['password']:
6             request.session['member_id'] = m.id
7             return HttpResponse("You're logged in.")
8     except Member.DoesNotExist:
9         return HttpResponse("Your username and password didn't
10 match.")
```

Y ésta le permite cerrar o salir de la sesión:

```
1 def logout(request):
2     try:
3         del request.session['member_id']
4     except KeyError:
5         pass
6     return HttpResponse("You're logged out.")
```

## DJANGO.CONTRIB.MESSAGES

El marco de mensajes de Django nos permite almacenar mensajes temporales en una solicitud, para luego recuperarlos y mostrarlos en una solicitud posterior (generalmente es la siguiente).

## CONFIGURACIÓN

El archivo predeterminado de `settings.py` ya tiene habilitado el soporte para mensajes. Pero puedes verificar que todo esté configurado correctamente, teniendo en cuenta lo siguiente:

```
1 INSTALLED_APPS = [
2     ...
3     "django.contrib.messages",
```



```
4 ]
5
6 MIDDLEWARE = [
7     ...
8     "django.contrib.sessions.middleware.SessionMiddleware",
9     "django.contrib.messages.middleware.MessageMiddleware",
10 ]
```

Y en tu variable **TEMPLATES** en la lista **context\_processors** agregar la siguiente cadena si no se encuentra:

```
1 "django.contrib.messages.context_processors.messages"
```

## NIVEL DE MENSAJES

Los niveles de mensajes nos permiten agrupar mensajes por su tipo, y posteriormente filtrarlos o mostrarlos de manera diferente en las vistas o plantillas. Por ejemplo: podemos mostrar el **ERROR** de color rojo, y el **SUCCESS** de color verde.

A continuación, se muestran todos los niveles de mensaje y sus respectivas etiquetas (tags):

CONSTANTE	NIVEL	ETIQUETA CSS	PROPÓSITO
DEBUG	10	debug	Mensajes relacionados con el desarrollo que se ignorarán (o eliminarán) en una implementación de producción
INFO	20	info	Mensajes informativos para el usuario
SUCCESS	25	success	Una acción fue exitosa, por ejemplo, "Su perfil se actualizó con éxito"
WARNING	30	warning	No ocurrió una falla pero puede ser más adelante
ERROR	40	error	Una acción no tuvo éxito o se produjo algún otro error

Por defecto, Django muestra niveles desde 20 a más. Es decir, todos menos **DEBUG**. En caso de que quieras cambiar este comportamiento, puedes abrir tu archivo **settings.py**, y agregar lo siguiente:

```
1 from django.contrib.messages import constants as message_constants
```

```
2 MESSAGE_LEVEL = message_constants.DEBUG
```

De esta forma podrás tomar valores desde el 10 en adelante (todos básicamente).

## DJANGO.CONTRIB.STATICFILES

El framework de archivos estáticos recopila archivos estáticos de cada una de sus aplicaciones (y cualquier otro lugar que especifique) en una única ubicación que se pueda servir fácilmente en producción. `django.contrib.staticfiles` expone tres comandos de administración:

## COLLECTSTATIC

```
1 django-admin collectstatic
```

Recopila los archivos estáticos en `STATIC_ROOT`. Los nombres de archivos duplicados se resuelven de manera predeterminada, de forma similar a cómo funciona la resolución de la plantilla: se utilizará el archivo que se encuentra por primera vez en una de las ubicaciones especificadas. Si está confundido, el comando `findstatic` puede ayudarlo a mostrar qué archivos se encuentran.

En posteriores `collectstatic` carreras (si `STATIC_ROOT` no está vacío), los archivos se copian sólo si tienen una mayor marca de tiempo modificado que la del archivo en `STATIC_ROOT`. Por lo tanto, si se elimina una aplicación de `INSTALLED_APPS`, es una buena idea utilizar la opción `collectstatic --clear` para eliminar archivos estáticos obsoletos.

Los archivos se buscan utilizando los `enabled finders`. El valor predeterminado es buscar en todas las ubicaciones definidas en `STATICFILES_DIRS`, y en el directorio `'static'` de aplicaciones especificado por la configuración `INSTALLED_APPS`.

El comando de gestión `collectstatic` llama al método `post_process()` de `STATICFILES_STORAGE` después de cada ejecución, y pasa una lista de rutas que ha encontrado el

comando de gestión. También recibe todas las opciones de línea de comandos de **collectstatic**. **ManifestStaticFilesStorage** lo utiliza de forma predeterminada.

De manera predeterminada, los archivos recopilados reciben permisos de **FILE\_UPLOAD\_PERMISSIONS**, y los directorios recopilados reciben permisos de **FILE\_UPLOAD\_DIRECTORY\_PERMISSIONS**. Si desea diferentes permisos para estos archivos y / o directorios, puede subclasificar cualquiera de las clases de almacenamiento de archivos estáticos, y especificar los **file\_permissions\_mode** y / o **directory\_permissions\_mode**, respectivamente.

## FINDSTATIC

```
1 django-admin findstatic staticfile [staticfile ...]
```

Busca uno o más caminos relativos con los buscadores habilitados.

## RUNSERVER

```
1 django-admin runserver [addrport]
```

Invalida el comando core runserver si la aplicación staticfiles está instalada, y agrega la publicación automática de archivos estáticos. La publicación de archivos no se ejecuta a través de **MIDDLEWARE**.

El comando **Runserver** añade estas opciones:

## NOSTATIC

Use la opción **--nostatic** para deshabilitar la publicación de archivos estáticos con la aplicación **staticfiles** por completo. Esta opción solo está disponible si la aplicación **staticfiles** está en la configuración **INSTALLED\_APPS** de su proyecto .

Ejemplo:

```
1 $ django-admin runserver --nostatic
2 ... \> django-admin runserver --nostatic
```

## INSECURE

Use la opción `--insecure` para forzar la publicación de archivos estáticos con la aplicación `staticfiles`, incluso si la configuración `DEBUG` es `False`. Al emplear esto, reconoce el hecho de que es extremadamente ineficiente, y probablemente inseguro. Solo está destinado al desarrollo local, nunca debe usarse en producción, y está disponible si la aplicación `staticfiles` está en la configuración `INSTALLED_APPS` de su proyecto. `insecure` no funciona con `ManifestStaticFilesStorage`.

Ejemplo:

```
1 $ django-admin runserver --insecure
2 ... \> django-admin runserver --insecure
```