
AWAKELAB
Desarrollo Web Python con Django
Desarrollador de aplicaciones Full Stack
Python Trainee

7.2.- Contenido 2: Implementar control de acceso a los recursos de un aplicativo web utilizando el modelo de autorización y permisos disponible en Django

Objetivo de la jornada

1. Explica el concepto de Mixins en proyectos Django para control y seguridad de acceso
2. Implementa el control de acceso a recursos de vista y de acciones a un aplicativo web utilizando el modelo de autorización y permisos de Django

7.2.1.- Uso de Mixins

7.2.1.1. Qué son los Mixin

Para hablar de Mixines necesitamos tocar el tópico de herencia múltiple. En el módulo 3, vimos en qué consiste la herencia en el marco de programación orientada a objetos.

La herencia múltiple es un tema delicado. En principio, es muy simple: una subclase que hereda de más de una clase principal puede acceder a la funcionalidad de ambas. En la práctica, esto es mucho menos útil de lo que parece y muchos programadores expertos recomiendan no usarlo. Entonces comenzaremos con una advertencia:

"Como regla general, si cree que necesita una herencia múltiple, probablemente esté equivocado, pero si sabe que la necesita, probablemente tenga razón."

La forma más simple y útil de herencia múltiple se llama "mixin". Un mixin es generalmente una superclase que no está destinada a existir por sí misma, sino que está destinada a ser heredada por alguna otra clase para proporcionar una funcionalidad adicional. Por ejemplo, digamos que queremos agregar una funcionalidad a una clase hipotética llamada **Contacto** que permite enviar un correo electrónico a **self.email**.

```
class Contacto:
        lista_contactos = []

    def __init__(self, nombre, email):
```

```
        self.nombre = nombre
                self.email = email
                Contacto.lista_contactos.append(self)
```

Enviar correo electrónico es una tarea común que podríamos querer utilizar en muchas otras clases. Entonces podemos escribir una clase mixin simple para enviarnos el correo electrónico:

```
class EnviarMail:
        def send_mail(self, mensaje):
                print("Enviando email a " + self.email)
        # Logica para mandar emails
        ...
```

Por simplicidad, no incluiremos aquí la lógica real del correo electrónico; Si está interesado en estudiar cómo se hace, consulte el módulo **smtplib** en la biblioteca estándar de Python.

Esta clase no hace nada especial (de hecho, apenas puede funcionar como una clase independiente), pero nos permite definir una nueva clase que es tanto un **Contacto** como un **EnviarMail**, usando herencia múltiple:

```
class EnviarEmailAContacto(Contacto, EnviarMail):
    pass
```

La sintaxis de la herencia múltiple parece una lista de parámetros en la definición de clase. En lugar de incluir una clase base dentro del paréntesis, incluimos dos (o más), separadas por una coma.

```
>>> e = EnviarEmailAContacto("Caupolicán Catrileo", "caupolican@catrileo.com")
>>> Contacto.lista_contactos
[<__main__.EnviarEmailAContacto at 0x109aba520>]
>>> e.send_mail("Enviando mensaje de prueba.")
Enviando email a caupolican@catrileo.com
```

El inicializador de **Contacto** aún agrega el nuevo contacto a la lista all_contacts, y el mixin puede enviar correo a **self.email** así que sabemos que todo está funcionando.

Eso no fue tan difícil, y probablemente se esté preguntando cuáles son las advertencias sobre la herencia múltiple. Entraremos en las complejidades en un minuto, pero consideremos qué opciones teníamos, además de usar un mixin aquí:

- Podríamos haber usado herencia única y agregar la función **send_mail** a la subclase. La desventaja aquí es que la funcionalidad del correo electrónico debe duplicarse para cualquier otra clase que necesite correo electrónico.

- Podemos crear una función Python independiente para enviar correo y simplemente llamarla, con la dirección de correo electrónico correcta proporcionada como parámetro, cuando sea necesario enviar un correo electrónico.

- Podríamos usar un monkey-patch para la clase **Contacto** y tener un método **send_mail** después de que se haya creado la clase. Lo cual es una pésima práctica:

```
e = Contacto("Caupolicán Catrileo", "caupolican@catrileo.com")
def send_mail(email):
    print("Mandando un correo a " + email)
e.send_mail = send_mail
e.send_mail(e.email)
```

La herencia múltiple funciona bien cuando se mezclan métodos de diferentes clases, pero se vuelve complicado cuando tenemos que trabajar llamando métodos en la superclase. ¿Por qué? Porque hay múltiples superclases. ¿Cómo sabemos a cuál llamar? ¿Cómo sabemos en qué orden llamarlos?

Para ilustrar más el punto vamos a crear una nueva clase, supongamos que necesitamos extender nuestra clase **Contacto** para que nos sea útil en la diferenciación con quienes son a la vez nuestros amigos:

```
class Amigo(Contacto):
    def __init__(self, nombre, email, telefono):
        super().__init__(nombre, email)
        self.telefono = telefono
```

Exploremos las preguntas anteriores agregando una dirección a nuestra clase de Amigo. ¿De qué maneras podríamos hacer esto? Una dirección es una colección de cadenas que representan la calle, la ciudad, el país y otros detalles relacionados del contacto. Podríamos pasar cada una de estas cadenas como parámetros al método **__init__** de la clase **Amigo**. También podríamos almacenar estas cadenas en una tupla o diccionario y pasarlas a **__init__** como un solo argumento. Este es probablemente el mejor curso de acción si no hay ninguna funcionalidad adicional que deba agregarse a la dirección.

Otra opción sería crear una nueva clase **Direccion** para mantener juntas esas cadenas y luego pasar una instancia de esta clase a __init__ en nuestra clase **Amigo**. La ventaja de esta solución es que podemos agregar comportamiento (digamos, un método para dar instrucciones a esa dirección o para imprimir un mapa) a los datos en lugar de simplemente almacenarlos estáticamente. Esto sería utilizar composición, la relación "tiene un(a)". La composición es una solución perfectamente viable a este problema y nos permite reutilizar las clases **Direccion** en otras entidades como edificios, negocios u organizaciones.

Sin embargo, la herencia también es una solución viable, y eso es lo que queremos explorar, así que agreguemos una nueva clase que contenga una dirección. Llamaremos a esta nueva clase **TitularDeDireccion** en lugar de **Direccion**, porque la herencia define una relación "es un(a)". No es correcto decir que un amigo es una dirección, pero dado que un amigo puede tener una dirección, podemos argumentar que un amigo es un titular de una dirección. Posteriormente, podríamos crear otras entidades (empresas, edificios) que también tengan direcciones. Aquí está nuestra clase **TitularDeDireccion**:

```
class TitularDeDireccion:
    def __init__(self, calle, ciudad, comuna, codigopostal):
        self.calle = calle
        self.ciudad = ciudad
        self.comuna = comuna
        self.codigopostal = codigopostal
```

Muy simple; simplemente tomamos todos los datos y los arrojamos a variables de instancia al inicializar.

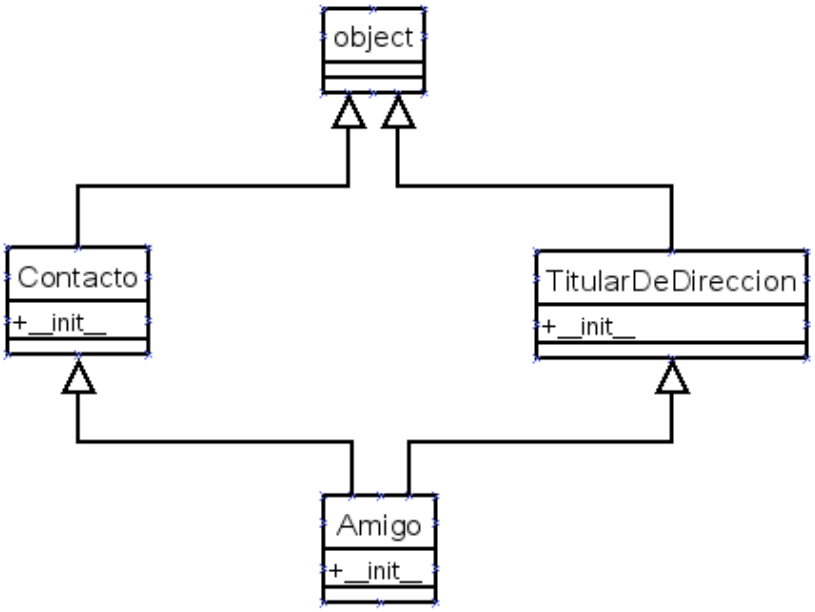
7.2.1.2. El problema del diamante

Pero ¿cómo podemos usar esto en nuestra clase **Amigo** existente, que ya hereda de **Contacto**? Herencia múltiple, por supuesto. La parte complicada es que ahora tenemos dos métodos principales **__init__** que deben inicializarse. Y deben inicializarse con diferentes argumentos. ¿Como hacemos eso? Bueno, podríamos comenzar con el enfoque ingenuo:

```
class Amigo(Contacto, TitularDeDireccion):
    def __init__(self, name, email, telefono, calle, ciudad, comuna, codigopostal):
        Contacto.__init__(self, name, email)
        TitularDeDireccion.__init__(self, calle, ciudad, comuna, codigopostal)
        self.telefono = telefono
```

En este ejemplo, llamamos directamente a la función **__init__** en cada una de las superclases y pasamos explícitamente el argumento self. Este ejemplo funciona técnicamente; podemos acceder a las diferentes variables directamente en la clase. Pero hay algunos problemas.

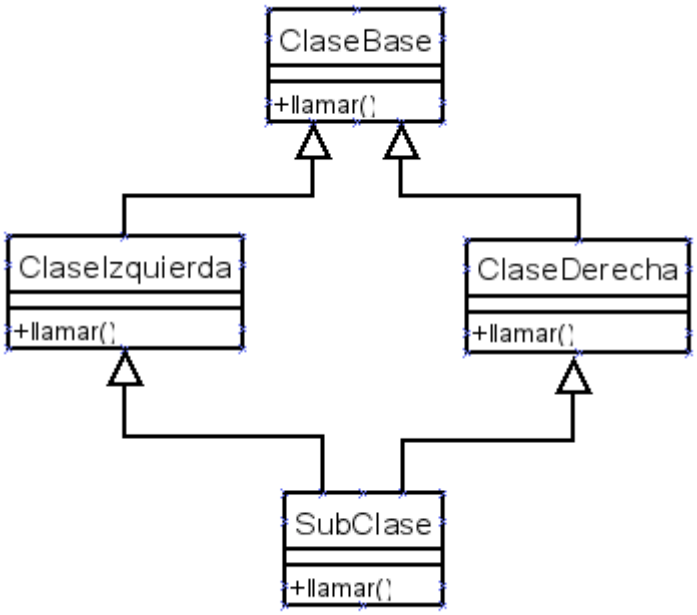
Primero, es posible que una superclase no se inicialice si descuidamos llamar explícitamente al inicializador. Esto no es malo en este ejemplo, pero podría causar fallas en el programa en escenarios comunes. Imagine, por ejemplo, intentar insertar datos en una base de datos a la que no se ha conectado.
En segundo lugar, y más siniestro, está la posibilidad de que una superclase sea llamada varias veces, debido a la organización de la jerarquía de clases. Mire este diagrama de herencia:



El método **__init__** de la clase **Amigo** llama primero a **__init__** en **Contacto** que inicializa implícitamente la superclase **object** (recuerde, todas las clases derivan de **object**). **Amigo** luego llama a **__init__** de **TitularDeDireccion**, que implícitamente inicializa la superclase **object** ...una vez más. La clase principal se ha configurado dos veces. En este caso, tal hecho es relativamente inofensivo, pero en algunas situaciones, podría significar un desastre. ¡Imagínese intentar conectarse a una base de datos dos veces por cada solicitud! La clase base solo debe llamarse una vez. Una vez, sí, pero ¿cuándo? ¿Llamamos a **Amigo**, luego a Contacto, luego Object y luego **TitularDeDireccion**? O Amigo, luego **Contacto**, luego **TitularDeDireccion** y luego **Object**?

Nota: Técnicamente, el orden en el que se pueden llamar los métodos se puede adaptar sobre la marcha modificando el atributo **__mro__** (Orden de resolución de métodos) en la clase. Pero eso está fuera del alcance de este curso. Si le interesa ver más del tema este puede ser un buen punto de partida https://en.wikipedia.org/wiki/C3_linearization

Veamos un segundo ejemplo artificial que ilustra este problema con mayor claridad. Aquí tenemos una clase base que tiene un método llamado call_me. Dos subclases anulan ese método, y luego otra subclase amplía ambos usando herencia múltiple. Esto se llama herencia de diamantes debido a la forma de diamante del diagrama de clases:



Los diamantes son lo que dificulta la herencia múltiple. Técnicamente, toda la herencia múltiple en Python 3 es herencia de diamante, porque todas las clases heredan de **object.** El diagrama anterior, que usa `object.__ init__`, también es un diamante.

Al convertir este diagrama a código, este ejemplo muestra cuándo se llaman los métodos:

```
class ClaseBase:
    numero_de_llamadas= 0

    def llamar(self):
        print("Llamando método en Clase Base")
        self.numero_de_llamadas += 1
```

```
class ClaseIzquierda(ClaseBase):
    numero_de_llamadas_izquierda = 0

    def llamar(self):
        ClaseBase.llamar(self)
        print("Llamando método en Clase Izquierda")
        self.numero_de_llamadas_izquierda += 1


class ClaseDerecha(ClaseBase):
    numero_de_llamadas_derecha = 0

    def llamar(self):
        ClaseBase.llamar(self)
        print("Llamando método en Clase Derecha")
self.numero_de_llamadas_derecha += 1

class SubClase(ClaseIzquierda, ClaseDerecha):
    numero_de_subllamadas = 0

    def llamar(self):
    ClaseIzquierda.llamar(self)
    ClaseDerecha.llamar(self)
    print("Llamando método en Sub Clase")
    self.numero_de_subllamadas += 1
```

Este ejemplo simplemente garantiza que cada método sobrescrito **llamar** invoque directamente al método padre con el mismo nombre. Cada vez que se llama, nos avisa imprimiendo la información en la pantalla y actualiza una variable estática en la clase para mostrar cuántas veces se ha llamado. Si instanciamos un objeto Subclase y llamamos al método una vez, obtenemos este resultado:

```
>>> s = SubClase()
>>> s.llamar()
Llamando método en Clase Base
Llamando método en Clase Izquierda
Llamando método en Clase Base
Llamando método en Clase Derecha
Llamando método en Sub Clase
>>> print(s.numero_de_subllamadas, s.numero_de_llamadas_izquierda,
s.numero_de_llamadas_derecha, s.numero_de_llamadas)
1 1 1 2
```

El método **llamar** de la clase base se ha llamado dos veces. Este no es un comportamiento esperado y puede dar lugar a errores (bugs) muy difíciles si ese método realmente funciona, por ejemplo, depositar dos veces en una cuenta bancaria.

Lo que hay que tener en cuenta con la herencia múltiple es que solo queremos llamar al método "siguiente" en la jerarquía de clases, no al método "principal". De hecho, el siguiente método puede no estar en un padre o antepasado de la clase actual. La palabra clave **super** viene a nuestro rescate una vez más. De hecho, **super** se desarrolló originalmente para hacer posibles formas complicadas de herencia múltiple. Aquí está el mismo código escrito usando super:

```
class ClaseBase:
    numero_de_llamadas = 0

    def llamar(self):
        print("Llamando método en Clase Base")
        self.numero_de_llamadas += 1

class ClaseIzquierda(ClaseBase):
    numero_de_llamadas_izquierda = 0

    def llamar(self):
        super().llamar()
        print("Llamando método en Clase Izquierda")
        self.numero_de_llamadas_izquierda += 1

class ClaseDerecha(ClaseBase):
    numero_de_llamadas_derecha = 0

    def llamar(self):
        super().llamar()
        print("Llamando método en Clase Derecha")
        self.numero_de_llamadas_derecha += 1

class SubClase(ClaseIzquierda, ClaseDerecha):
    numero_de_subllamadas = 0

    def llamar(self):
        super().llamar()
        print("Llamando método en Sub Clase")
        self.numero_de_subllamadas += 1
```

El cambio es bastante menor; simplemente reemplazamos las llamadas directas ingenuas con llamadas a **super()**. Esto es bastante simple, pero mire la diferencia cuando lo ejecutamos:

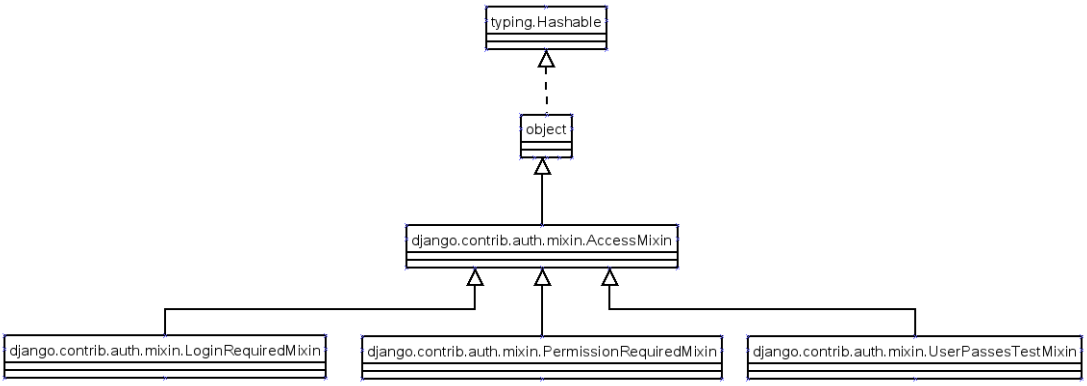
```
>>> s = SubClase()
>>> s.llamar()
Llamando método en Clase Base
Llamando método en Clase Derecha
Llamando método en Clase Izquierda
Llamando método en Sub Clase
>>> print(s.numero_de_subllamadas, numero_de_llamadas_izquierda,
s.numero_de_llamadas_derecha, s.numero_de_llamadas)
1 1 1 1
```

Se ve bien, nuestro método base solo se llama una vez. Pero ¿qué está haciendo **super()** realmente aquí? Dado que las instrucciones **print** se ejecutan después de las llamadas a **super**, la salida impresa está en el orden en que se ejecuta cada método. Veamos la salida de atrás hacia adelante para ver quién llama a qué.

El primer **llamar** de **SubClase** llama a **super().llamar()**, que se refiere a **ClaseIzquierda.llamar()**. **ClaseIzquierda.llamar()** luego llama a **super().llamar()**, pero en este caso, **super()** se refiere a **ClaseDerecha.llamar()**. Preste especial atención a esto; la super llamada no está llamando al método en la superclase de **ClaseIzquierda** (que es **ClaseBase**), está llamando a **ClaseDerecha**, ¡aunque no es un padre de **ClaseIzquierda**! Este es el método "siguiente", no el método "principal". **ClaseDerecha** luego llama a **ClaseBase** y las `super` llamadas han asegurado que cada método en la jerarquía de clases se ejecute una vez.

7.2.1.3 Para qué nos sirven los Mixin en el modelo Auth

Como acabamos de ver los mixins nos permiten utilizar herencia múltiple, y tales mixines no están destinados a existir por cuenta propia, sino que son heredados por otras clases con el fin de contar con funcionales adicionales. En el contexto del sistema de autenticación de Django, tenemos acceso a funcionalidades proporcionadas por mixines.



Como se ve en el diagrama existen funcionalidades provistas por una super clase y mixines que, aún siendo ellos mismos mixines, dependen de esta clase para implementar características adicionales.

7.2.1.4. Aplicando LoginRequiredMixin

Así, como la función **login_required** puede ser aplicada en distintas partes de nuestro código con el fin de permitir que ciertas páginas sean vistas sólo por usuarios que cuenten con acceso, cuando usamos vistas basadas en clases podemos utilizar el mixin **LoginRequired**.

login required puede ser utilizado en directamente en nuestras URLs:

```
...
urlpatterns = [
    ...
    path('departamento', login_required(DepartamentoCreate.as_view()),
         name="url_departamento_create"),
        ...
```

También se puede utilizar vistas de clase:

```
class DepartamentoCreate(CreateView):
        ...
    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
```

Análogamente podemos hacer:

```
class DepartamentoCreate(LoginRequiredMixin, CreateView):
    template_name = 'comuna01/departamento_form.html'
    form_class = DepartamentoCustomForm
    ...
```

Y nuestra vista permitirá el acceso solo a usuarios registrados.

7.2.1.5. Aplicando PermissionRequiredMixin

Hemos visto las situaciones para **login_required** existe una contraparte similar que es el mixin **PermissionRequiredMixin**, el cual más allá de solo hacer un chequeo de login también examina los permisos. Ambas funciones en código no son muy distintas de usar, en el caso de vistas basadas en funciones podemos usar el decorador **@permission_required**. Y como hemos visto, ya que las vistas basadas en clase son más flexibles y en ellas nos hemos enfocado, lo podemos usar de la siguiente forma:

Nota: el nombre del permiso es irrelevante para nuestro ejemplo, usted puede usar lo que estime conveniente en un caso real de acuerdo con sus necesidades.
Primero creamos el permiso en el modelo, usando **Meta**

```
...
class Departamento(models.Model):
        ...
    Class Meta:
        permissions = (
            ("permiso_departamento", "Un permiso necesario en departamento"),
        )
```

Como hicimos cambios en el modelo necesitamos ejecutar migraciones:

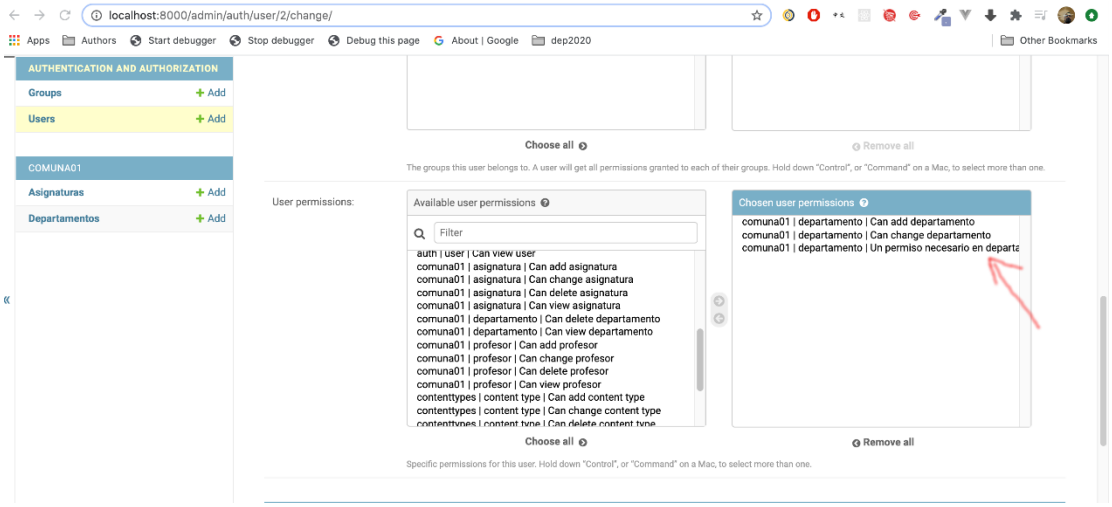
```
$ python manage.py makemigrations
Migrations for 'comuna01':
  comuna01/migrations/0005_auto_20200926_0927.py
    - Change Meta options on departamento

$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, comuna01, contenttypes,
sessionsRunning migrations:
  Applying comuna01.0005_auto_20200926_0927... OK
```

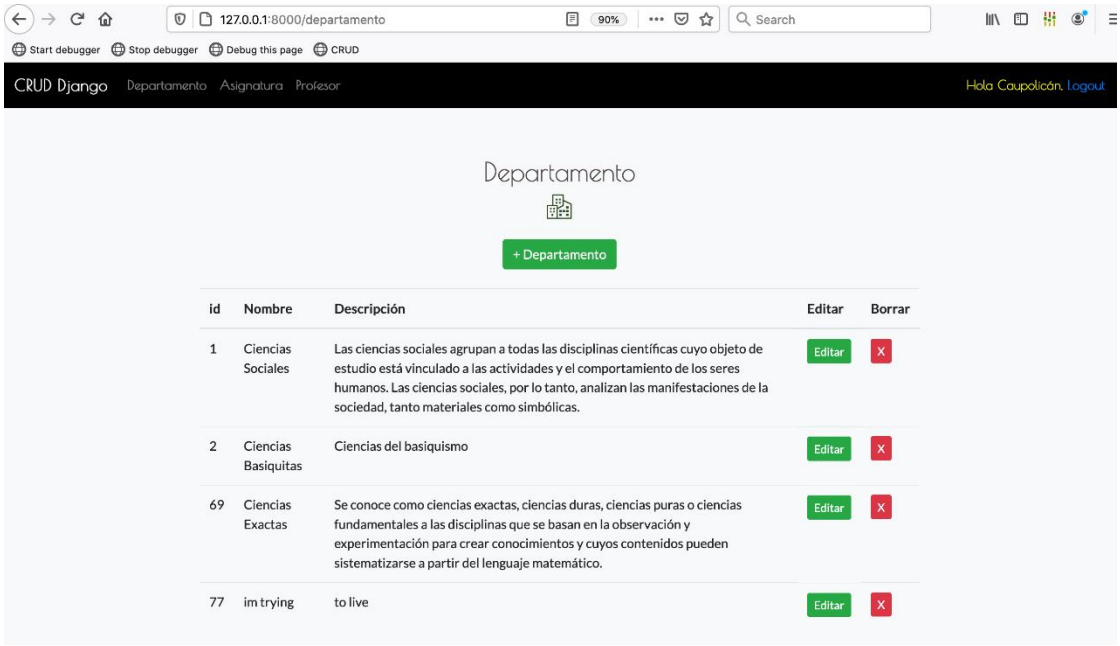
Ahora en **views.py**, usaremos nuestra vista para crear “departamento” pero podría ser cualquier vista definida por nosotros, lo haremos por simplicidad:

```
from django.contrib.auth.mixins import LoginRequiredMixin, PermissionRequiredMixin
...
class DepartamentoCreate(PermissionRequiredMixin, LoginRequiredMixin, CreateView):
        ...
    permission_required= 'comuna01.permiso_departamento'
        ...
```

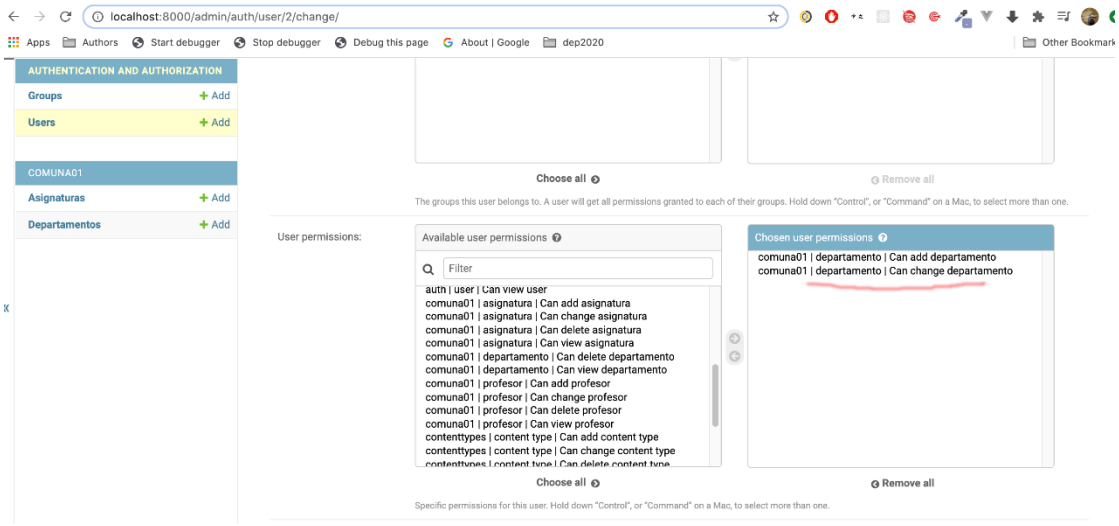
Ahora que ya lo tenemos configurado lo podemos asignar en la interfaz del sitio administrativo> Nosotros lo asignaremos a nuestro usuario de staff Caupolicán:



Y nuestro usuario es capaz de crear y "acceder" a Departamentos:



Hasta aquí nada nuevo, pero ¿qué sucede si le sacamos tal permiso a Caupolicán en el sitio administrativo?



En tal caso cuando nuestro usuario trate de ingresar se encontrará con:

Es decir, le ha sido negado el acceso por no contar con los permisos suficientes.

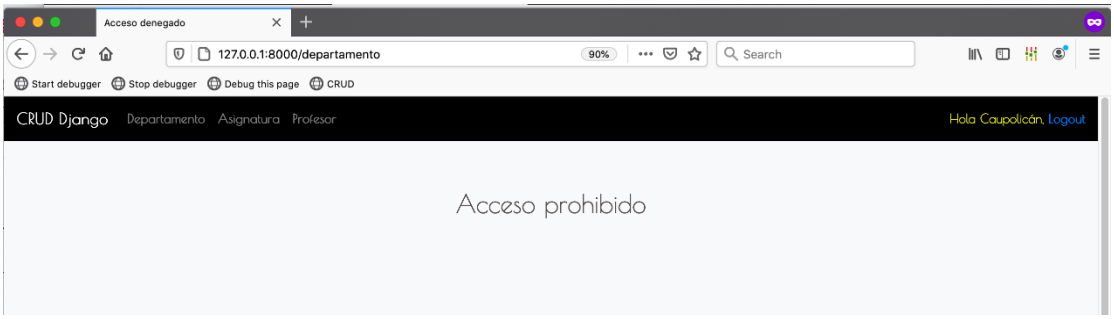
Si queremos embellecer esa respuesta podemos utilizar las facilidades de Django para el despliegue de códigos 404, 500, etc.

Simplemente creamos una plantilla llamada `403.html` en el directorio templates y podemos mostrar algo más ameno:

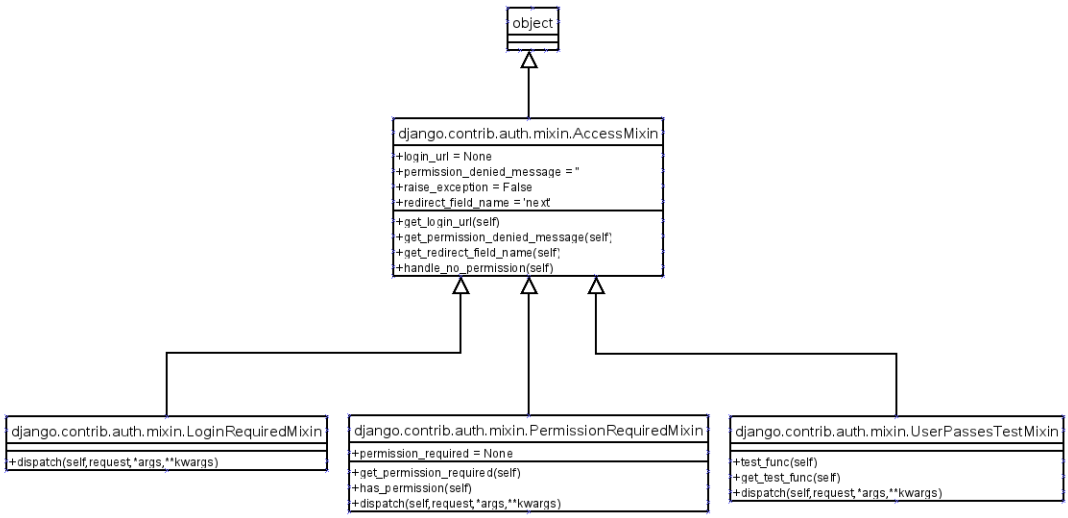
```
{% extends 'comuna01/base.html' %}

{% block title %}Acceso denegado{% endblock %}

{% block top_title %}
    <h2 class="text-center">Acceso prohibido</h2>
{% endblock %}
```



Si quiere explorar más a fondo este tópico este diagrama más completo es un buen punto de partida:



7.2.2. 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.