DJANGO

Para chequear errores en la aplicación, entramos en la carpeta del proyecto y digitamos

Manage.py check

MIGRACIONES:

Si construimos tablas dentro de la aplicación, debemos migrarlas, para eso usamos el comando:

Primero debemos añadir nuestra aplicación a settings.py:

INSTALLED\_APPS = [

'products.apps.ProductsConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Luego manage.py makemigrations

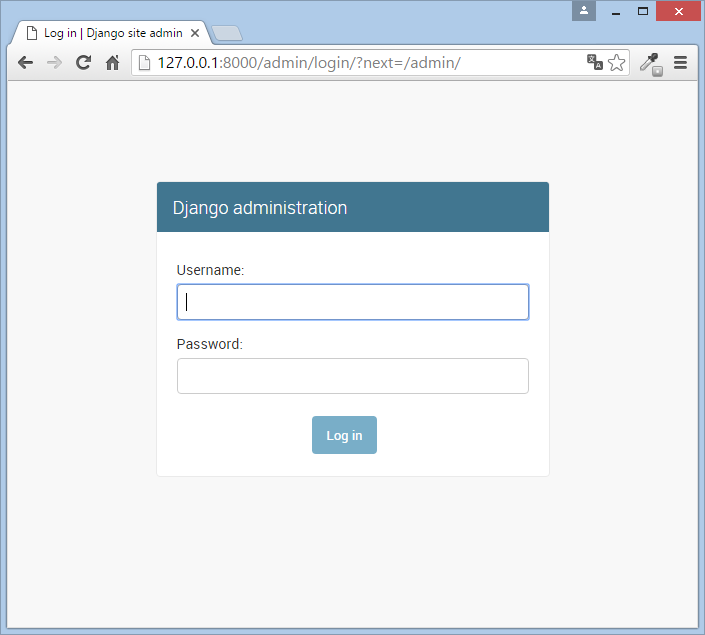
Y por ultimo

Manage.py migrate

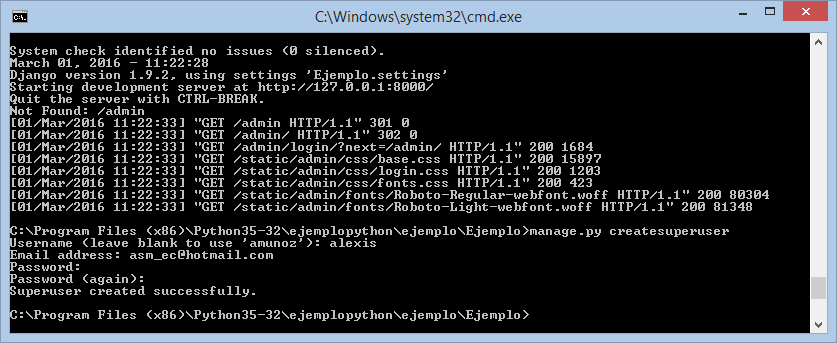
Corremos el servidor:

Manage.py runserver

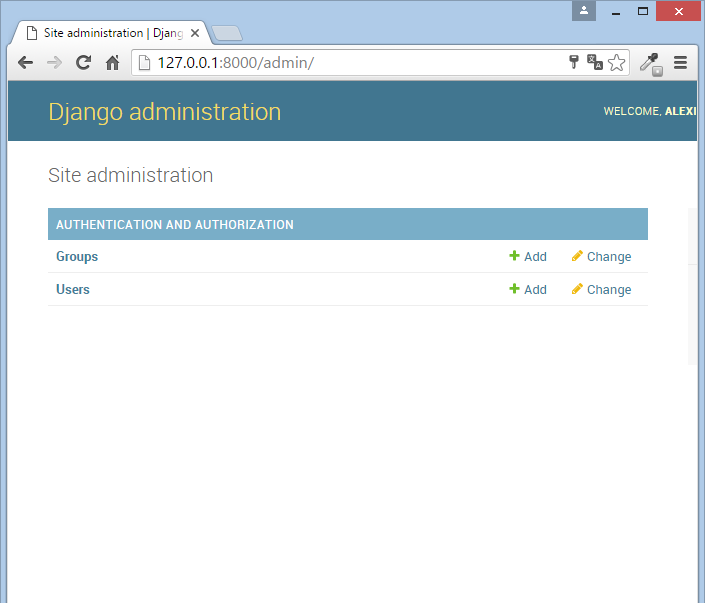
Vamos al sitio y digitamos:



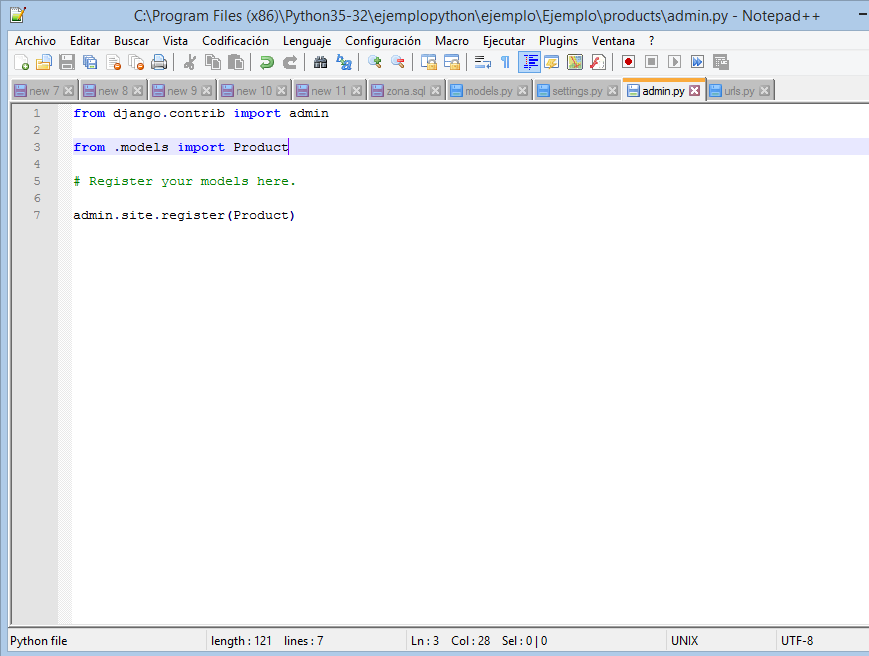
Necesitamos crear un usuario, para esto digitamos: manage.py createsuperuser



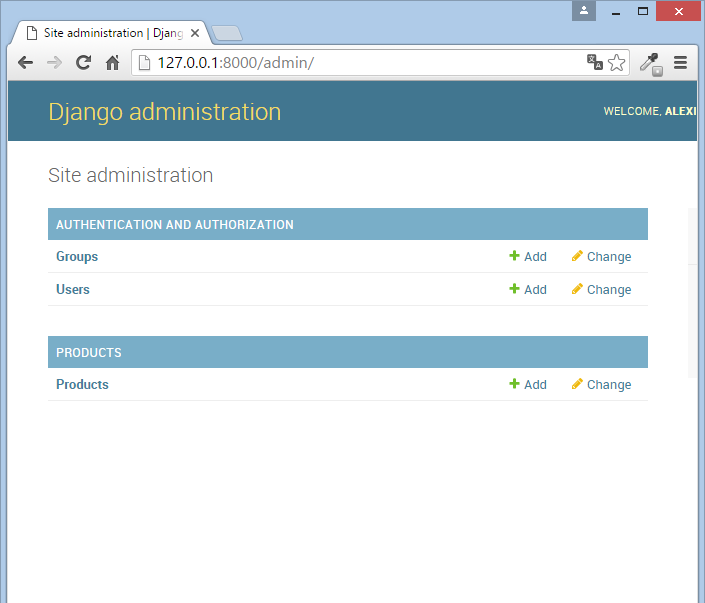
Nuevamente nos logueamos y tenemos la siguiente pantalla:



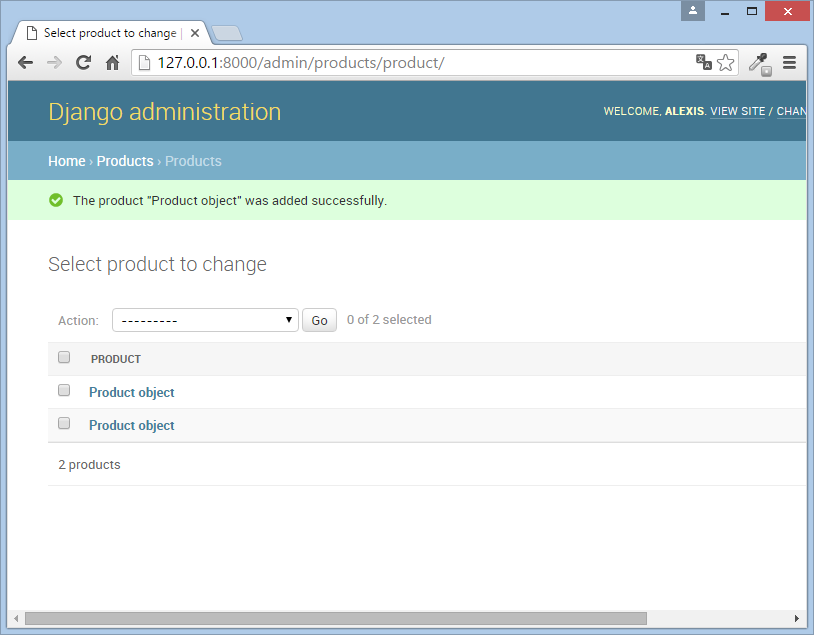
Para visualizar nuestro modelo hacemos una modificación en admin:



Y una vez que recargamos, tenemos:

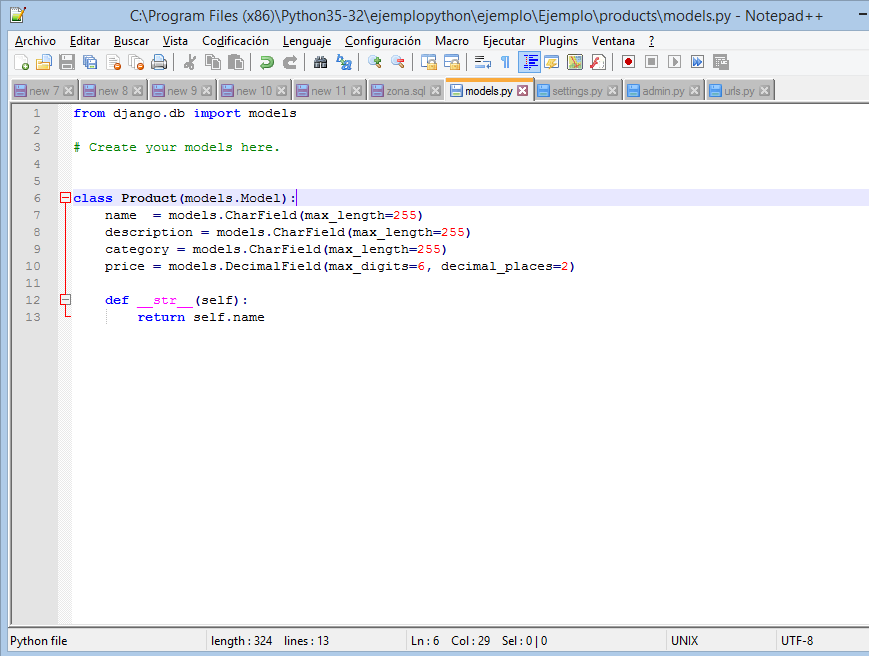


Al momento de crear un nuevo producto nos aparece Product Objec en vez del nombre del producto:

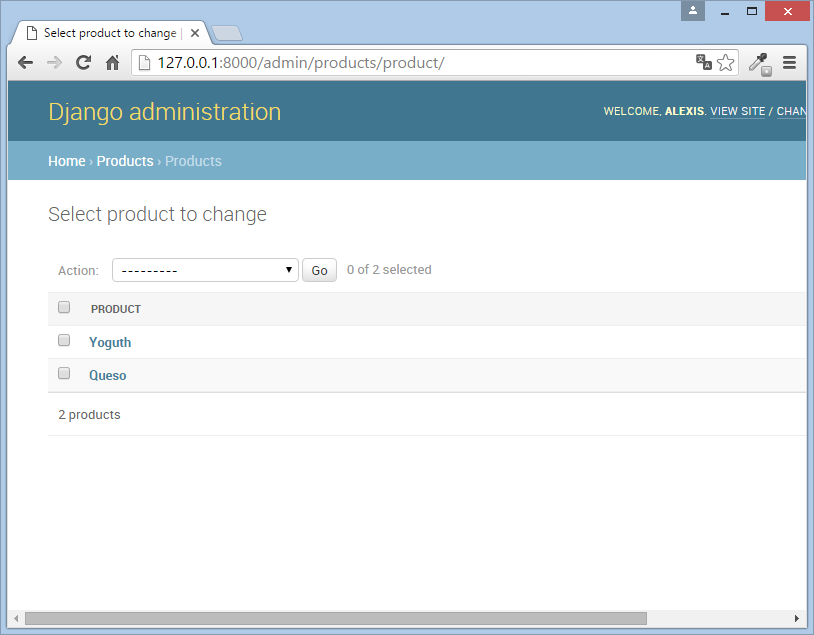


Para esto vamos a modificar de la siguiente forma:

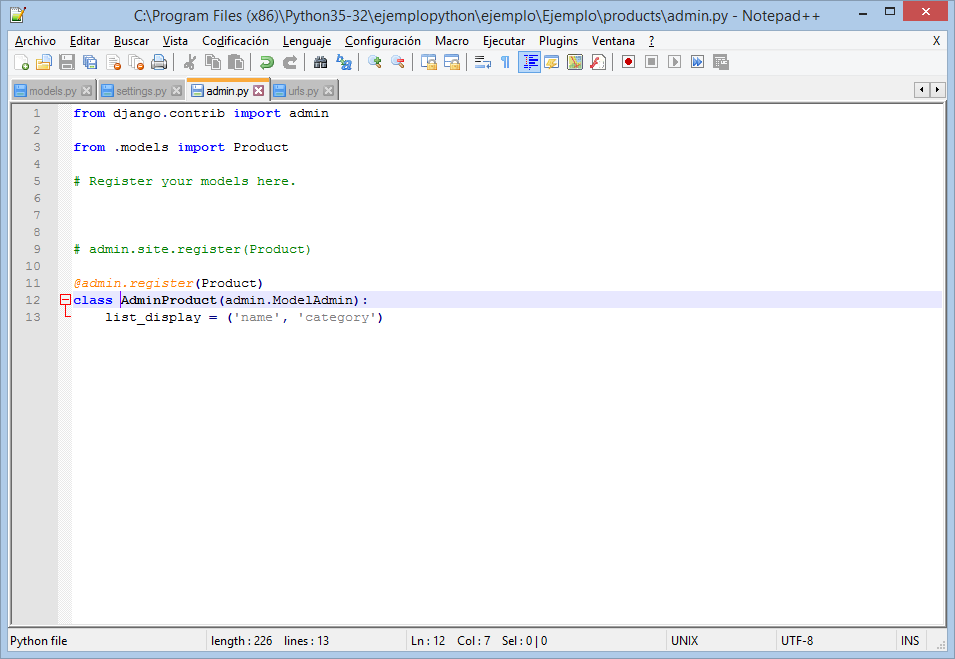
En Models:



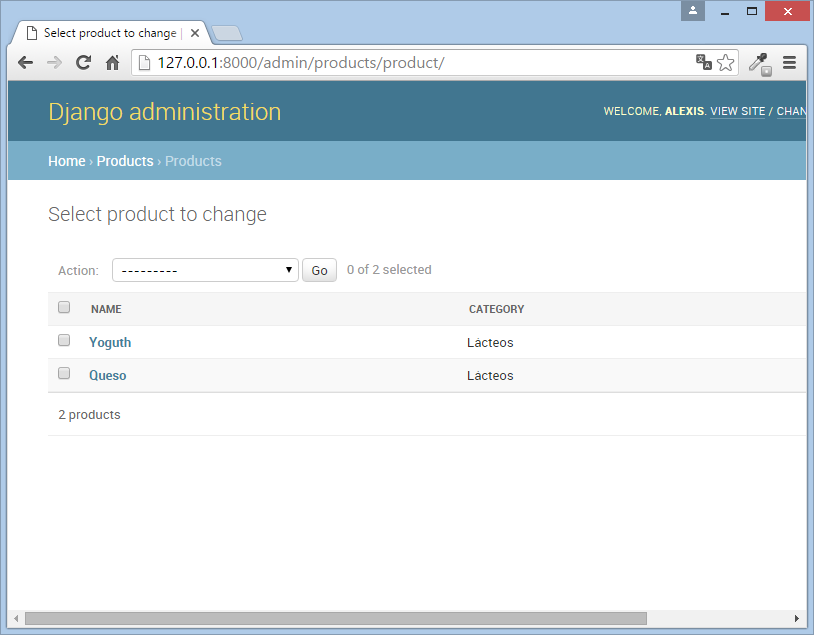
Y verificacmos:



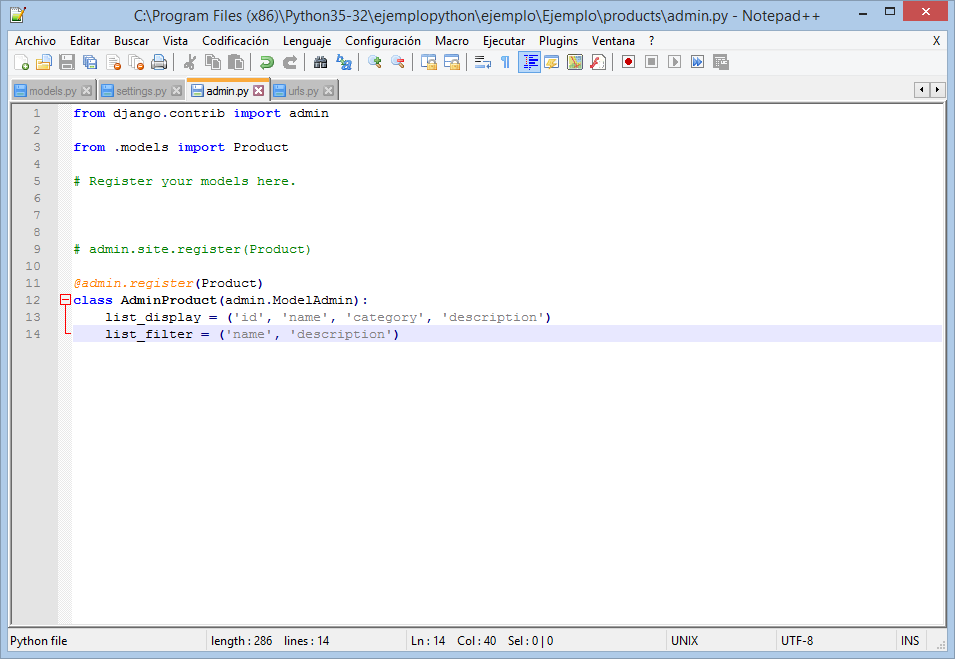
Si queremos que despliegue nombre y categoría, modificacmos el admin de la siguiente forma:



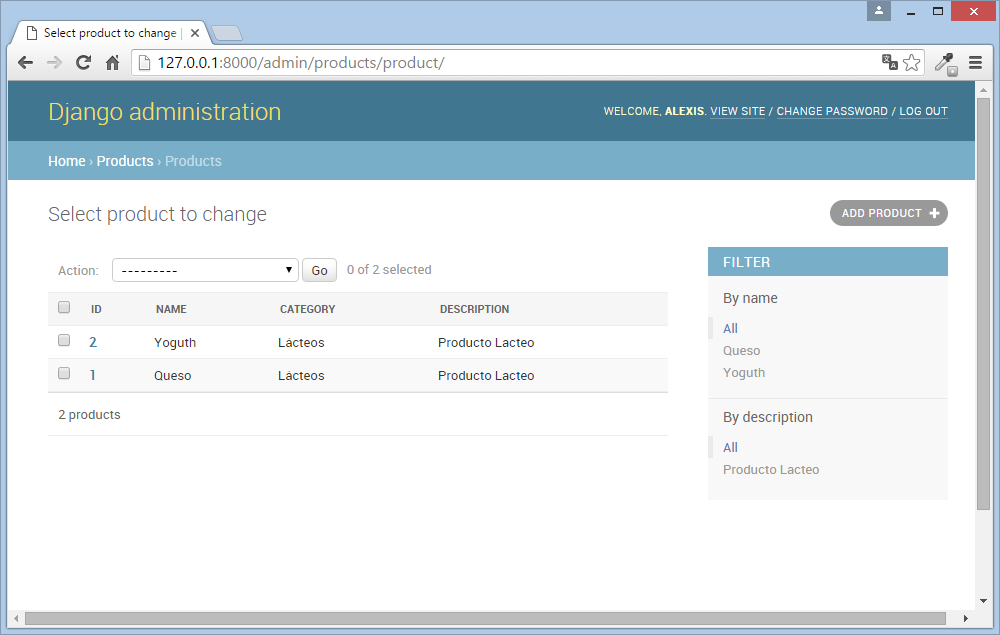
Y obtenemos:



Podemos incluir un filtro:

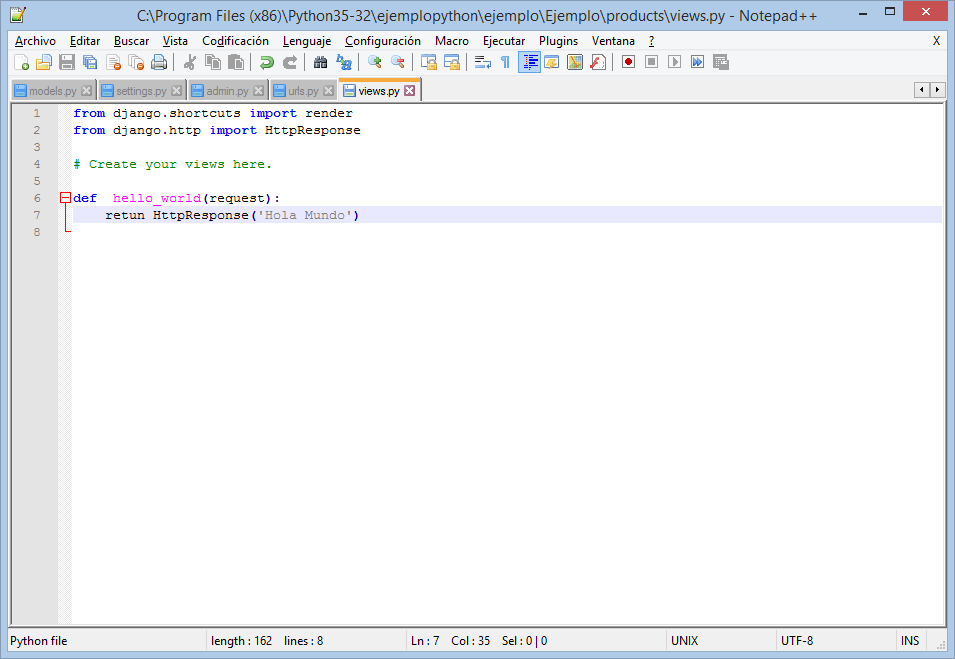


Cuyo resultado es:

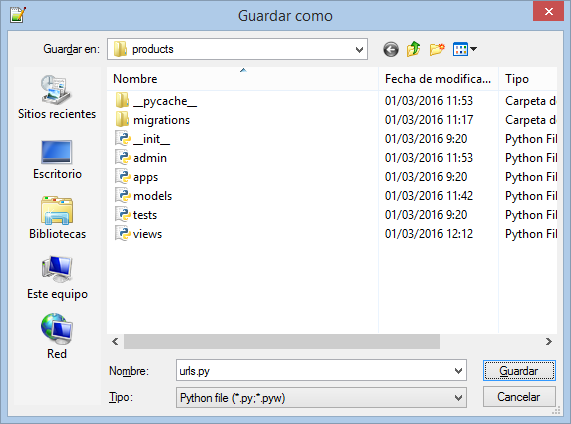


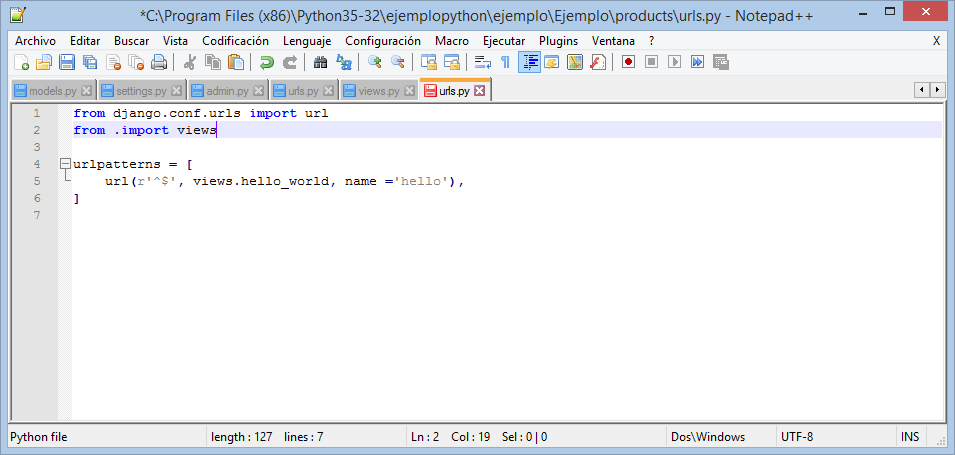
CREACION DE VISTAS:

Creamos en VIEWS:

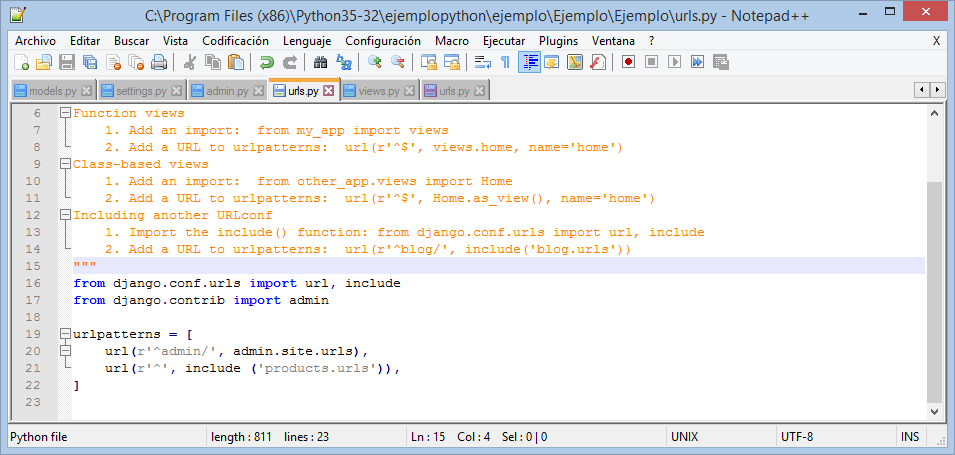


Sin embargo no pasa nada. Para esto vamos a crear un nuevo archivo dentro de Products llamao urls.py:



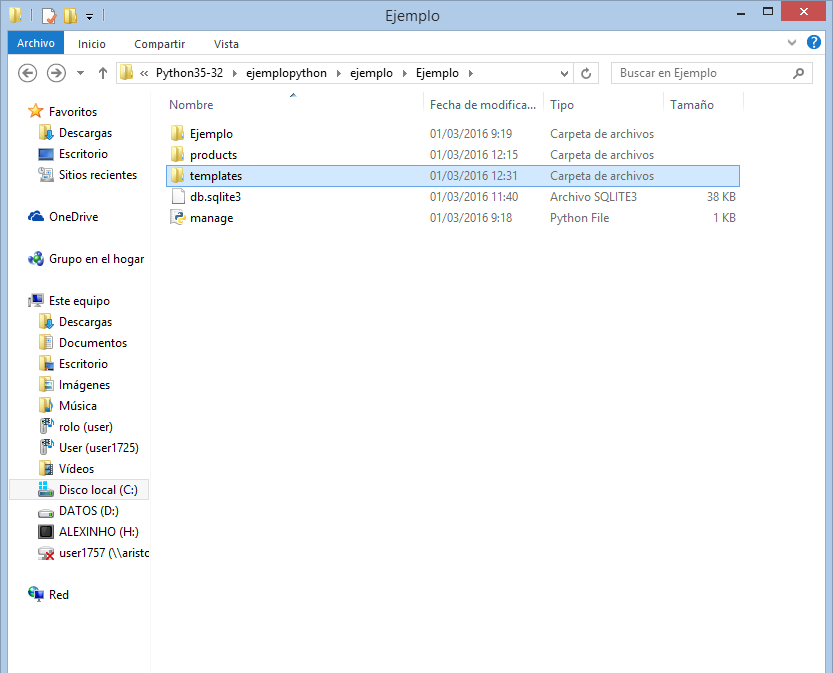


Volvemos al archivo original url y modificamos:



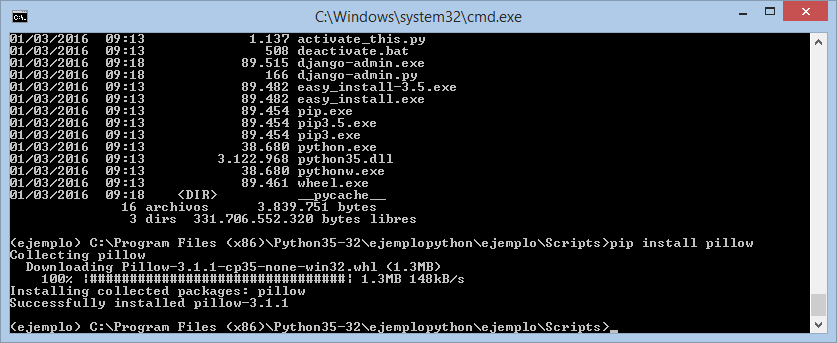
USO DE TEMPLATES:

Creamos una carpeta templates y dentro un archivo index.html

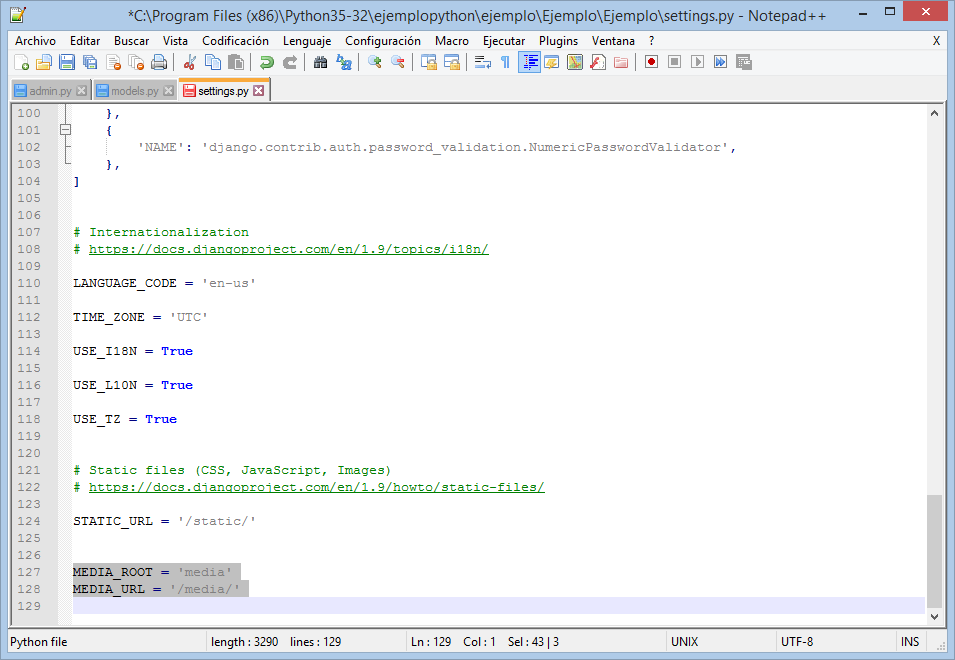


TRATAMIENTO DE IMAGNES:

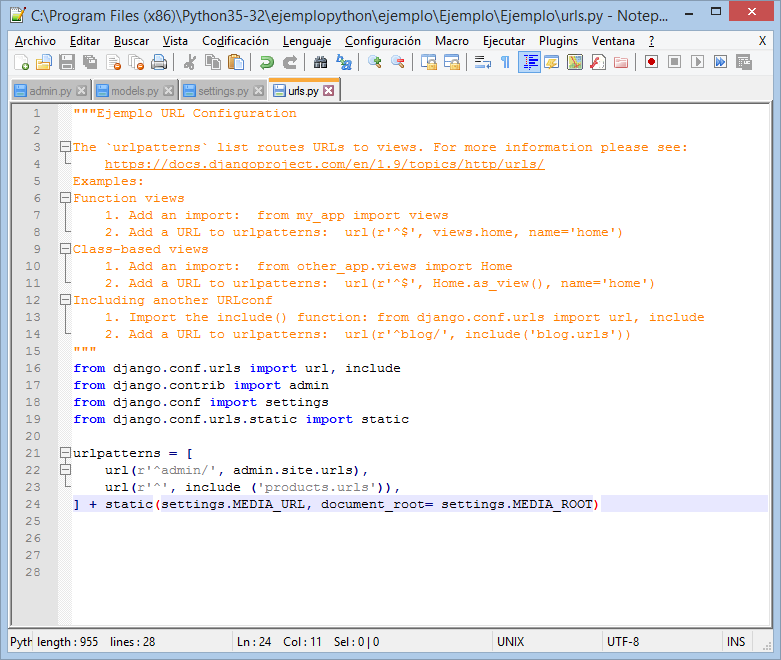
Instalamos la librería pillow



Añadir los campos, necesarios y activar lo siguiente en settings:

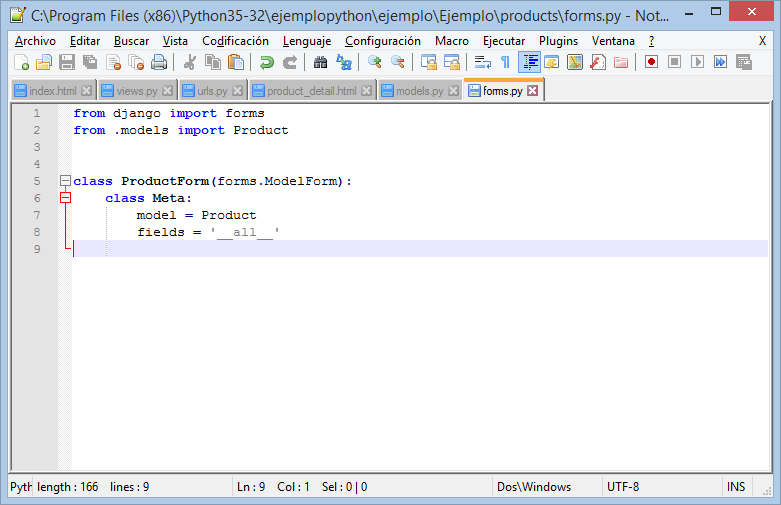


Configuramos en url del proyecto:

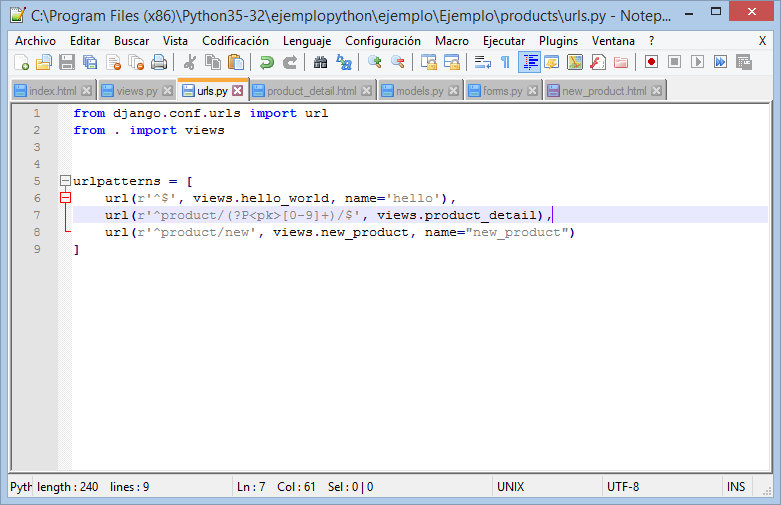


FORMULARIOS:

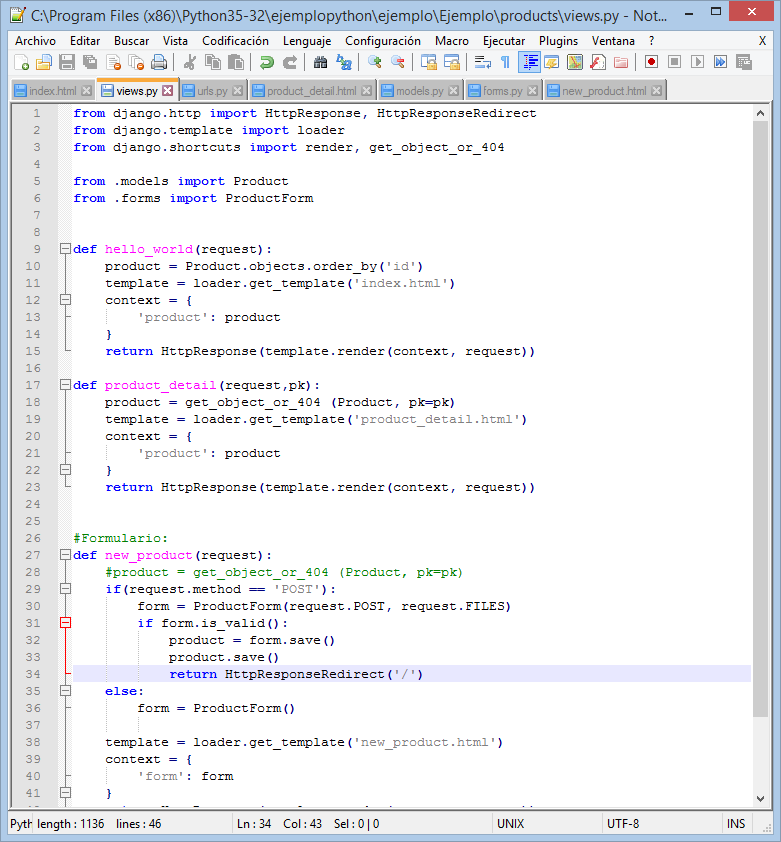
Creamos un forms.py y digitamos:

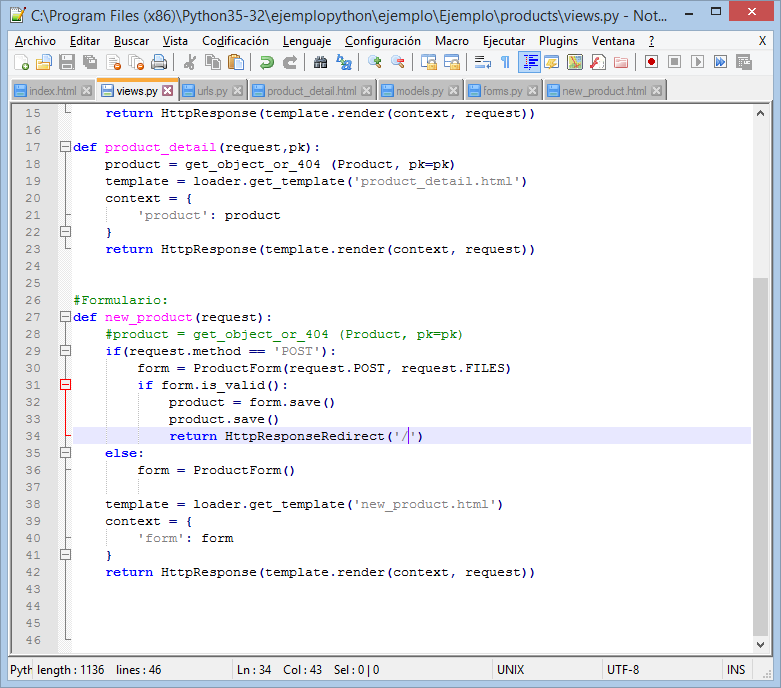


Url:

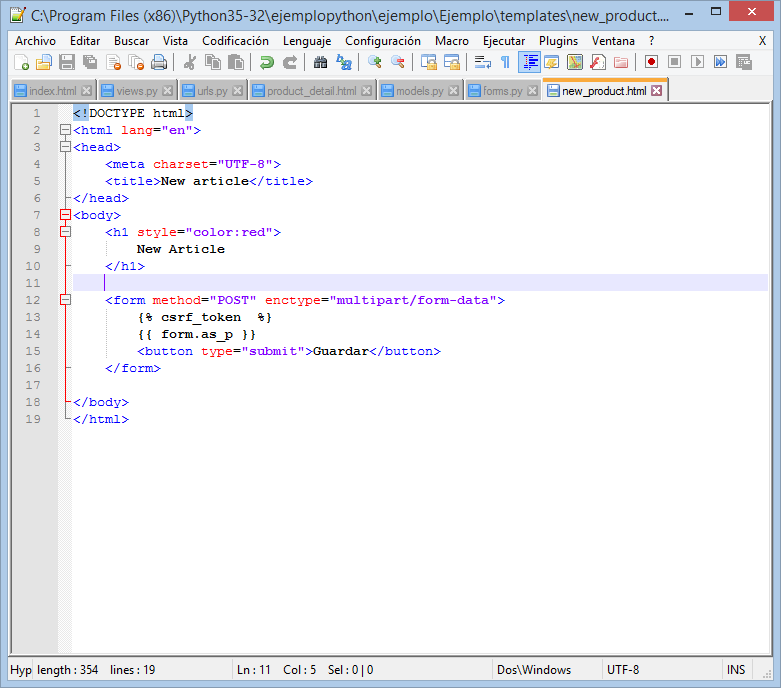


View:





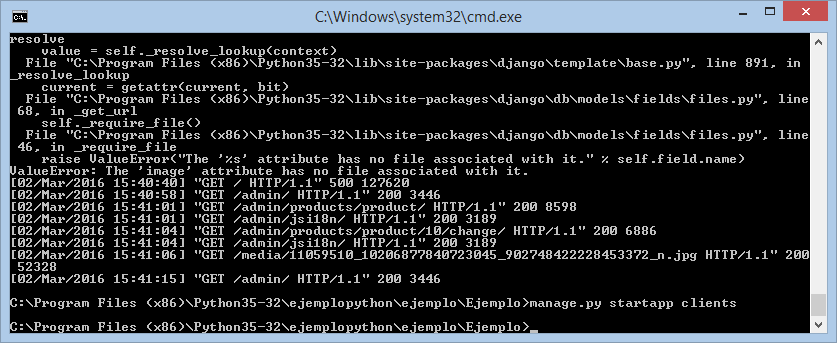
New\_product.html:



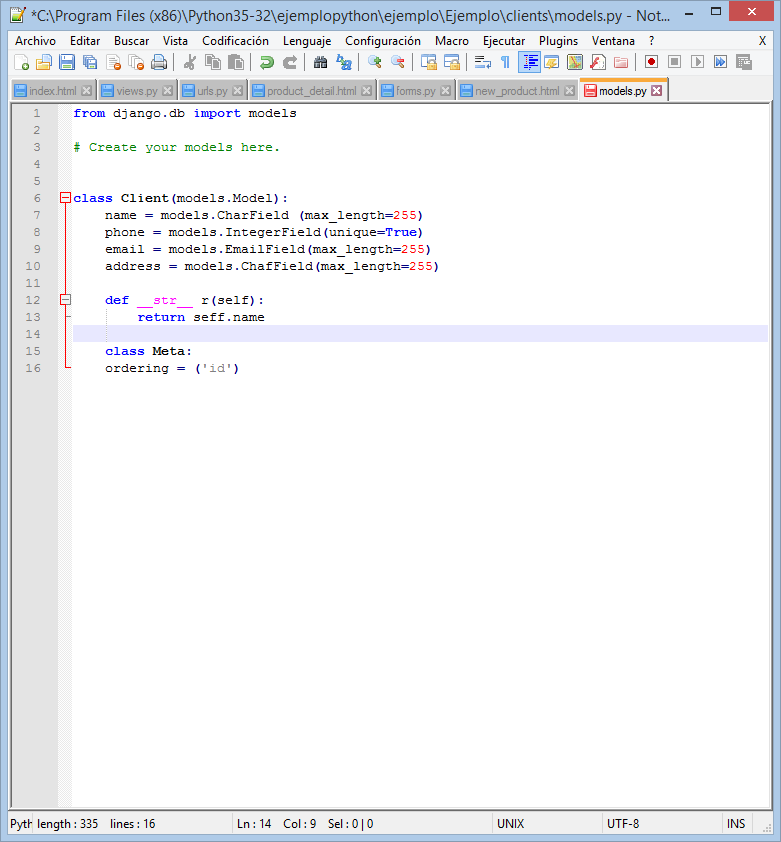
RELACIONES ENTRE MODELOS:

Creamos un nuevo modelo:

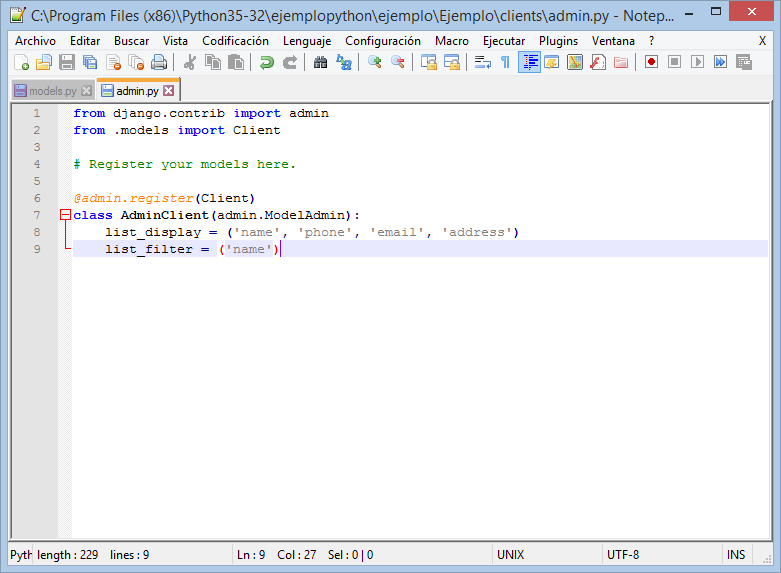
Iniciamos nuevo modelo:



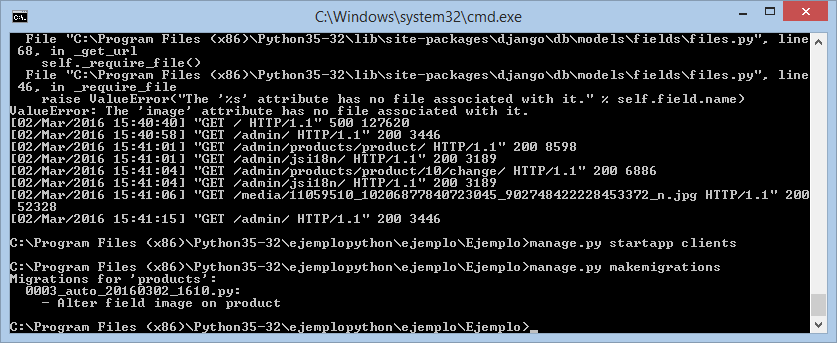
Modifico modelo de Clients:



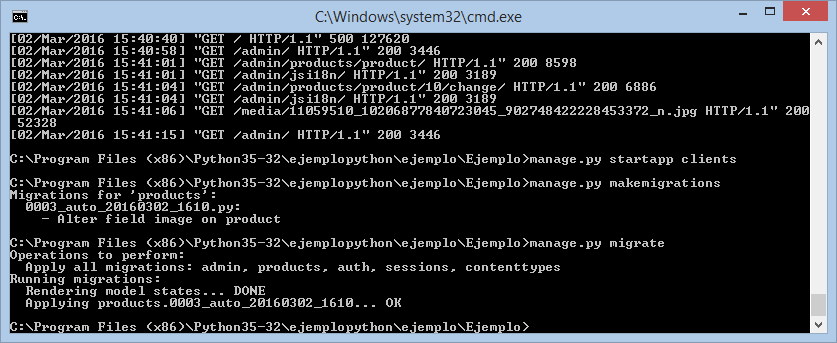
Modifico admin:



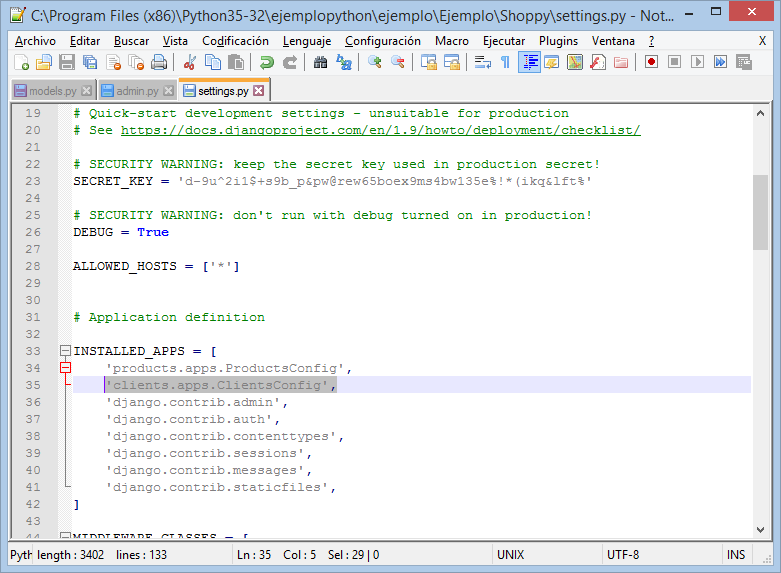
Hacemos la migración:



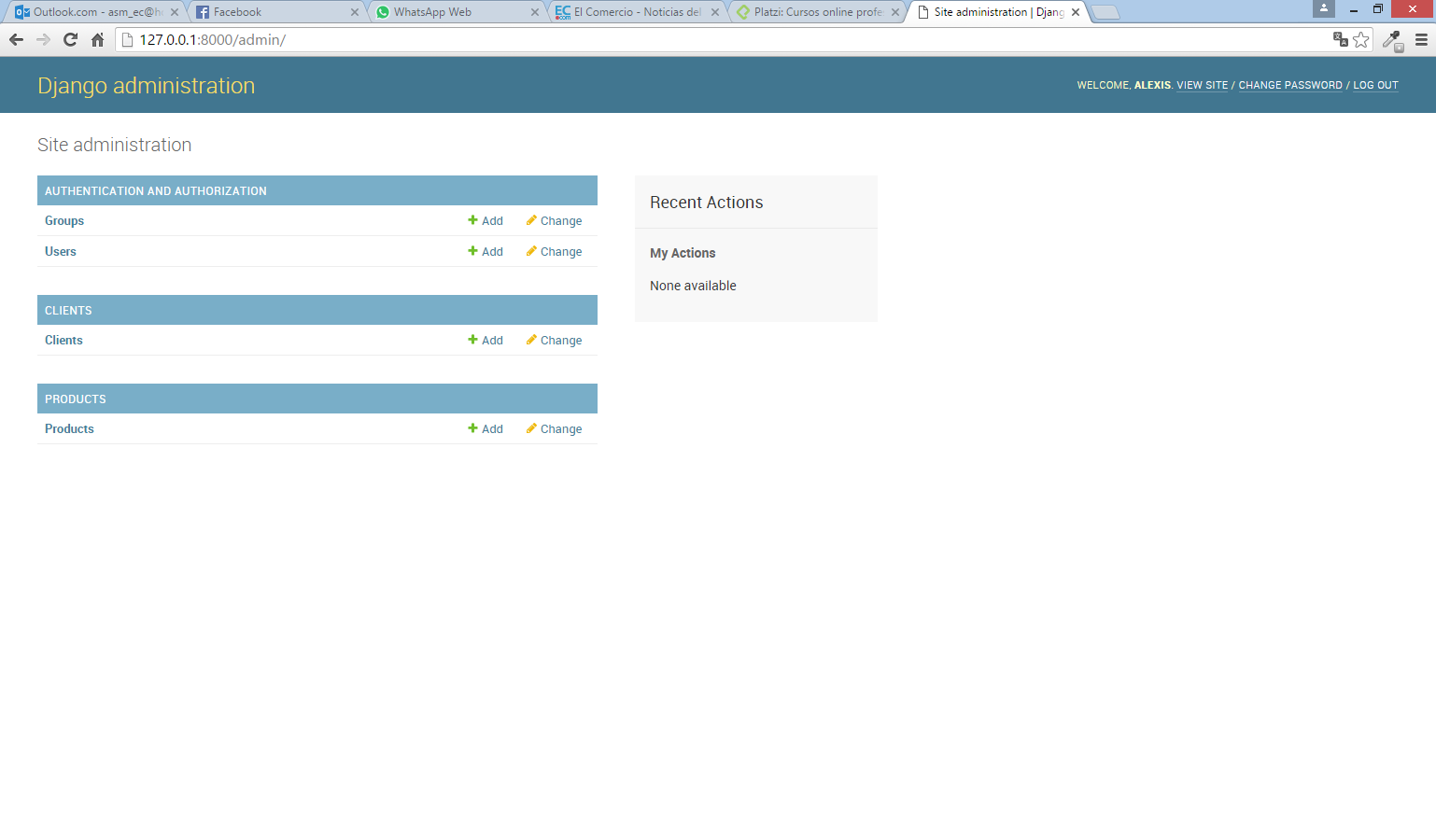
Luego ejecutamos MIgrate:



Registramos en settings.py de nuestra aplicación.

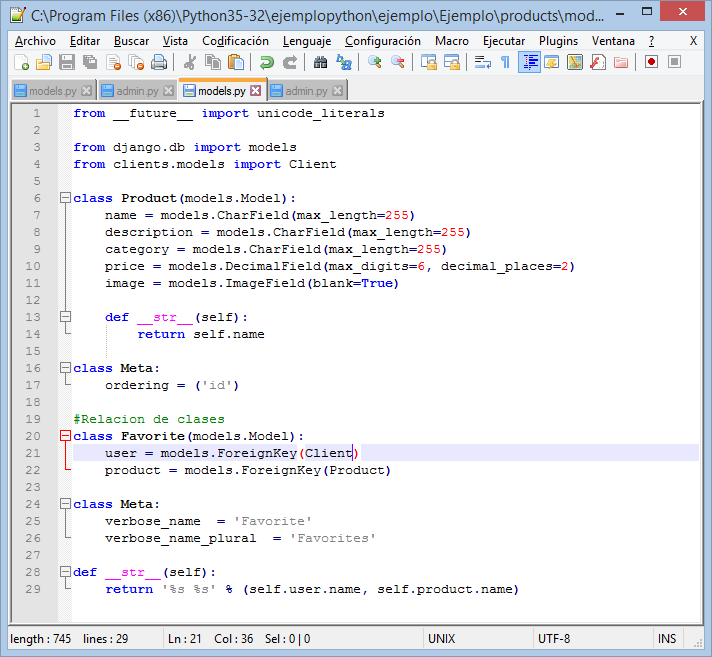


Verifico en el navegador:

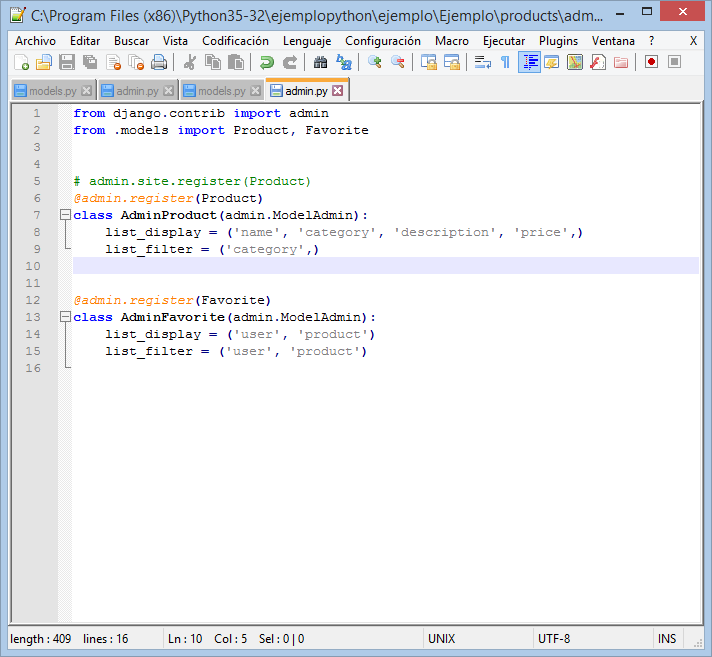


Para crear una relación, me voy a admin de Product y digito:

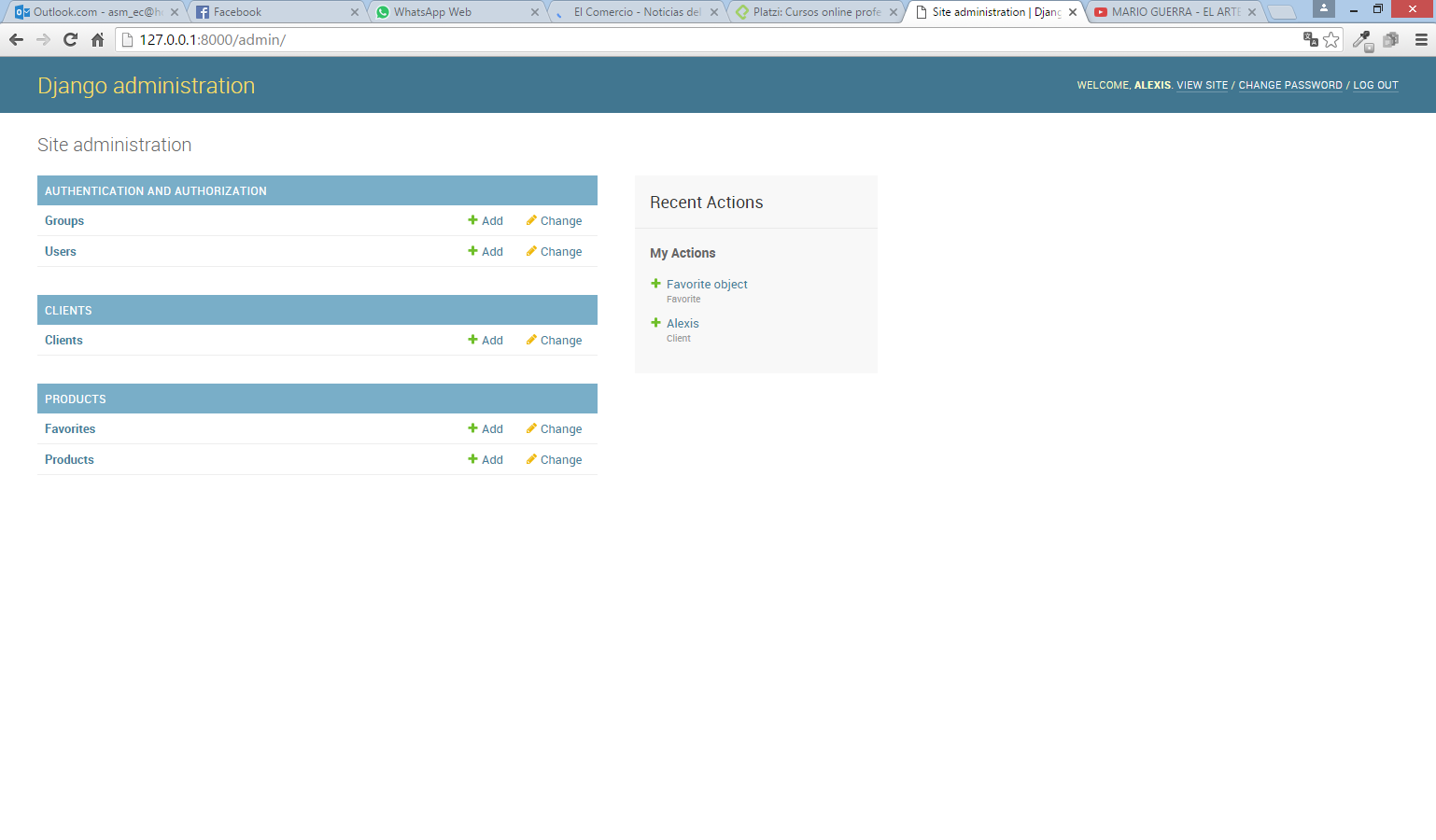
MOdels:

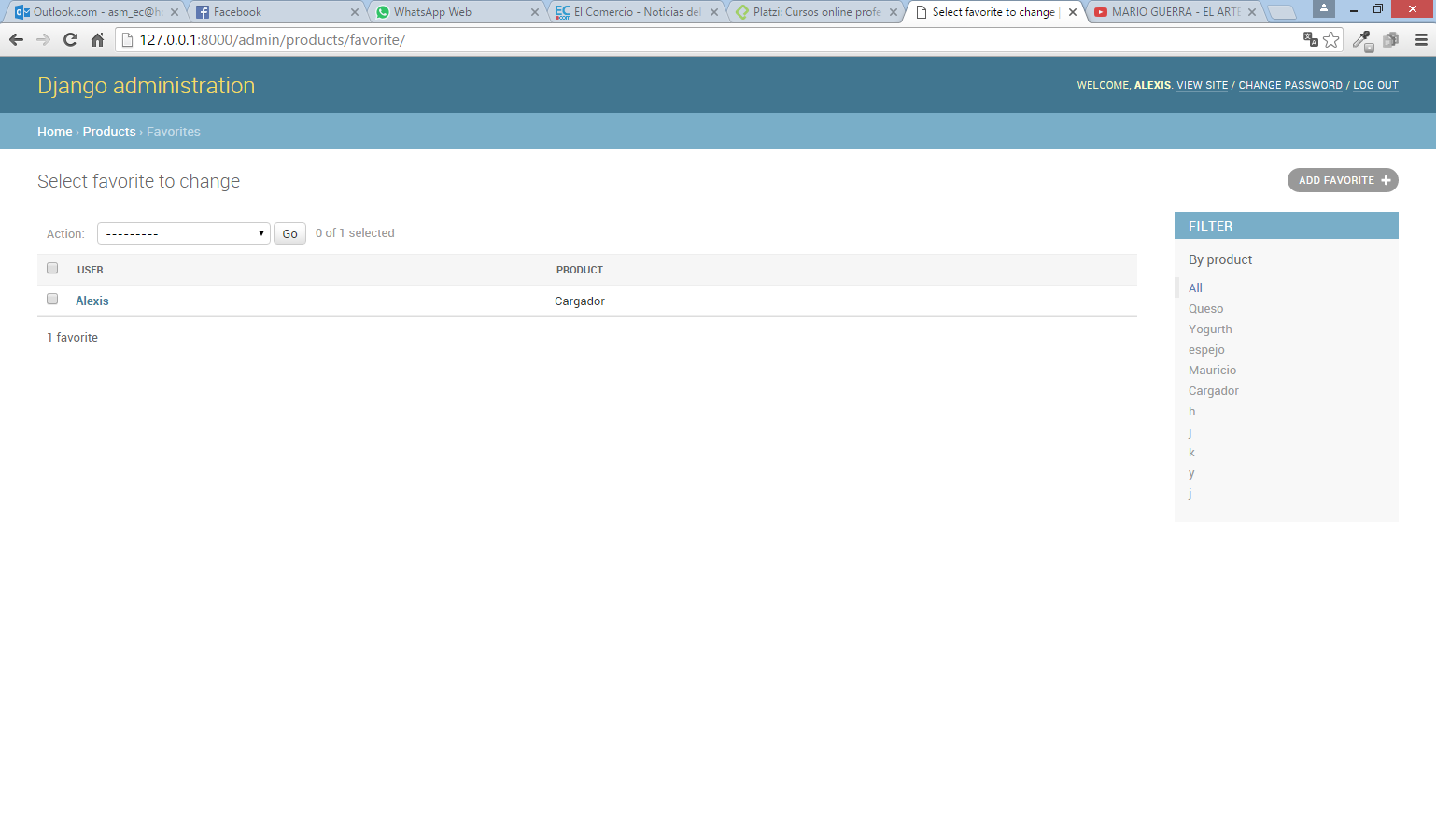


Admin:



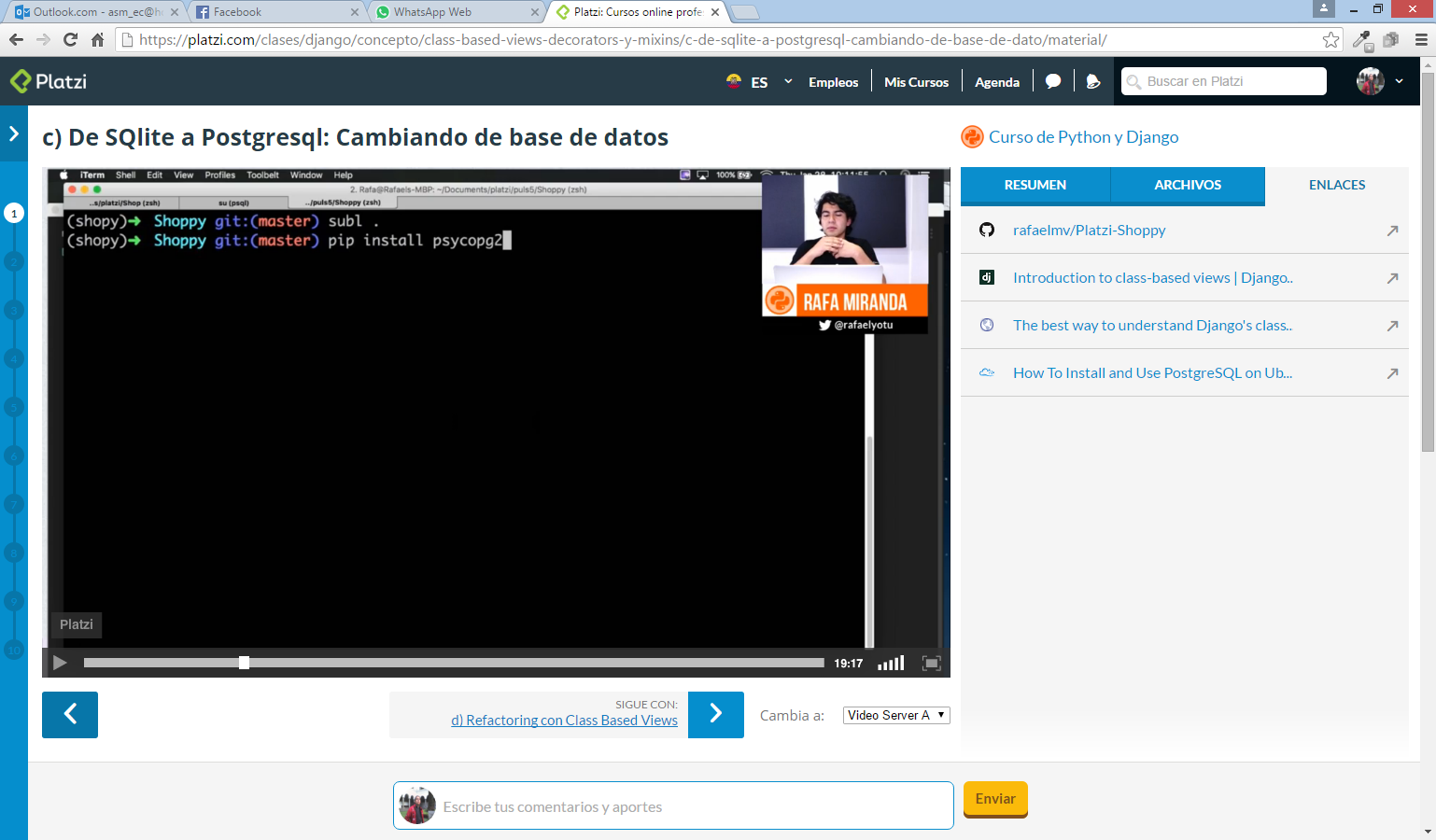
Tenemos lo siguiente:





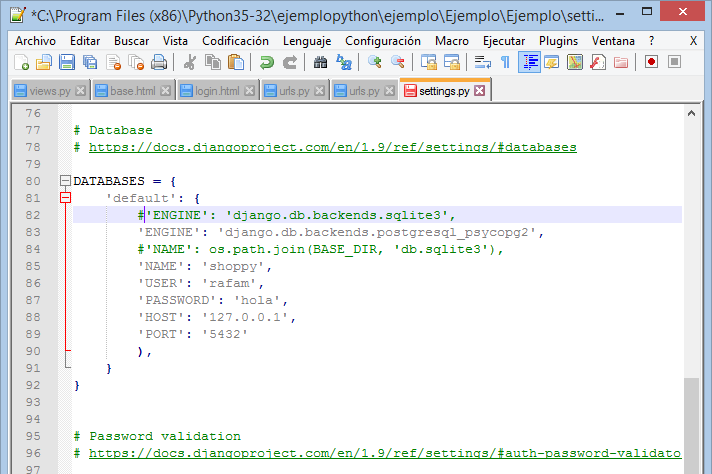
CAMBIO DE SQLITE A POSTGRES:

Instalar motor:

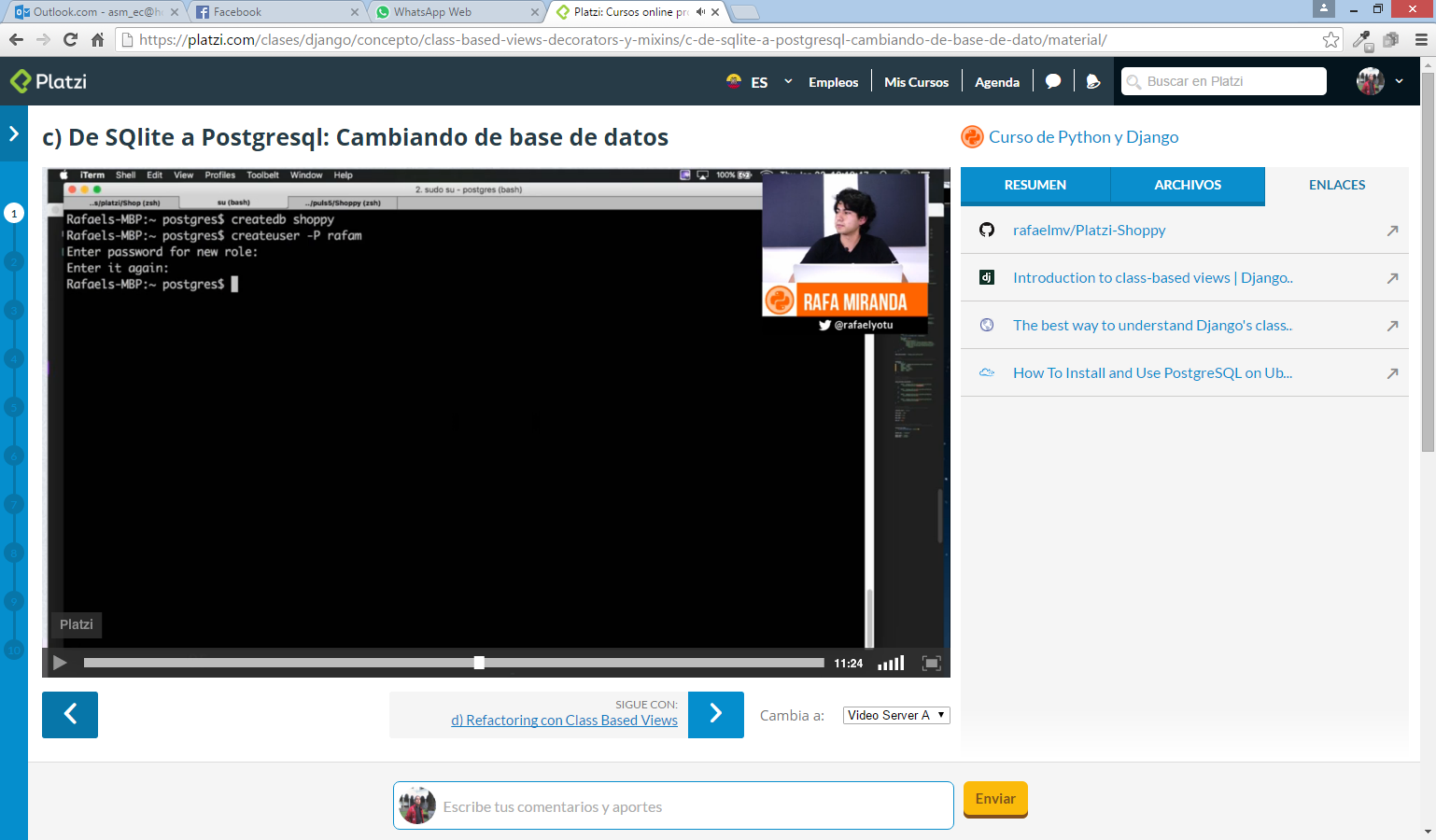


Se debe tener instalado también Postgres

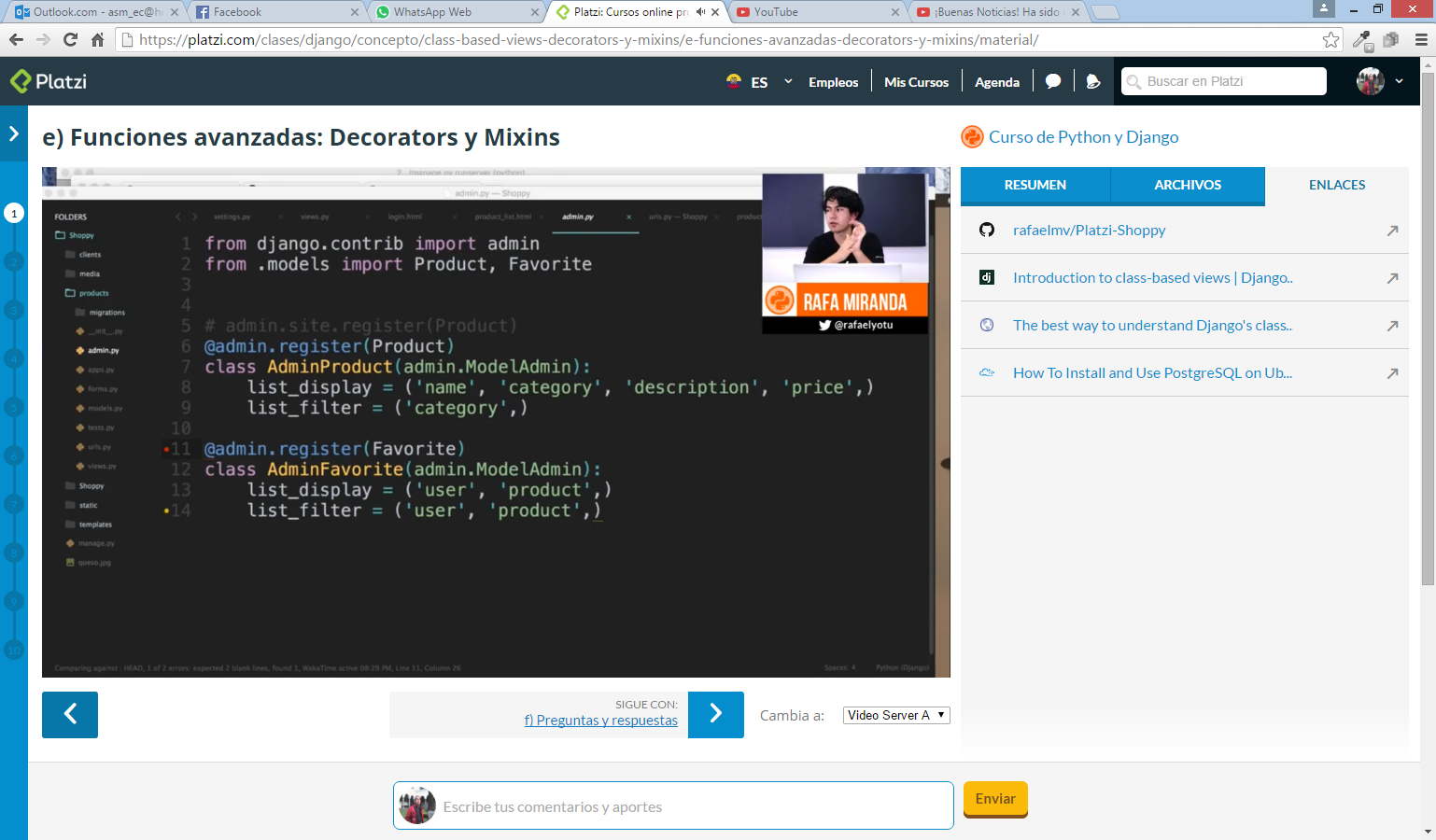
Se modifica settings py:



Al momento de realizar la migración nos dara un error, por lo que debemos ingresar a postgress con usuario y contraseña para poder crear la base de datos. Adicionalmente se debe crear el usuario y contraseña que está descrito en el archivo anterior:



Decoradores:



**Tutorial de autenticación con redes sociales usando Django**

pablotrinidadpablotrinidad Ene. 8, 2016En el desarrollo de un proyecto, a veces es más fácil dejar la parte de autenticación a otros como Facebook o Twitter. Hacer esto no sólo es más sencillo sino que **nos evita pensar en algunos temas delicados de seguridad** y nos permite poner más atención en otras partes más divertidas del desarrollo. Además, es común que los usuarios prefieran iniciar sesión con una red social que ya conocen a tener que recordar otra contraseña nueva.  
  
En este artículo te mostraré, paso a paso y desde cero, cómo crear un sencillo proyecto que nos permita iniciar sesión con Facebook, Twitter y GitHub.  
  
  
  
   
  
Recuerda que para empezar un nuevo proyecto con Django, es recomendable crear a la par un **nuevo entorno virtual de Python** para instalar todas sus dependencias.  
  
 En este artículo no tocaremos este tema, pero si usas Python2, puedes crearlos con ***[virtualenv](https://virtualenv.pypa.io/en/latest/" \t "_blank)****;*o si ya usas Python3, puedes crearlos con ***[pyvenv](https://docs.python.org/3/library/venv.html" \t "_blank)***o usar una librería super cool llamada **[virtualenv wrapper](https://virtualenvwrapper.readthedocs.org/en/latest/" \t "_blank)**.  
  
Lo primero que haremos será instalar Django con el comando:

**pip install django**

Con el siguiente comando crearemos una carpeta llamada Platzi que será donde vivirá todo el proyecto:

**mkdir Platzi**

Una vez dentro de la carpeta *Platzi,* iniciamos un nuevo proyecto de Django con el comando:

**django-admin.py startproject socialproject**

Ahora instalamos ***python social auth*** que es la librería que hace toda la magia con los proveedores de autenticación y nos permitirá iniciar sesión con Facebook, Twitter, Github, etc.

**pip install python-social-auth**

Después, instalamos la app en el ***settings.py***

INSTALLED\_APPS = (  
 'django.contrib.admin',  
 'django.contrib.auth',  
 'django.contrib.contenttypes',  
 'django.contrib.sessions',  
 'django.contrib.messages',  
 'django.contrib.staticfiles',  
  
 # Third-party apps  
 'social.apps.django\_app.default',  
)

E incluímos los procesadores de contexto de ***python-social-auth:***

'context\_processors': [  
 'django.template.context\_processors.debug',  
 'django.template.context\_processors.request',  
 'django.contrib.auth.context\_processors.auth',  
 'django.contrib.messages.context\_processors.messages',  
  
 # Python Social Auth Context Processors  
 'social.apps.django\_app.context\_processors.backends',  
 'social.apps.django\_app.context\_processors.login\_redirect',  
],

También incluimos las las urls en el archivo ***urls.py***

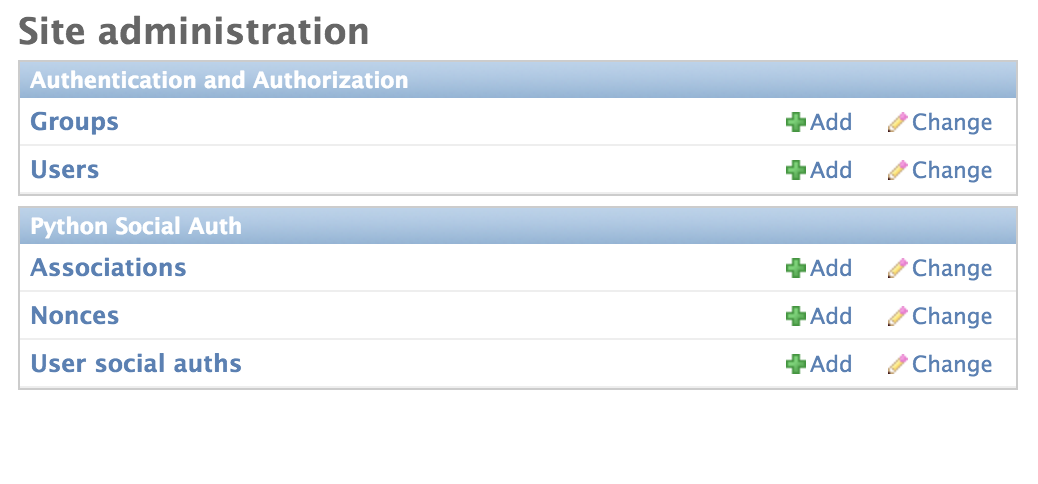
urlpatterns = [  
url(r'^admin/', include(admin.site.urls)),  
# Python Social Auth URLs  
url('', include('social.apps.django\_app.urls', **namespace**='social')),  
]

Para acabar de configurar ***python social auth,*** tenemos que aplicar los cambios en la base de datos con el comando:

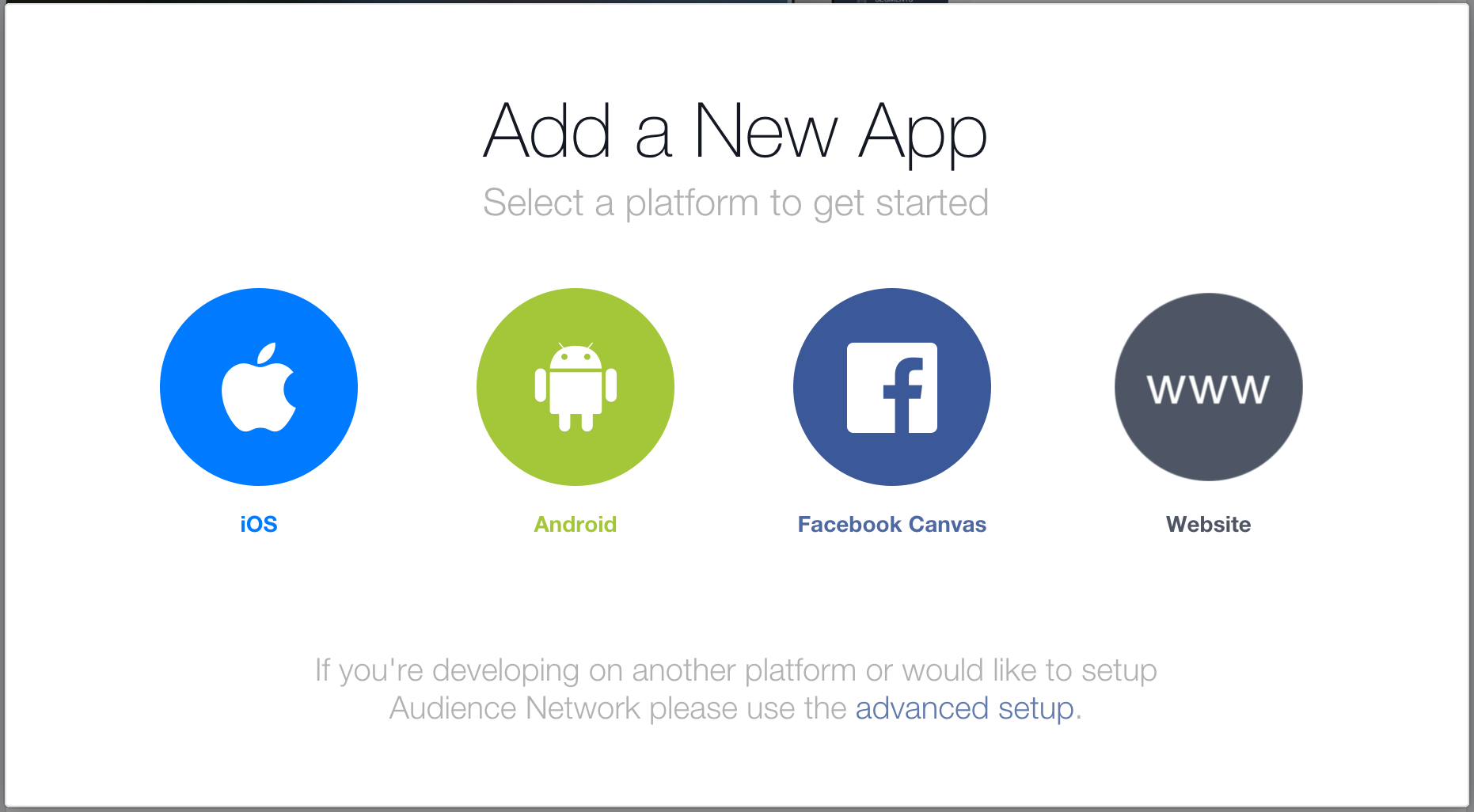
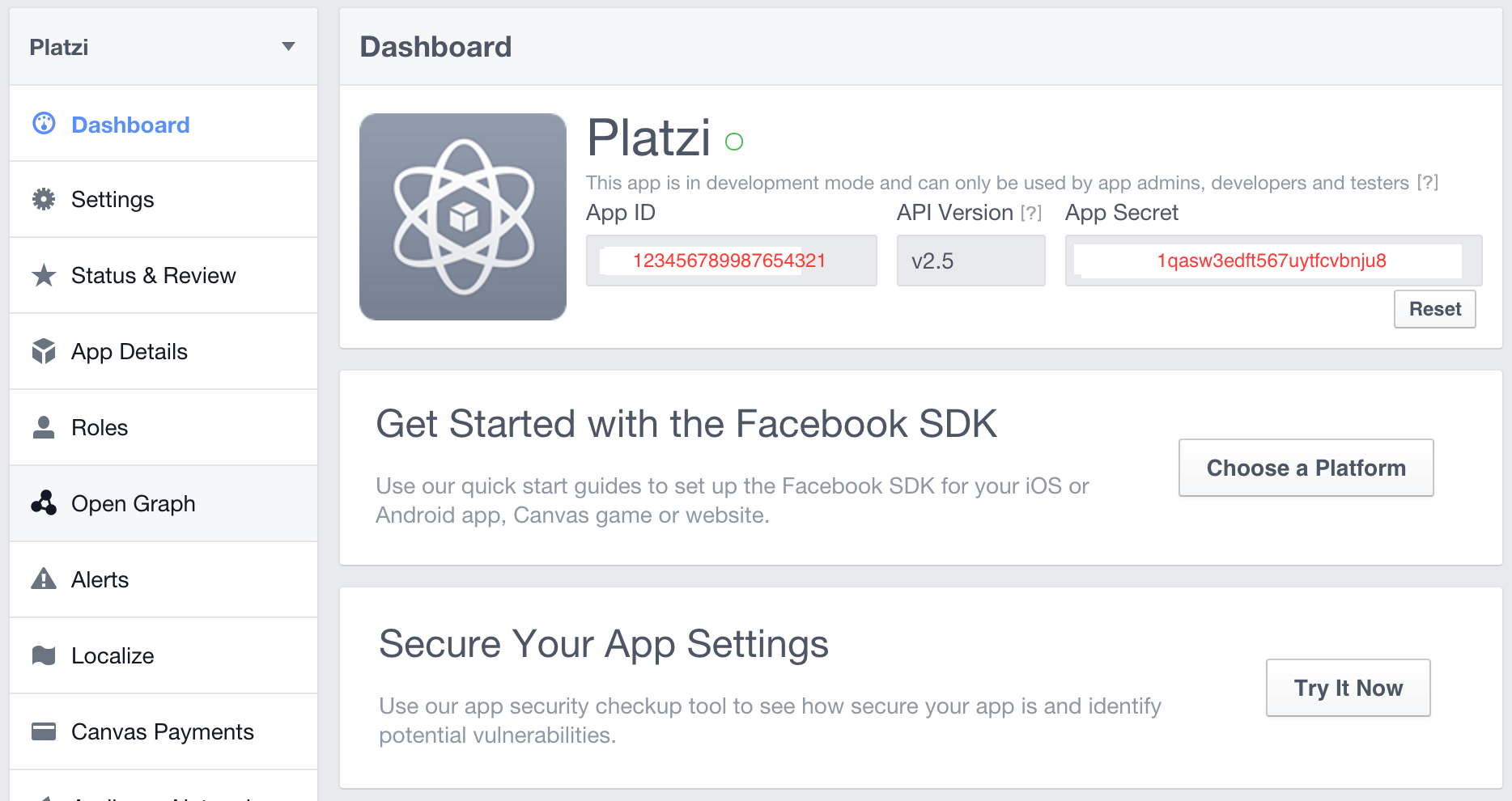
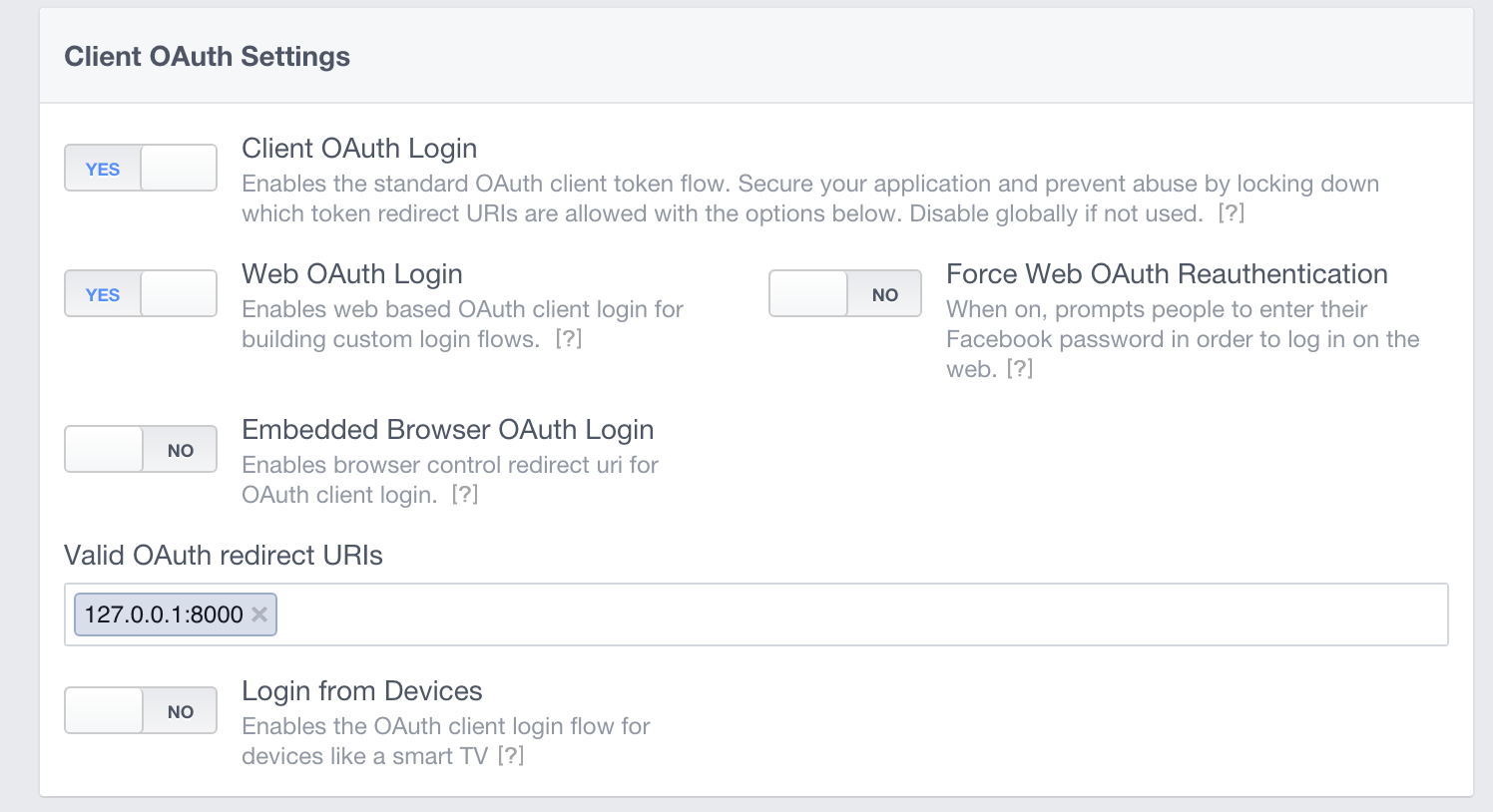
**python manage.py migrate**

Continuemos creando un super usuario para ver los cambios en la base de datos con el siguiente comando:

**python manage.py createsuperuser**

En el admin podemos ahora notar que se agregó la app **python social auth.**  
  
  
  
  
Hasta este punto ya tenemos todo configurado para empezar a trabajar con cada red social en específico.

**Facebook**

Para habilitar el login con Facebook tendremos que registrar un app en [**developers.facebook.com**](https://developers.facebook.com/). Seleccionamos website:  
  
  
  
  
  
Una vez dentro de nuestra app, hay dos valores que nos interesan, el **App ID**y el**App Secret.**  
  
  
  
  
  
Por último en **Settings > Advanced** tenemos que configurar las URL válidas a las que Facebook podrá redirigir una vez que haya concluido con el proceso de autenticación. Como ejemplo pondremos *127.0.0.1:8000* pero es recomendable configurar un host diferente.  
  
  
  
  
  
En nuestro **settings.py**ahora tenemos que agregar una nueva variable que le especifica a Django cuáles serán los servicios que usará para autenticar usuarios. Esta es **AUTHENTICATION\_BACKENDS:**

AUTHENTICATION\_BACKENDS = (  
# Facebook  
'social.backends.facebook.FacebookOAuth2',  
  
# Django  
'django.contrib.auth.backends.ModelBackend',  
)

No olvides incluir *‘****django.contrib.auth.backends.ModelBackend’***si estás usando ***django.contrib.auth***para autenticar usuarios o no podrás hacer el inicio de sesión por medio de username y password.  
  
En el **settings.py** tenemos que agregar las llaves que Facebook nos dio con sus respectivas variables y una variable que le indique a **python-social-auth**cual será la URL a la que se redirigirá la petición una vez que el proceso de autenticación haya concluido exitosamente. Recuerda que estas llaves son delicadas, así que cuida no exponerlas públicamente.

SOCIAL\_AUTH\_FACEBOOK\_KEY = '123456789'  
SOCIAL\_AUTH\_FACEBOOK\_SECRET = '123456ytdcvby78uhgfdsqw345678'  
  
SOCIAL\_AUTH\_LOGIN\_REDIRECT\_URL = "/"

El siguiente paso será crear una URL que nos muestre un template con el mensaje “Inicia sesión con Facebook” y otra URL que cierre la sesión del usuario. Para la primer URL usaremos la vista genérica ***TemplateView***que tenemos que importar. 

**from** django.views.**generic** **import** TemplateView

Y agregar la siguiente línea dentro de los **URL patterns**.

url(r'^$', TemplateView.as\_view(template\_name="home.html"), name="home"),

Para la URL de logout podemos usar la siguiente función ya escrita y sólo agregar esta línea:

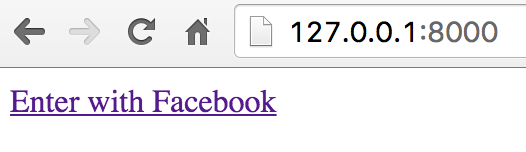
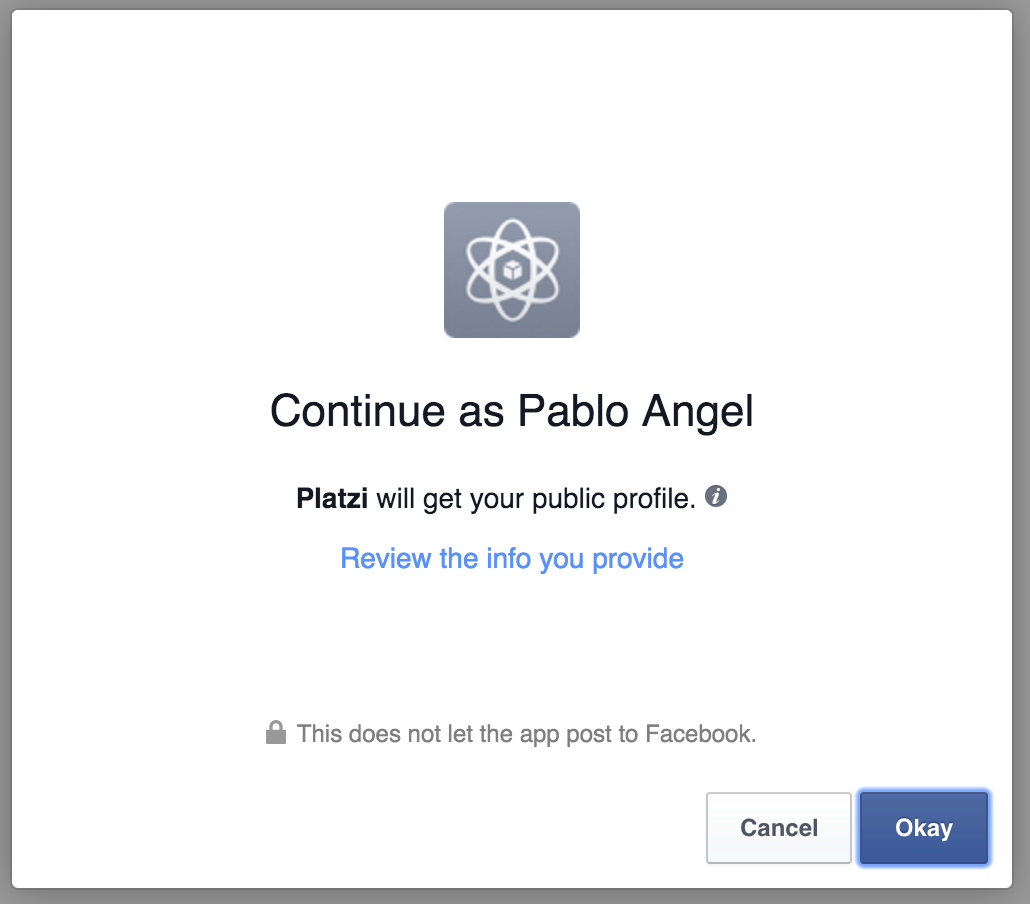
url(r'^users/logout/$', 'django.contrib.auth.views.logout', {'next\_page': '/'}, name="user-logout"),

Por lo que nuestro archivo final de **urls.py**se verá de la siguiente manera:

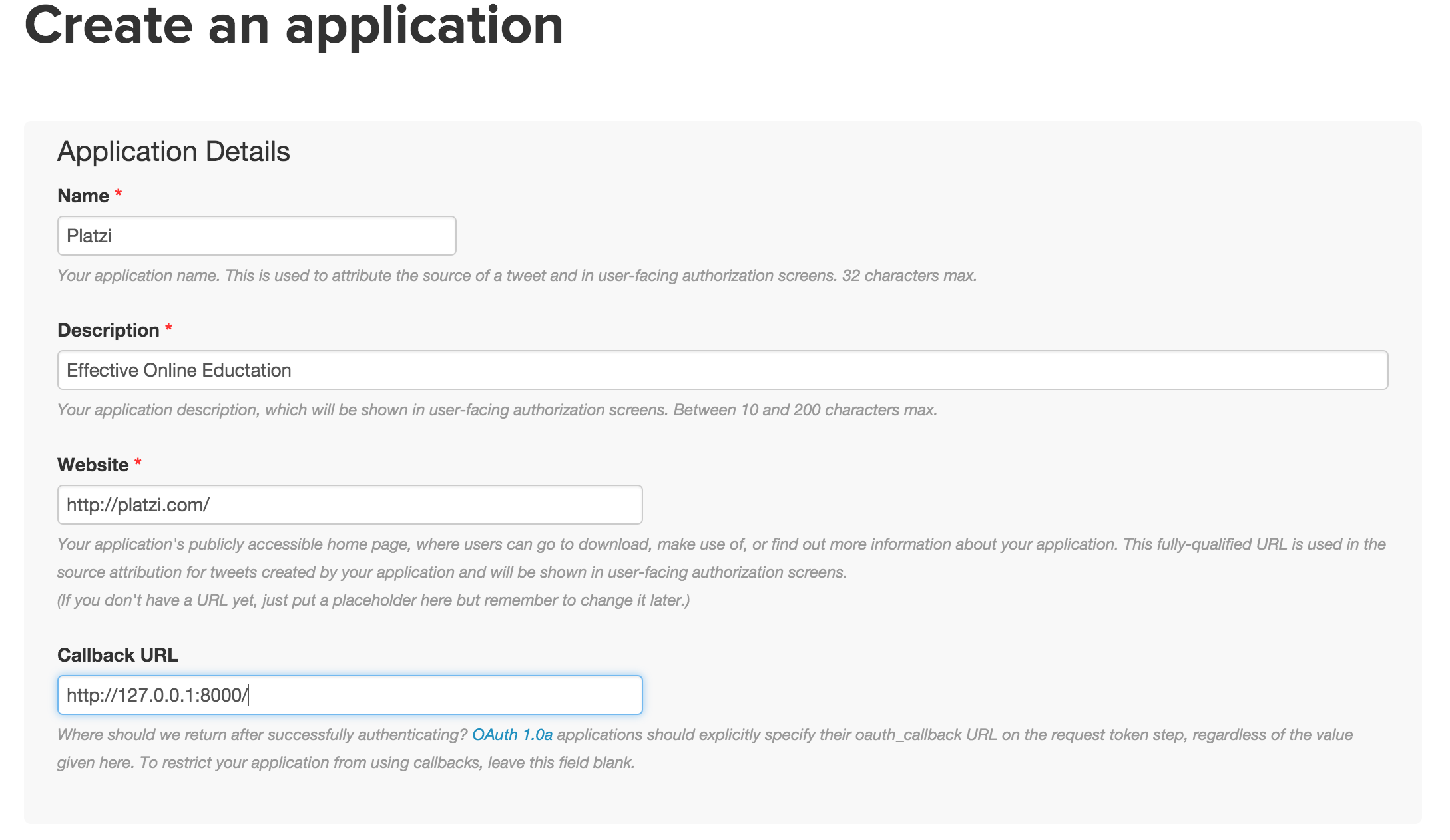
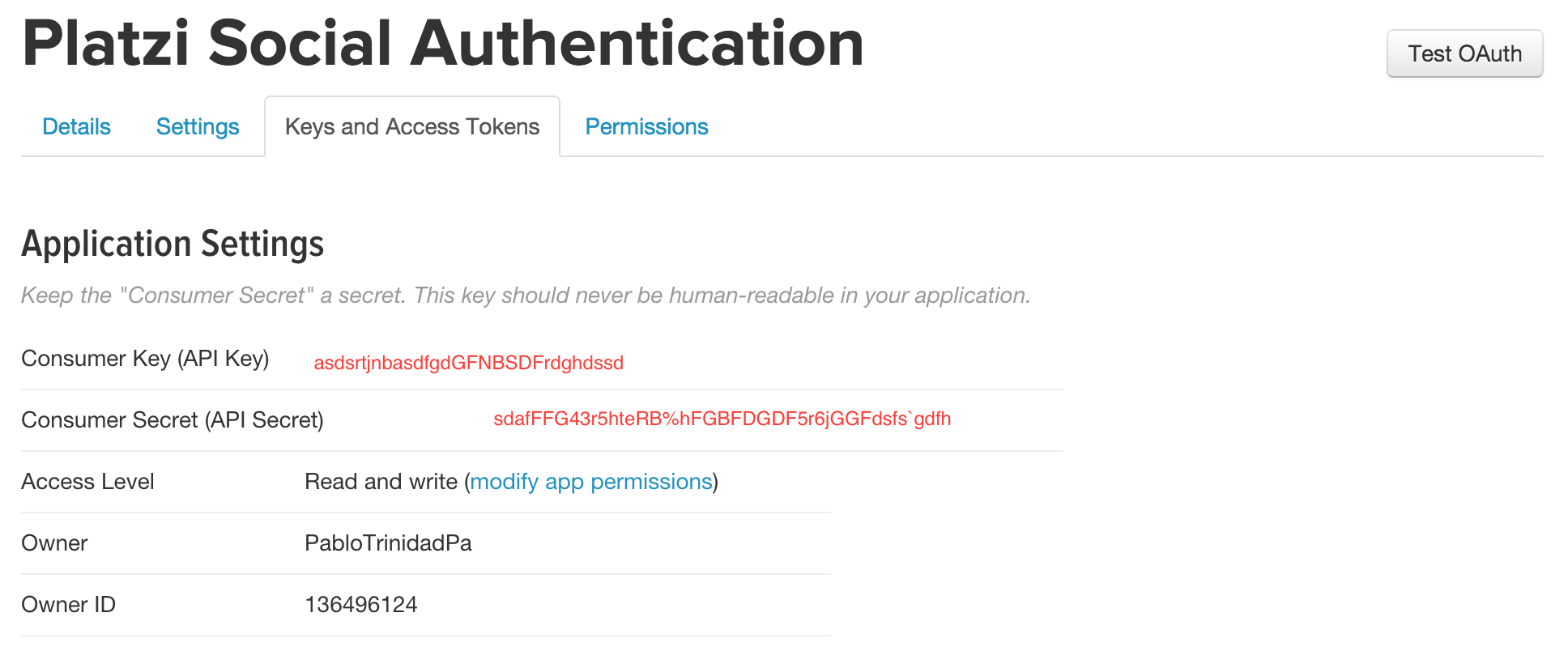
**from** django.conf.urls **import** include, url  
**from** django.contrib **import** admin  
**from** django.views.**generic** **import** TemplateView  
  
urlpatterns = [  
url(r'^admin/', include(admin.site.urls)),  
  
# Python Social Auth URLs  
url('', include('social.apps.django\_app.urls', **namespace**='social')),  
  
# Home URL  
url(r'^$', TemplateView.as\_view(template\_name="home.html"), name="home"),  
  
# Logout URL  
url(  
r'^users/logout/$',  
'django.contrib.auth.views.logout',  
{'next\_page': '/'},  
name="user-logout"  
),  
]

Ahora sólo tenemos que crear el template **home.html** y tendremos todo listo:

<!DOCTYPE html>  
**<html** **lang**="en"**>**  
**<head>**  
 **<meta** **charset**="UTF-8" **/>**  
 **<title>**Platzi**</title>**  
**</head>**  
**<body>**  
 {% if user and not user.is\_anonymous %}  
  
Welcome back {{ user.get\_full\_name }}.  
 **<a** **href**="{% url 'user-logout' %}"**>**Logout**</a>**  
  
 {% else %}  
 **<a** **href**="{% url 'social:begin' 'facebook' %}?next={{ request.path }}"**>**  
 Enter with Facebook  
 **</a>**  
 {% endif %}  
**</body>**  
**</html>**

Incluimos una sentencia *if* que valida si hay una sesión que no sea anónima; en caso de que no, le muestra un saludo con su nombre. En caso opuesto le mostrará el mensaje, “Inicia sesión con Facebook”.  
  
  
  
  
  
  
  
  
  
  

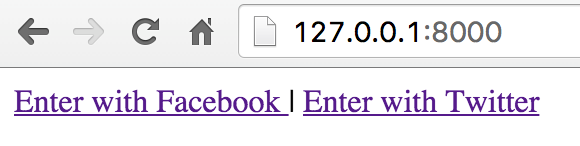
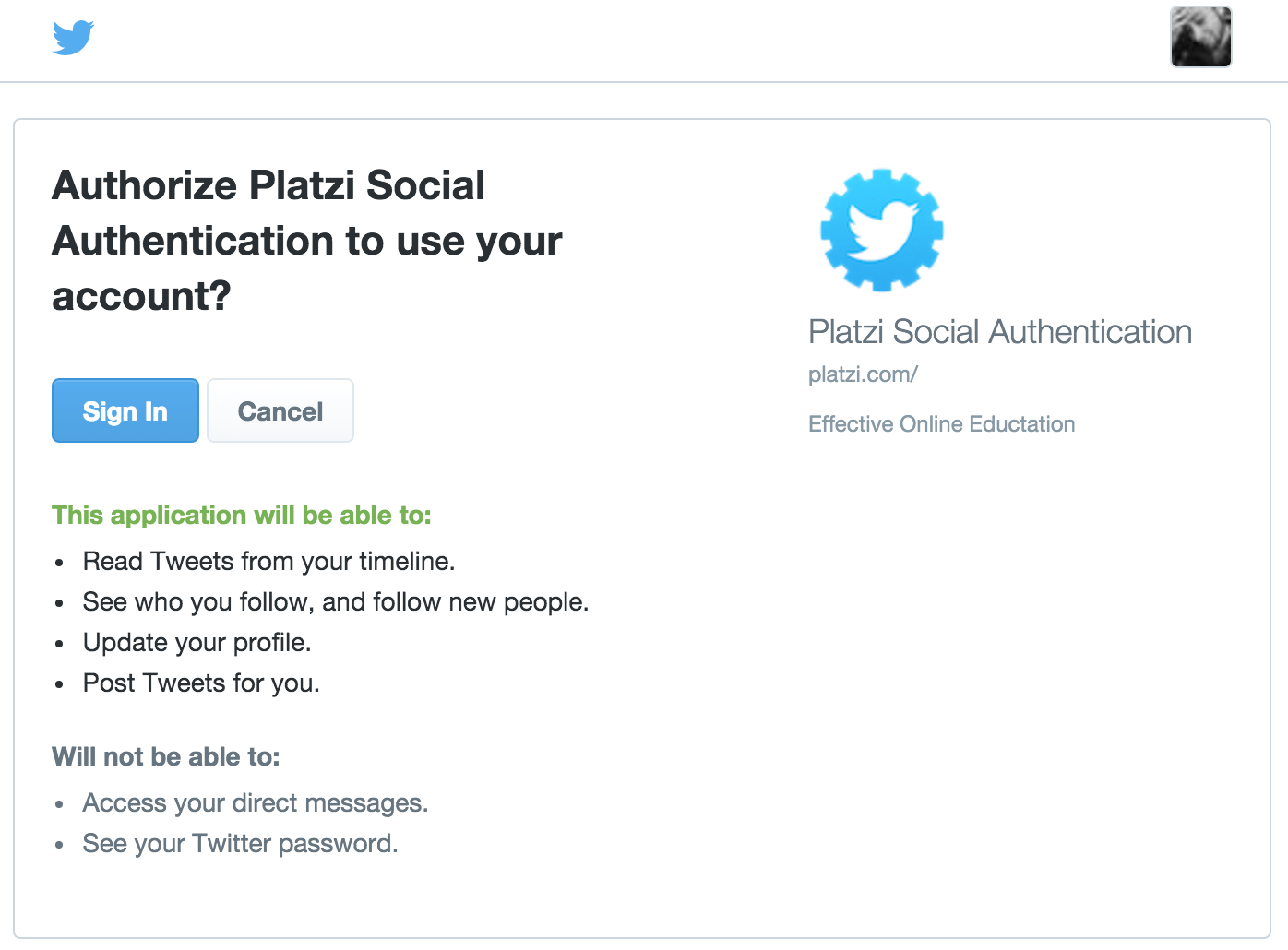
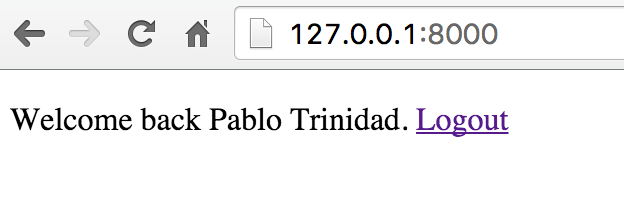

**Twitter**

De igual manera que con Facebook, tenemos que registrar una app en [**apps.twitter.com/app/new**](https://apps.twitter.com/app/new):  
  
  
  
  
  
  
En la sección **Keys and Access Tokens** encontrarás las llaves que necesitamos:  
  
  
  
  
Agregamos las llaves y el *Authentication Backend*al **settings.py:**

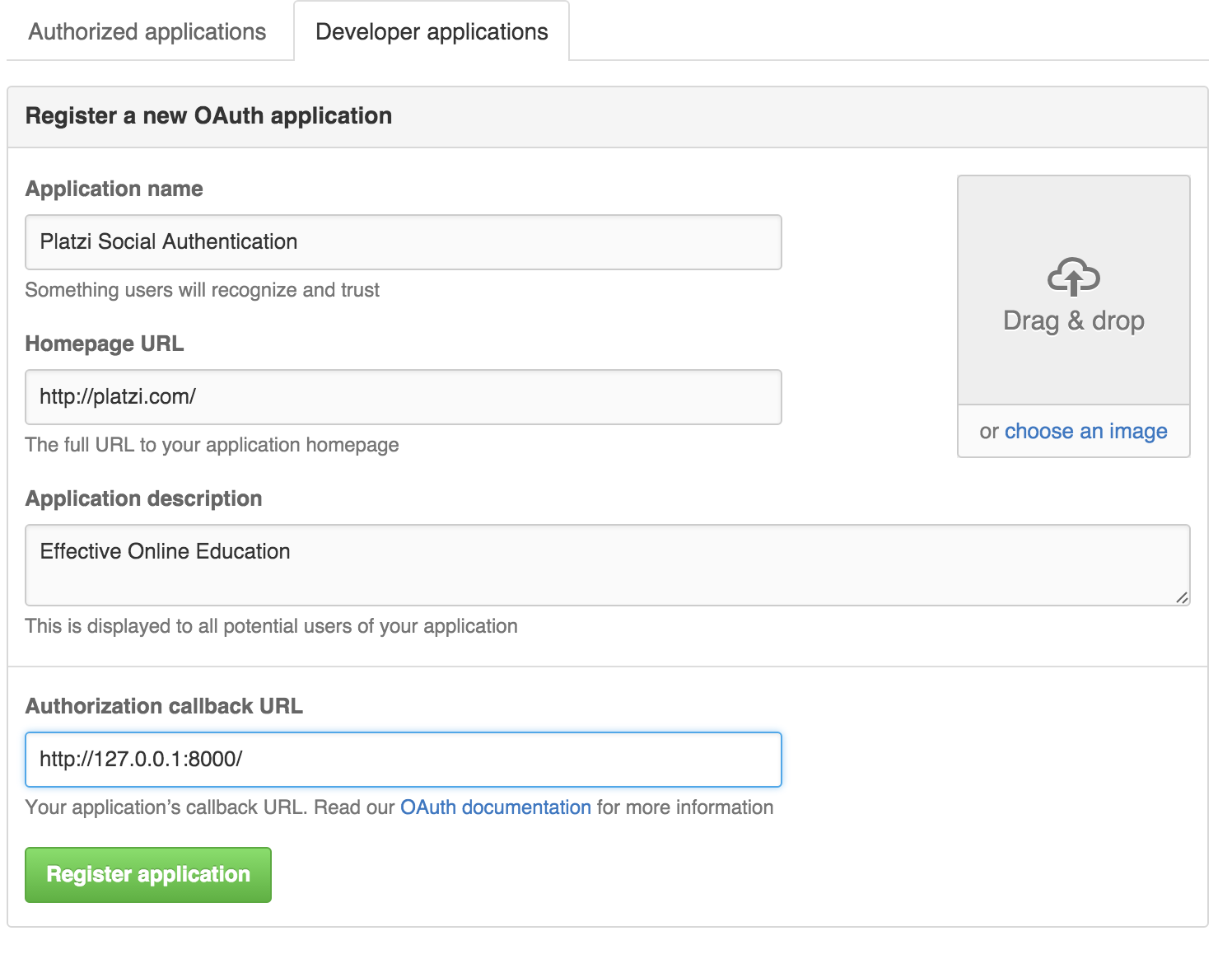
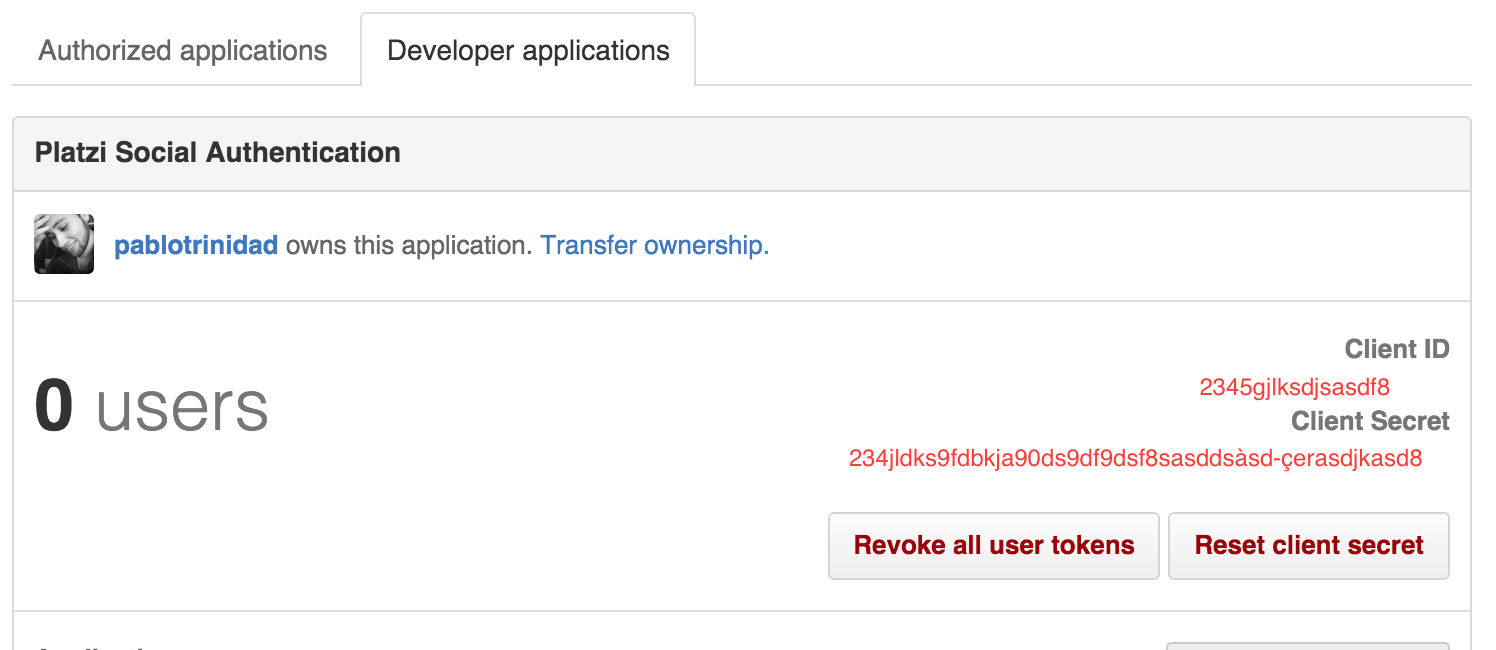
AUTHENTICATION\_BACKENDS = (  
 # Facebook  
 'social.backends.facebook.FacebookOAuth2',  
 # Twitter  
 'social.backends.twitter.TwitterOAuth',  
 # Django  
 'django.contrib.auth.backends.ModelBackend',  
)  
  
# Facebook Keys  
SOCIAL\_AUTH\_FACEBOOK\_KEY = 'q23456uhgf'  
SOCIAL\_AUTH\_FACEBOOK\_SECRET = 'qw4trgfdxc'  
# Twitter Keys  
SOCIAL\_AUTH\_TWITTER\_KEY = '123456yhgfdsvc'  
SOCIAL\_AUTH\_TWITTER\_SECRET = '123456789okjhgfd'

Sólo basta con editar el HTML para indicarle que puede iniciar sesión con Twitter agregando lo siguiente:

**<a** **href**="{% url 'social:begin' 'twitter' %}?next={{ request.path }}"**>**  
 Enter with Twitter  
 **</a>**

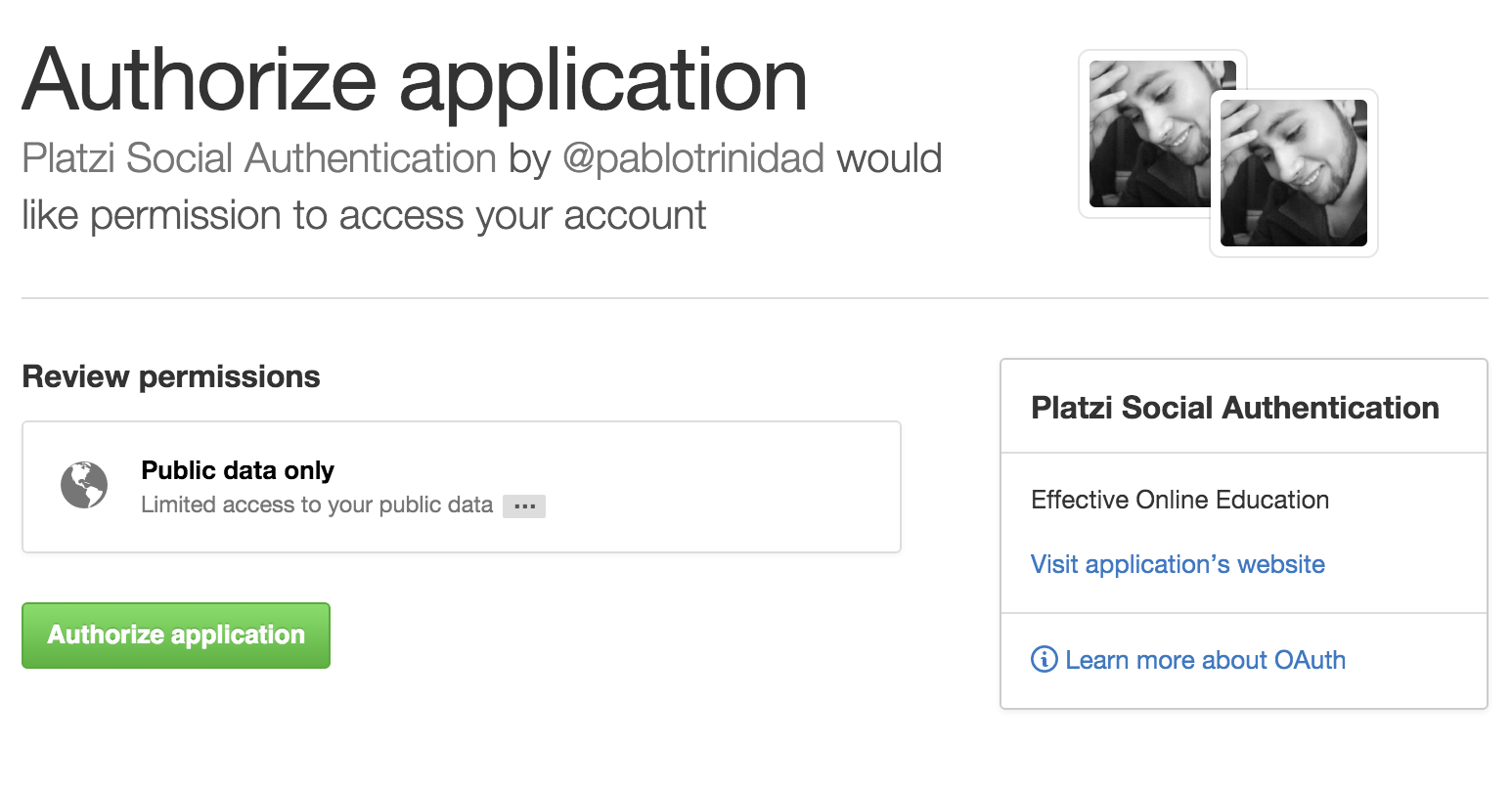
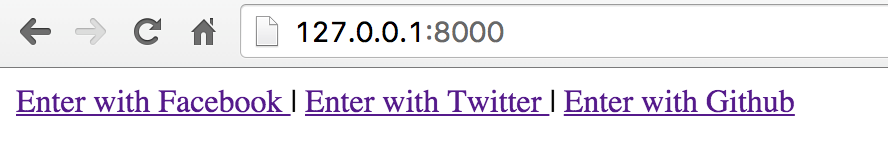
**GitHub**

Acá también tenemos que registrar una app en [**github.com**](https://github.com/settings/applications/new):  
  
  
  
  
Obtenemos las llaves:  
  
  
  
  
  
Registramos el ***Authentication Backend***y las llaves.

AUTHENTICATION\_BACKENDS = (  
 # Facebook  
 'social.backends.facebook.FacebookOAuth2',  
 # Twitter  
 'social.backends.twitter.TwitterOAuth',  
 # Github  
 'social.backends.github.GithubOAuth2',  
 # Django  
 'django.contrib.auth.backends.ModelBackend',  
)  
  
# Facebook Keys  
SOCIAL\_AUTH\_FACEBOOK\_KEY = 'q23456uhgf'  
SOCIAL\_AUTH\_FACEBOOK\_SECRET = 'qw4trgfdxc'  
  
# Twitter Keys  
SOCIAL\_AUTH\_TWITTER\_KEY = '123456yhgfdsvc'  
SOCIAL\_AUTH\_TWITTER\_SECRET = '123456789okjhgfd'  
  
# Github Keys  
SOCIAL\_AUTH\_GITHUB\_KEY = '21345thgfds'  
SOCIAL\_AUTH\_GITHUB\_SECRET = '324trghfdcvgfh'

Y agregamos el link para iniciar con GitHub:

**<a** **href**="{% url 'social:begin' 'github' %}?next={{ request.path }}"**>**  
Enter with Github  
**</a>**

¡Y listo!  
  
  
  
  
En resumen, lo que tienes que hacer es:  
  
Instalar Python Social Auth  
Configurar el settings  
Solicitar las llaves de acceso al medio de autenticación  
Registrar las llaves en el settings  
  
Puedes revisar y contribuir con el código fuente en **[GitHub](https://github.com/pablotrinidad/social-authentication" \t "_blank)**.

**Tu primera app CRUD en Django con Class Based Views**

pablotrinidadpablotrinidad Feb. 5, 2016Una vista es un elemento que se encarga de tomar una petición y regresar una respuesta HTTP. Éstas no necesariamente tienen que ser una función de Python, ¡también es posible usar clases en su lugar!  
  
El uso de clases como vistas en Django es también conocido como **Clases basadas en vistas** o **Class Based Views** y son útiles para mantener el código más legible y reutilizable.  
  
Cuando comenzamos a usar las clases basadas en vistas surge siempre la siguiente pregunta:  
  
*“¿Por qué no simplemente usar funciones y sólo copiar y pegar la lógica cada vez que lo necesite?”*  
Y la respuesta suele estar en la pregunta: *¿Por qué habríamos de copiar y pegar código a través de todo nuestro proyecto para hacer tareas tan cotidianas, cuando****la filosofía de Django siempre ha sido Don’t repeat yourself (DRY)****?*  
Las vistas basadas en clases fueron creadas con el mismo objetivo que las vistas basadas en funciones: **hacer el desarrollo de nuestro proyecto más fácil**. Aunque no nos guste admitirlo, el desarrollo web puede volverse monótono y aburrido en ciertos casos.  
  
Las **vistas genéricas** o **generic views** fueron creadas para aliviar ese dolor (pereza); tomando tareas comunes como mostrar una lista de objetos, crear, borrar o editar un objeto de manera simple al alcance de una clase.  
  
En este artículo iremos paso a paso a través de la creación de un sitio web que nos da control completo de los cursos que Platzi tiene mediante un **CRUD** (*create, read, update and delete*).

**Creación del proyecto**

Comencemos creando un proyecto llamado ***platzi.***

django-admin startproject platzi

Desde la raíz de nuestro proyecto crearemos una app llamada ***courses*** para almacenar toda la información de los cursos.

./manage.py startapp courses

Agregamos *‘courses’* a la variable **INSTALLED\_APPS** dentro de ***platzi/settings.py***:

INSTALLED\_APPS = (  
 :  
 'courses',  
 :  
)

**Creación del modelo**

Para este ejemplo, consideraremos que todos los cursos constan de un *nombre*, *fecha de inicio*,*fecha de fin* y una *imagen*. Aunque sabemos que en un caso real, el modelo tendría algunas relaciones extra como a los profesores, edición, carrera, etc.  
  
El arhivo del modelo dentro de ***courses/models.py*** quedará de la siguiente forma:

**from** django.db **import** models  
  
  
**class** Course(models.Model):  
 name = models.CharField(max\_length=140)  
 start\_date = models.DateTimeField()  
 end\_date = models.DateTimeField()  
 picture = models.ImageField(upload\_to='media/courses/pictures')

Después de definir el modelo creamos las migraciones:

./manage.py makemigrations

Y aplicamos las migraciones para crear la tabla en la base de datos.

./manage.py migrate

**Las vistas**

Para listar todos los cursos en una página usaremos la vista genérica**ListView.**Esta requiere que definamos el modelo que usará:

**from** django.views.**generic** **import** ListView  
**from** .models **import** Course  
  
**class** CourseList(ListView):  
 model = Course

Para mostrar el detalle de un curso en específico usaremos **DetailView,** que también sólo requiere que sea definido el modelo:

**from** django.views.**generic**.detail **import** DetailView  
  
**class** CourseDetail(DetailView):  
 model = Course

Para la creación de un nuevo curso usaremos **CreateView,**que requiere que sea definido el modelo, los campos que usará para la creación y la URL a la que debe redireccionar cuando la creación haya sido concluida:

**from** django.views.**generic**.edit **import** CreateView  
**from** django.core.urlresolvers **import** reverse\_lazy  
  
**class** CourseCreation(CreateView):  
 model = Course  
 success\_url = reverse\_lazy('courses:list')  
 fields = ['name', 'start\_date', 'end\_date', 'picture']

Para editar un objeto ya existente mediante un formulario usaremos **UpdateView** que requiere el modelo, los campos y la URL de éxito:

**from** django.views.**generic**.edit **import** UpdateView  
**from** django.core.urlresolvers **import** reverse\_lazy  
  
**class** CourseUpdate(UpdateView):  
 model = Course  
 success\_url = reverse\_lazy('courses:list')  
 fields = ['name', 'start\_date', 'end\_date', 'picture']

Por último, para borrar un objeto usaremos **DeleteView**que requiere del modelo y la URL de éxito:

**from** django.views.**generic**.edit **import** DeleteView  
**from** django.core.urlresolvers **import** reverse\_lazy  
  
**class** CourseDelete(DeleteView):  
 model = Course  
 success\_url = reverse\_lazy('courses:list')

Nuestro archivo final ***courses/views.py*** será:

**from** django.core.urlresolvers **import** reverse\_lazy  
  
**from** django.views.**generic** **import** ListView  
**from** django.views.**generic**.detail **import** DetailView  
**from** django.views.**generic**.edit **import** (  
 CreateView,  
 UpdateView,  
 DeleteView  
)  
  
**from** .models **import** Course  
  
  
**class** CourseList(ListView):  
 model = Course  
  
  
**class** CourseDetail(DetailView):  
 model = Course  
  
  
**class** CourseCreation(CreateView):  
 model = Course  
 success\_url = reverse\_lazy('courses:list')  
 fields = ['name', 'start\_date', 'end\_date', 'picture']  
  
  
**class** CourseUpdate(UpdateView):  
 model = Course  
 success\_url = reverse\_lazy('courses:list')  
 fields = ['name', 'start\_date', 'end\_date', 'picture']  
  
  
**class** CourseDelete(DeleteView):  
 model = Course  
 success\_url = reverse\_lazy('courses:list')

**Las URLs**

Ahora debemos tener las URLs donde nuestras vistas serán llamadas.Comencemos extendiendo el módulo principal de URLs dentro de ***platzi/urls.py*** de la siguiente forma:

**from** django.conf.urls **import** url, include  
**from** django.contrib **import** admin  
  
urlpatterns = [  
 url(r'^cursos/', include('courses.urls', **namespace**='courses')),  
 url(r'^admin/', admin.site.urls),  
]

Necesitamos agregar  un archivo **urls.py** dentro de **courses** de manera que el archivo***courses/urls.py*** luzca así:

**from** django.conf.urls **import** url  
  
**from** .views **import** (  
 CourseList,  
 CourseDetail,  
 CourseCreation,  
 CourseUpdate,  
 CourseDelete  
)  
  
urlpatterns = [  
  
 url(r'^$', CourseList.as\_view(), name='list'),  
 url(r'^(?P<pk>\d+)$', CourseDetail.as\_view(), name='detail'),  
 url(r'^nuevo$', CourseCreation.as\_view(), name='new'),  
 url(r'^editar/(?P<pk>\d+)$', CourseUpdate.as\_view(), name='edit'),  
 url(r'^borrar/(?P<pk>\d+)$', CourseDelete.as\_view(), name='delete'),  
  
]

**Templates**

Por último, necesitamos crear los templates HTML. Estos siguen una convención para se nombrados y así poder ser usados de manera automática por las clases genéricas (cosa que puede ser modificada).Para listar todos los cursos usaremos la convención ***model\_list.html***por lo que el nuestro se llamará ***course\_list.html*** y vivirá dentro de ***courses/templates/courses/course\_list.html***:

**<h1>**Cursos de Platzi**</h1>**  
  
**<p>**  
 **<a** **href**="{% url "**courses:new**" %}"**>**Agregar curso**</a>**  
**</p>**  
  
**<ul>**  
 {% for course in object\_list %}  
 **<li>**  
 **<p>**{{ course.name }}**</p>**  
 **<p>**  
 **<a** **href**="{% url "**courses:detail**" course.id %}"**>**Ver**</a>** |   
 **<a** **href**="{% url "**courses:edit**" course.id %}"**>**Editar**</a>** |   
 **<a** **href**="{% url "**courses:delete**" course.id %}"**>**Borrar**</a>**   
 **</p>**  
 **</li>**  
 {% endfor %}  
**</ul>**

Para agregar y editar necesitamos crear ***courses/templates/courses/course\_form.html*** de la siguiente manera:

**<form** **method**="post"**>**  
 {% csrf\_token %}  
 {{ form.as\_p }}  
 **<input** **type**="submit" **value**="Submit" **/>**  
**</form>**

Para ver el detalle necesitamos crear ***courses/templates/courses/course\_detail.html***:

**<h1>**{{ course.name }}**</h1>**  
**<p>**Fecha de inicio: **<i>**{{ course.start\_date }}**</i></p>**  
**<p>**Fecha de fin: **<i>**{{ course.end\_date }}**</i></p>**  
**<p>**  
 **<a** **href**="{% url "**courses:list**" %}"**>**Regresar**</a>**  
**</p>**

Por último necesitamos la pantalla de confirmación para borrar un curso que vivirá en***courses/templates/courses/courseconfirmdelete***:

**<form** **method**="post"**>**{% csrf\_token %}  
 ¿Estás seguro que deseas borrar el curso "{{ object }}"?  
 **<input** **type**="submit" **value**="Submit" **/>**  
**</form>**

Es necesario comentar que estas clases pueden ser modificadas con mucho mayor libertad para lograr cosas a la medida y más completas. ¡Toda esta *“magia”* es posible con simple herencia de clases!

PDF EN PYTHON Y DJANGO:

<https://youtu.be/UVXu-N_Zojw>

APIS EN PHYTON:

<https://youtu.be/lDD37Tuhfes>