AWAKELAB

Acceso a Bases de Datos con Python/Django

Reconoce las aplicaciones preinstaladas con el motor Django distinguiendo su utilidad como apoyo al desarrollo

Objetivo de la jornada

1. Reconoce las aplicaciones preinstaladas de Django y su utilidad como apoyo al desarrollo
2. Inspecciona el modelo Django para aplicaciones preinstaladas

Aplicaciones Django preinstaladas

Revisando las aplicaciones preinstaladas Django

Hemos visto que Django viene con algunas aplicaciones pre-instaladas, son aquellas que se encuentran en **settings.py**, nosotros agregamos **'<nuestras aplicaciones>'** en la misma lista **INSTALLED_APPS**

```
... INSTALLED_APPS
= [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    '<nuestras aplicaciones>'
]
...
```

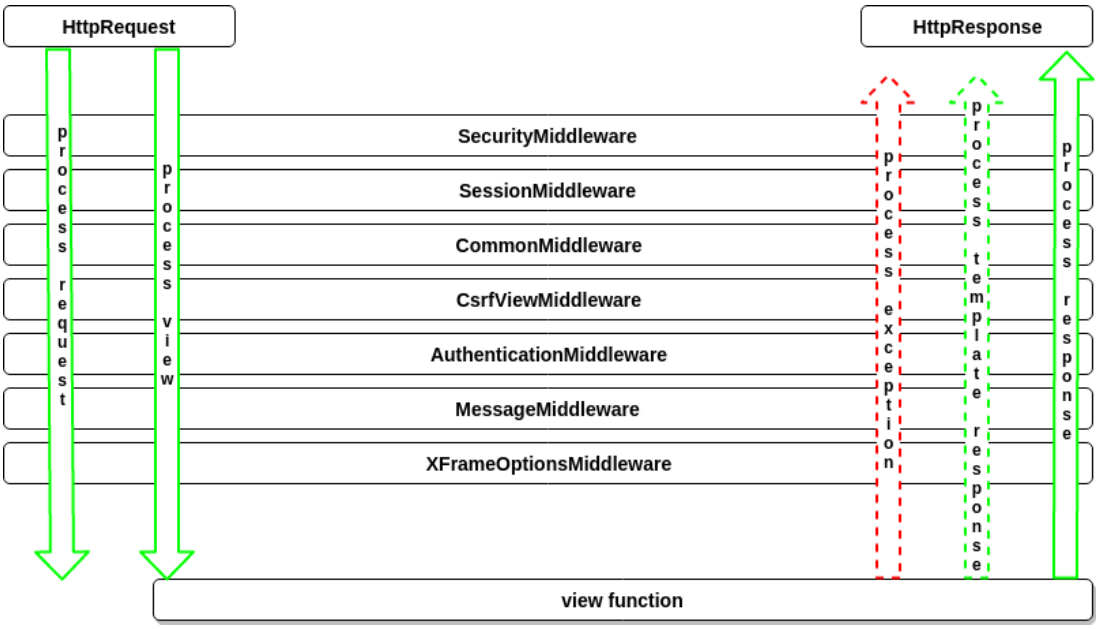
En este apartado vamos a tratar con aquellas aplicaciones que vienen instaladas de la mano con Django. Una de las muchas fortalezas de Python es su filosofía de “baterías incluidas”: cuando instala Python, viene con una gran biblioteca estándar de paquetes que puede comenzar a usar de inmediato, sin tener que descargar nada más. Django tiene como objetivo seguir esta filosofía e incluye su propia "biblioteca estándar" de complementos (Django Standard Library) útiles para tareas comunes de desarrollo web.

La biblioteca estándar de Django se encuentra en el paquete **django.contrib**. Dentro de cada subpaquete hay una pieza separada de funcionalidad adicional. Estas piezas no están necesariamente relacionadas, pero algunos subpaquetes **django.contrib** pueden requerir otros.

No hay requisitos estrictos para los tipos de funcionalidad en **django.contrib.** Algunos de los paquetes incluyen modelos (y por lo tanto requieren que instale sus tablas de base de datos en su base de datos - en el capítulo de migraciones vimos que inicialmente se crean una serie de tablas por defecto), pero otros consisten únicamente en middleware o etiquetas de plantilla (tags).

La única característica que tienen en común los paquetes **django.contrib** es esta: si eliminara el paquete **django.contrib** por completo, aún podría usar las características fundamentales de Django sin problemas. Cuando los desarrolladores de Django agregan una nueva funcionalidad al framework, usan esta regla para decidir si la nueva funcionalidad debe residir en **django.contrib** o en otro lugar.

El middleware, como vimos anteriormente de forma superficial, es un conjunto de ganchos (hooks) en el procesamiento de solicitudes/respuestas de Django. Es un sistema "plug-in" ligero y de bajo nivel capaz de alterar globalmente tanto la entrada como la salida de Django.



Un componente de middleware es simplemente una clase de Python que se ajusta a una determinada API. Lo podemos comprobar, por ejemplo, dando un vistazo al archivo **env/lib/python3.8/site-packages/django/contrib/sessions/middleware.py**

y los métodos que implementa. Existe middleware que es parte incorporada de Django como se puede ver en el paquete **django.middleware**

Para activar middleware basta con incluirlo en la lista **MIDDLEWARE** de **settings.py** cuando creamos un proyecto con **start-project** se instala middleware por defecto como se puede ver en dicho archivo.

En teoría, una instalación de Django no requiere ningún middleware: la lista **MIDDLEWARE** puede estar vacía, pero lo recomendable es contar con tales utilidades. En versiones anteriores de Django la variable era una tupla llamada **MIDDLEWARE_CLASSES**

Métodos de middleware

Ahora que sabemos en qué consiste el middleware con más claridad y también cómo activarlo para hacerlo disponible en **settings.py**, echemos un vistazo a algunos de los métodos disponibles más importantes que las clases de middleware pueden definir.

Inicializador:_init_____(self)

El método**___init___()** es útil para realizar una configuración de todo el sistema para una clase de middleware determinada.

Por motivos de rendimiento, una instancia de cada clase middleware es activada solo una vez por proceso de servidor. Esto significa que **__init __()** se llama solo una vez, al iniciar el servidor, y no para solicitudes individuales.

Una razón común para implementar un método **__init _()** es verificar si el middleware es realmente necesario. Si **init ()** genera **django.core.exceptions.MiddlewareNotUsed**, Django eliminará el middleware de la pila de middleware. Se puede utilizar esta función para comprobar si hay algún software que requiera la clase de middleware, o comprobar si el servidor está ejecutando el modo de depuración o cualquier otra situación de entorno similar.

Si una clase de middleware define un método**_init_()**, el método no debe tomar argumentos aparte del **self** estándar.

django.contrib.messages

La función de este middleware es almacenar y recuperar mensajes temporales basados en cookies o sesiones. Para disponer de este framework necesitamos tenerlo configurado en **settings.py**

```
INSTALLED_APPS = [
        ...
    'django.contrib.messages',
        ...
]

MIDDLEWARE = [
    ...
    'django.contrib.messages.middleware.MessageMiddleware',
        ...
]

TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ... 'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Vamos a ver un ejemplo, donde al crear un usuario le informamos al usuario que la operación se llevó a cabo exitosamente.

En **views.py** tenemos

```
...
class DepartamentoCreate(CreateView):
        ...
    def form_valid(self, form):
      messages.info(self.request, 'Departamento creado exitosamente')
      return super().form_valid(form)
```

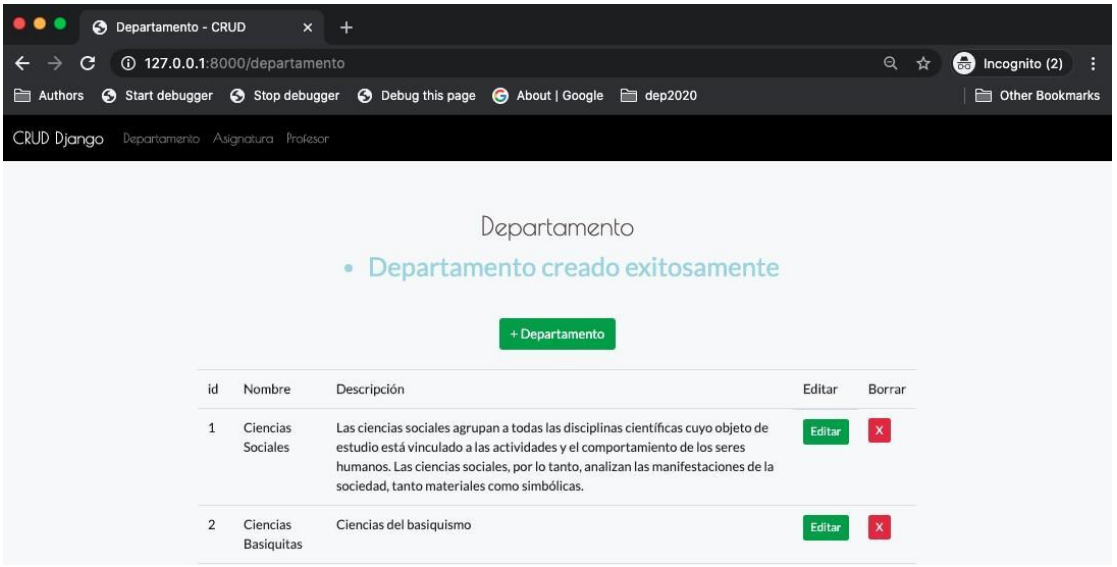
Aplicamos un poco de estilo en **styles.css**, utilizaremos en este caso un mensaje de información

```
.messages {
    color: lightblue;
    font-size: 2rem;
}
```

En nuestra template podemos incluir este código tal cual se ejemplifica en la documentación de Django:

```
...
                                        <div class="col-md-12">
                                            {% if messages %}
                                                <ul class="messages">
                                                    {% for message in messages
                                                    %}
                                                        <li{% if message.tags
                                                        %} class="{{
                                                        message.tags }}"{%
endif %}>{{ message }}</li>
                                                    {% endfor %}
                                                </ul>
                                            {% endif %}
                                        </div>
...
```

Luego de crear el departamento **"Ciencias Basiquitas"** podemos ver el mensaje de información:



Es importante mencionar el "context processor" utilizado por el framework messages, el cual configuramos en **settings.py** dentro de **TEMPLATES**.

Un context processor es una función de Python que toma el objeto **request** como argumento y devuelve un diccionario que se agrega al contexto de la solicitud. Resultan útiles cuando necesita que algo esté disponible globalmente para todas las plantillas.

Un uso de un procesador de contexto es cuando siempre desea insertar ciertas variables dentro de su plantilla. En lugar de escribir código para insertarlas en cada

vista, simplemente puede escribir un procesador de contexto y agregarlo a **TEMPLATES**

django.contrib.sessions

El objetivo de Django es ser eficaz, por lo que viene con un framework **session** (middleware framework) diseñado para evitar dificultades. Para contar con esta facilidad al igual que con el middleware anterior debemos habilitarlo **'django.contrib.sessions'** en **INSTALLED_APPS**; y **'django.contrib.sessions.middleware.SessionMiddleware'** en la lista **MIDDLEWARE**. También queda disponible si usted comienza su proyecto con **startproject**.

El "framework session" admite sesiones anónimas y de usuario que le permiten almacenar datos arbitrarios para cada visitante. Los datos de sesión se almacenan en el lado del servidor y las cookies contienen el ID de la sesión a menos que utilice un motor (engine) de sesión basado en cookies. El middleware de sesión gestiona el envío y la recepción de cookies. El motor de sesión predeterminado almacena los datos de la sesión en la base de datos, pero puede elegir entre diferentes motores de sesión.

El middleware de sesión hace que la sesión actual esté disponible en el objeto **request**. Puede acceder a la sesión actual utilizando **request.session**, tratándola como un diccionario de Python para almacenar y recuperar datos de la sesión. El diccionario de sesión acepta cualquier objeto Python que de forma predeterminada se pueda serializar en JSON. Puede establecer una variable en la sesión de esta forma:

```
request.session['foo'] = 'bar'
```

Se recuperar una variable sesión de la siguiente manera:

```
request.session.get('foo')
```

Puede eliminar una variable almacenada previamente en la sesión de la siguiente manera:

```
del request.session['foo']
```

Puede tratar **request.session** como un diccionario estándar de Python.

Cuando los usuarios inician sesión en un sitio, su sesión anónima se pierde y se crea una nueva sesión para los usuarios autenticados. Si almacena elementos en una sesión anónima que necesita conservar después de que el usuario inicia sesión, deberá copiar los datos de la sesión anterior en la nueva sesión.

Hay varias opciones que puede usar para configurar sesiones para su proyecto. La más importante es **SESSION_ENGINE** lo cual sobreescribe la variable en la configuración global de Django (tal como **TEMPLATES**, **INSTALLED_APPS**, etc.) Esta configuración le permite establecer el lugar donde se almacenan las sesiones. Por defecto, Django almacena sesiones en la base de datos usando el modelo **Session** de la aplicación **django.contrib.sessions**.

```
class Session(AbstractBaseSession):
    objects = SessionManager()
    ...


class AbstractBaseSession(models.Model):
    session_key = models.CharField(_('session key'), max_length=40,
primary_key=True)
    session_data = models.TextField(_('session data'))
    expire_date = models.DateTimeField(_('expire date'), db_index=True)
        ...
```

Django ofrece las siguientes opciones para almacenar datos de sesión **(sessions.backends)**:

- Sesiones de base de datos: los datos de la sesión se almacenan en la base de datos. Nota: este es el motor de sesión predeterminado.

- Sesiones basadas en archivos: los datos de la sesión se almacenan en el sistema de archivos.

- Sesiones en caché: los datos de la sesión se almacenan en un backend de caché. Puede especificar backends de caché mediante la configuración de **CACHES**. El almacenamiento de los datos de la sesión en un sistema de caché proporcionan el mejor rendimiento.

- Sesiones de base de datos en caché: los datos de la sesión se almacenan en una caché y una base de datos de escritura directa. Solo-lectura utiliza la base de datos si los datos aún no están en la caché.

- Sesiones basadas en cookies: los datos de la sesión se almacenan en las cookies que se envían al navegador.

Para un mejor rendimiento, se puede utilizar un motor de sesión basado en caché. Django soporta Memcached y puede encontrar backends de caché de terceros para Redis y otros sistemas de caché. Tales tópicos están fuera del alcance de este curso. Puede personalizar sesiones con configuraciones específicas. A continuación, se muestran algunas de las configuraciones importantes relacionadas con la sesión:

- **SESSION_COOKIE_AGE:** la duración de las cookies de sesión en segundos. El valor predeterminado es 1209600 (dos semanas).

- **SESSION_COOKIE_DOMAIN**: el dominio utilizado para las cookies de sesión. Configure esto con una cadena como **midominio.com** para habilitar las cookies entre dominios o use **None** para una cookie de dominio estándar (por defecto).

- **SESSION_COOKIE_SECURE:** un valor booleano que indica que la cookie solo debe enviarse si la conexión es HTTPS.

- **SESSION_EXPIRE_AT_BROWSER_CLOSE**: Un booleano que indica que la sesión debe caducar cuando se cierra el navegador.

- **SESSION_SAVE_EVERY_REQUEST:** un valor booleano que, si es **True**, guardará la sesión en la base de datos en cada solicitud. La caducidad de la sesión también se actualiza cada vez que se guarda.

Puede optar por utilizar sesiones de duración del navegador o sesiones persistentes mediante la variable **SESSION_EXPIRE_AT_BROWSER_CLOSE**. Esto variable es **False** de forma predeterminada, forzando la duración de la sesión al valor almacenado en la variable **SESSION_COOKIE_AGE**. Si configura **SESSION_EXPIRE_AT_BROWSER_CLOSE** como **True**, la sesión caducará cuando el usuario cierre el navegador y la configuración **SESSION_COOKIE_AGE** no tendrá ningún efecto. Puede usar el método **set_expiry()** de **request.session** para sobrescribir la duración de la sesión vigente.

Para ilustrar el uso de este middleware vamos a suponer que tenemos algunos computadores conectados a nuestro servidor. Si dispone de más de un computador conectado a su red privada (su teléfono puede ser uno) puede iniciar el servidor de desarrollo para aceptar conexiones de otros dentro de su red privada ejecutando:

```
$ python manage.py runserver 0.0.0.0:80
```

Además debe aceptar que ese computador tenga acceso a nuestro servidor de desarrollo, eso lo hacemos listando la ip del dispositivo en **ALLOWED_HOSTS**, en este caso tenemos un computador con la dirección IP **'192.168.178.158**

```
ALLOWED_HOSTS = ['192.168.178.158', '127.0.0.1']
```

Nota: no es necesario que realice este experimento con más de un dispositivo, sólo queremos ilustrar como funcionaría esta experiencia a un nivel más amplio, pero basta con su máquina de desarrollo. En este último caso solo tendrá un dispositivo llenando y enviando el formulario, y por lo tanto información solo de dicho pc.

Vamos a suponer que, en nuestra página, alguien tratando de jugar a ser simpático quiere escribir la palabra "rechazo" en el nombre cuando crea un departamento.

Nosotros tenemos la capacidad de saber la IP de cada máquina conectada a nuestro servidor (PC de desarrollo en este caso) o bien, nuestro PC es el terminal donde se ingresan datos y tiene horarios asignados para su uso en periodos bien determinados periodos por ciertas personas específicas.

Nuestro objetivo es descubrir a quien está tratando de ser simpático y para eso usaremos las sesiones almacenadas en nuestro servidor (estamos usando el motor por defecto que es usando base de datos).

En views vamos a detectar el incidente y la IP donde se produjo

```
...
class DepartamentoCreate(CreateView):
        ...
    def form_valid(self, form):

        if "rechazo" in self.request.POST["nombre"].lower():
            self.request.session['incidente'] =
timezone.now().strftime("%X")

        x_forwarded_for = self.request.META.get('HTTP_X_FORWARDED_FOR')
```

```
        if x_forwarded_for:
            ip = x_forwarded_for.split(',')[0]
        else:
            ip = self.request.META.get('REMOTE_ADDR')

        self.request.session'ip'] = ip
        ...
```

Nota: por si no se entiende este "snippet" que agregamos cubre el caso donde existe un servidor proxy, tal tópico es propio de un curso de redes y desarrollo.

Si revisamos en nuestra base datos tenemos dos sesiones, es decir dos computadores se han conectado a nuestra máquina. Ahora nos queda por saber si ingresaron la palabra que no nos agrada como broma.

```
municipio01=> SELECT * FROM django_session;

session_key                |             session_data
| expire_date
                           +                                                 +
bjnu768p5c7meg35ipp748vpkdcwk03e|eyJpbmNpZGVudGUiOiIxOToxNTo0NyIsImlwIjoiMTkyLjE2OC4xNzguMTg3I
n0:1kHXTX:aGwxnIq0Okq6nJH23JaSRSdBcMZZjYKnCFK9zgoY_zQ                | 2020-09-27
19:15:47.794254+00 nu3ck0fm3jz7t1iwn8wny4c1wcc28908|
.eJyrVkpJLUgsKknMTc0ryVeyQuXqKGXmJWemANmpQClDcysjQytjY5BwAYhvZK5nAISGSrUAaMEVhQ:1kHiSq:en1MGQf
pP6KbHCMjrcdCxEY4CBhe-VmpO7T8r0S_2H0 | 2020-09-28 06:59:48.688909+00
(2 rows)
```

La información tiene ese aspecto porque como dijimos anteriormente "El diccionario de sesión acepta cualquier objeto Python que de forma predeterminada se pueda serializar en JSON". Naturalmente la información en este formato "hash" no nos resulta útil.

Para facilitar nuestra investigación vamos a escribir un "comando de consola" que nos permita decodificar qué ha sucedido. Cree un archivo llamado **leer_sesion.py** en un directorio que usualmente aloja los comandos creados por el desarrollador ".../management/commands/" (debe crear tal carpeta)

Nuestro comando **leer_sesion** consiste en el siguiente código:

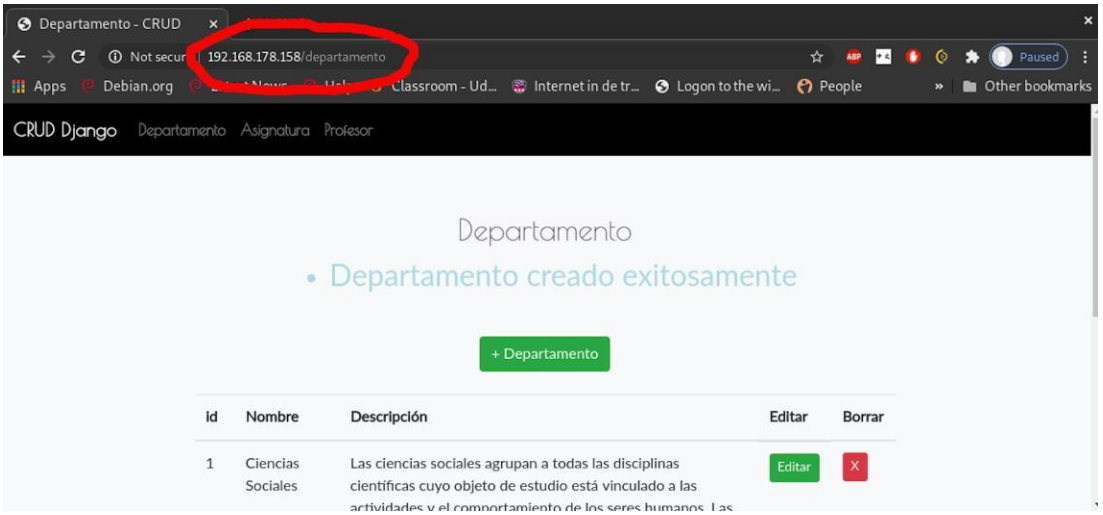
```
from django.core.management.base import BaseCommand
from django.contrib.sessions.models import Session
from django.utils import timezone

class Command(BaseCommand):
    help = "Buscamos sesiones sospechosas..."
```

```
    def handle(self, *args, **options):
        sessions = Session.objects.filter(expire_date__gt=timezone.now())
        for session in sessions:
            data = session.get_decoded()
            print("Incidente a las ",
            data["incidente"]) print(data)
```

Ahora nuestro comando es un comando más que tenemos disponible, tal cual los que hemos usado antes como **python manage.py migrate** o **python manage.py makemigrations**

```
$ python manage.py
leer_sesion Incidente a las
17:21:33
{'departamento': 'departamento', 'incidente': '17:21:33', 'ip':
'127.0.0.1'} Incidente a las 19:15:47{'incidente': '19:15:47', 'ip':
'192.168.178.187'}
```



En este caso nuestro servidor es **192.168.178.158**. Esto nos permitió detectar que desde las dos máquinas el último que ingresó datos trató de ser simpático y escribió "rechazo". Naturalmente podríamos agregar funcionalidades a este código para que nos reporte cuando tal evento suceda vía email, SMS, whatsapp, etc. esas son alternativas muy recomendables a realizar como ejercicios.

django.contrib.staticfiles

Django viene con un framework para administrar archivos estáticos. Al igual que con las otras aplicacioones o (mini) framworks. Debemos habilitarlo, asegúrese de tener **'django.contrib.staticfiles'** incluido en la lista **INSTALLED_APPS**.

Vamos a usar el directorio que viene configurado por defecto en **settings.py** para los archivos estáticos.

```
...
STATIC_URL = '/static/'
...
```

Necesitamos crear tal directorio en nuestra aplicación, y vamos a copiar un archivo **departamento.jpg** en tal carpeta. Nuestra estructura queda así:

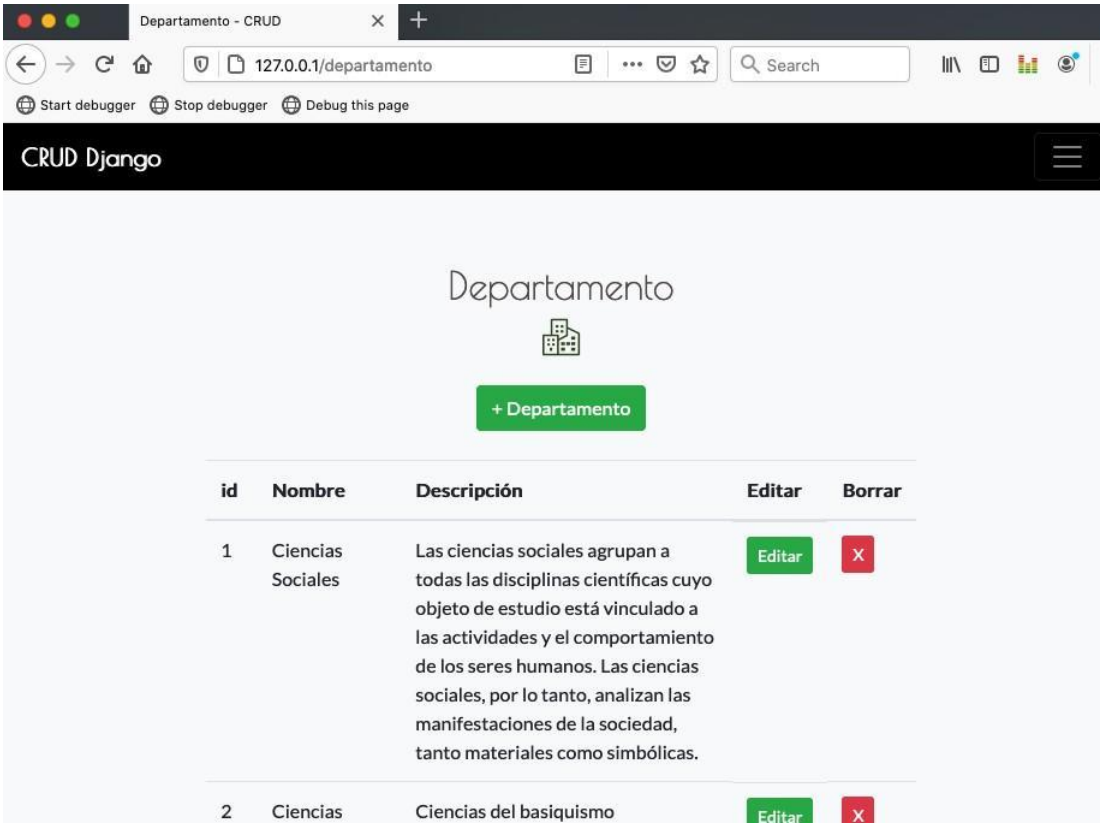
```
├──_init_.py
...
├── management
│   └── commands
│       └── leer_sesion.py
├── models.py
├── static
│   ├── comuna01
│   │   └── departamento.jpg
│   └── css
...
├── urls.py
└── views.py
```

Vamos a mostrar la siguiente imagen bajo nuestro encabezado

En nuestra plantilla **comuna01/templates/comuna01/departamento_form.html** agregaremos las siguientes líneas de código:

```
...
<div id="row">
    <div class="text-center" class="col-md-12">
        {% load static %}
        <img src="{% static "comuna01/departamento.jpg" %}"
alt="departamento"
            width="32px" height="32px">
                                    </div>
                                        </div>
...
```

Eso basta para incluir la imagen a la vez que podemos tener nuestros archivos estáticos como imágenes localizados en una carpeta determinada.



django.contrib.contenttypes

Esta aplicación puede rastrear todos los modelos instalados en su proyecto y proporciona una interfaz genérica para interactuar con sus modelos.

La aplicación **django.contrib.contenttypes** se incluye en la configuración INSTALLED_APPS por defecto cuando se crea un nuevo proyecto usando el comando **startproject**. Además, es utilizado por otros paquetes **contrib**.

La aplicación contenttypes contiene un modelo **ContentType.** Las instancias de este modelo representan los modelos de su aplicación, y las nuevas instancias de **ContentType** se crean automáticamente cuando se instalan nuevos modelos en su proyecto. El modelo **ContentType** tiene los siguientes campos:

- **app_label**: Indica el nombre de la aplicación a la que pertenece el modelo. Esto se toma automáticamente del atributo **app_label** de las opciones del modelo **Meta**. Por ejemplo, nuestro modelo **Departamento** pertenece a nuestra aplicación **comuna01**.

- **model**: el nombre de la clase modelo.

- **name**: Esto indica el nombre legible del modelo. Esto se toma automáticamente del atributo **verbose_name** de las opciones del modelo **Meta**.

Si revisamos nuestra base de datos veremos:

```
municipio01=> SELECT * FROM django_content_type;
 id| app_label     |    model
  +              +
1 | admin         | logentry
2 | auth          | permission
3 | auth          | group
4 | auth          | user
5 | contenttypes| contenttype
6 | sessions      | session
7 | comuna01      | departamento
8 | comuna01      | asignatura
9 | comuna01      | profesor
```

Como se aprecia, están nuestros modelos más aquellos de las aplicaciones instaladas por django cuando hacemos **startproject**.

Echemos un vistazo a cómo podemos interactuar con los objetos **ContentType**. Abramos el shell con el comando **python manage.py shell**. Puede obtener el objeto **ContentType** correspondiente a un modelo específico realizando una consulta con los atributos **app_label** y **model**, de la siguiente manera:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> departamento_type =
ContentType.objects.get(app_label='comuna01',model='departamento')>>>
departamento_type
<ContentType: comuna01 | departamento>
```

También puede obtener la clase modelo de un objeto **ContentType** llamando a su método **model_class()**:

```
>>> departamento_type.model_class()
<class 'comuna01.models.Departamento'>
```

También es común obtener el objeto **ContentType** para una clase de modelo en particular, de la siguiente manera:

```
>>> from comuna01.models import Departamento
>>> ContentType.objects.get_for_model(Departamento)
<ContentType: comuna01 | departamento>
```

Estos son solo algunos ejemplos del uso de tipos de contenido. Django ofrece más formas de trabajar con ellos. Puede encontrar la documentación oficial sobre el framework tipos de contenido en https://docs.djangoproject.com/en/3.1/ref/contrib/contenttypes/

django.contrib.auth

Django viene con un framework de autenticación incorporado que puede manejar la autenticación de usuarios, sesiones, permisos y grupos de usuarios. El sistema de autenticación incluye vistas (views) para acciones comunes del usuario, como inicio de sesión (login), cierre de sesión (logout), cambio de contraseña y restablecimiento de contraseña. El framework de autenticación se encuentra en **django.contrib.auth** y lo utilizan otros paquetes **contrib** de Django. Consiste en la aplicación **django.contrib.auth** y las siguientes dos clases de middleware que se encuentran en la configuración de **MIDDLEWARE**:

- **AuthenticationMiddleware**: asocia a los usuarios con solicitudes mediante sesiones.

- **SessionMiddleware**: maneja la sesión actual entre solicitudes.

Como hemos explicado un middleware es una clase con métodos que se ejecutan globalmente durante la fase de solicitud o respuesta.

El framework de autenticación también incluye los siguientes modelos:

- **User**: un modelo de usuario con campos básicos; los campos principales de este modelo son **username**, **password**, **email**, **first_name**, **last_name** e **is_active**.

- **Group**: un modelo de grupo para categorizar usuarios.

- **Permission**: banderas (flags) para que los usuarios o grupos realicen determinadas acciones.

Este framework también incluye vistas y formularios de autenticación predeterminados.

django.contrib.admin

Una de las grandes ventajas que nos presenta Django, es su interfaz de administración automática que viene ya incorporada en este framework. Django es capaz de leer los metadatos de sus modelos para proporcionar una interfaz donde los mismos usuarios puedan administrar el contenido de su sitio y/o proyecto.

El administrador, cuenta con muchas opciones para realizar una propia personalización. Gracias a las tablas y campos que se crean en nuestra base de datos al momento de migrar nuestro proyecto, es posible ir generando nuestras propias vistas.

Agregando un modelo a las páginas de administración

Se debe tener en cuenta que cuando creamos Models en una aplicación, debemos registrarlos para que puedan estar visibles en la pagina de administración de Django.

```
#myapp/admin.py
from django.contrib import admin
from myproject.myapp.models import MyModel

admin.site.register(MyModel)
```

Esto debemos importarlo en el submódulo de admin en nuestro archivo admin.py (Si creamos nuestra app utilizando manage.py startapp, este archivo, se debe haber generado de manera automática en el módulo de nuestra app.

Estas son algunas opciones que se encuentran definidas en la subclase ModelAdmin

```
class MyCustomAdmin(admin.ModelAdmin):
    list_display = ('name','age','email')  # Campos para mostrar en una lista
    empty_value_display = '-empty-'        # Mostrar el valor, cuando se encuentre vacío
    list_filter = ('name', 'company')      # Habilitar el filtrado de resultados
    list_per_page = 25                     # Numero de items por pagina ( 25 son en este ejemplo )
    ordering = ['-pub_date', 'name']       # Orden de resultados por defecto

# Y luego para dejarlo regsitrado
admin.site.register(MyModel, MyCustomAdmin)
```

Eliminando un modelo en administrador

En Django podemos encontrar varios modelos registrados de forma predeterminada. Existirán algunas ocasiones las cuales necesitemos eliminar uno de estos Modelos de nuestra página de administración.

En nuestro archivo admin.py que se encuentra en nuestra aplicación, debemos llamar la librería `from django.contrib import admin` y utilizar el método `admin.site.unregister`

```
#myapp/admin.py
from django.contrib import admin
from django.contrib.auth.models import User

admin.site.unregister(User)
```

Referencias

[1] Django Documentation https://docs.djangoproject.com/en/3.1/

[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.

[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.

[4] Django CRUD Tutorial – Operations and Application Development https://data-flair.training/blogs/django-crud-example/