

Python: Django Framework

Posma Group



Contenido

- 1 Introducción
- 2 Preparando un ambiente de desarrollo
- 3 Django: web framework
- 4 Creando un nuevo proyecto
- 5 Servidor de desarrollo
- 6 Configuración
- 7 Aplicaciones
- 8 Modelos
- 9 El administrador de Django
- 10 Django desde el intérprete interactivo
- 11 QuerySets
- 12 Vistas
- 13 Plantillas
- 14 Probando nuestro sitio

Introducción

En los talleres anteriores hemos expuesto las bases de la programación en Python. Sin embargo, es nuestro objetivo adquirir suficiente conocimiento para desarrollar proyectos de aplicación web, para ello exploraremos de manera práctica las herramientas que nos provee el framework *Django*.

Ambiente de desarrollo

Lo primero que haremos será crear un directorio dedicado al proyecto en el que vamos a trabajar.

```
$ mkdir mi_proyecto  
$ cd mi_proyecto
```

Ambiente de desarrollo

Lo primero que haremos será crear un directorio dedicado al proyecto en el que vamos a trabajar.

```
$ mkdir mi_proyecto  
$ cd mi_proyecto
```

Para comenzar a implementar cualquier proyecto, es necesario preparar un entorno de trabajo que nos provea las herramientas necesarias, facilitándonos el desarrollo de manera ordenada y sin conflictos con otros proyectos existentes.

Entorno virtual

virtualenv

- Es una herramienta que nos permite definir entornos aislados para cada uno de nuestros proyectos Python.

Entorno virtual

virtualenv

- Es una herramienta que nos permite definir entornos aislados para cada uno de nuestros proyectos Python.
- Hace posible que en una misma máquina convivan diversos proyectos con versiones distintas de python y de las distintas dependencias externas.

Entorno virtual

virtualenv

- Es una herramienta que nos permite definir entornos aislados para cada uno de nuestros proyectos Python.
- Hace posible que en una misma máquina convivan diversos proyectos con versiones distintas de python y de las distintas dependencias externas.

Para saber si virtualenv está ya instalado en el sistema, podemos intentar consultar su versión:

```
$ virtualenv --version
```


Entorno virtual

Si *virtualenv* no está instalado, en Debian puede hacerse de la siguiente forma:

```
$ sudo apt-get install python-virtualenv
```

Entorno virtual

Si *virtualenv* no está instalado, en Debian puede hacerse de la siguiente forma:

```
$ sudo apt-get install python-virtualenv
```

Una vez que está instalado, ya podemos crear un entorno virtual:

```
$ virtualenv env --no-site-packages
```

Entorno virtual

Si *virtualenv* no está instalado, en Debian puede hacerse de la siguiente forma:

```
$ sudo apt-get install python-virtualenv
```

Una vez que está instalado, ya podemos crear un entorno virtual:

```
$ virtualenv env --no-site-packages
```

La opción `--no-site-packages` significa que el nuevo entorno virtual no instalará los paquetes que se encuentran instalados globalmente.

Entorno virtual

Para activar el entorno, nos movemos dentro del directorio `bin`, y ejecutamos:

```
$ cd env/bin  
$ source activate  
(env)$
```

Entorno virtual

Para activar el entorno, nos movemos dentro del directorio `bin`, y ejecutamos:

```
$ cd env/bin  
$ source activate  
(env)$
```

Para salir del entorno, ejecutamos la instrucción `deactivate`.

Instalación de paquetes

pip

Es un manejador de paquetes para Python, que hace posible la instalación y actualización de bibliotecas fácilmente desde la línea de comandos.

Instalación de paquetes

pip

Es un manejador de paquetes para Python, que hace posible la instalación y actualización de bibliotecas fácilmente desde la línea de comandos.

Para comenzar, instalaremos *Django*, que es el framework que vamos a utilizar.

```
$ pip install django
```

Instalación de paquetes

pip

Es un manejador de paquetes para Python, que hace posible la instalación y actualización de bibliotecas fácilmente desde la línea de comandos.

Para comenzar, instalaremos *Django*, que es el framework que vamos a utilizar.

```
$ pip install django
```

Si queremos ver una lista de las bibliotecas instaladas en el entorno actual, ejecutamos:

```
$ pip freeze
```


Instalación de paquetes

Si la instalación de Django se ha hecho correctamente, podemos importarlo desde el intérprete. Hagamos la siguiente prueba:

```
>>> import django
>>> print django.get_version()
1.6
```

Instalación de paquetes

Si la instalación de Django se ha hecho correctamente, podemos importarlo desde el intérprete. Hagamos la siguiente prueba:

```
>>> import django
>>> print django.get_version()
1.6
```

Ya estamos listos para comenzar a usar Django!

Django framework

- Django es uno de los frameworks más populares para el desarrollo web en Python.

Django framework

- Django es uno de los frameworks más populares para el desarrollo web en Python.
- Está diseñado para permitir el desarrollo de aplicaciones web de grandes dimensiones, en un tiempo relativamente corto.

Django framework

- Django es uno de los frameworks más populares para el desarrollo web en Python.
- Está diseñado para permitir el desarrollo de aplicaciones web de grandes dimensiones, en un tiempo relativamente corto.
- Soporta varias bases de datos (MySQL, SQLite, Postgres, MS-SQL)

Django framework

- Django es uno de los frameworks más populares para el desarrollo web en Python.
- Está diseñado para permitir el desarrollo de aplicaciones web de grandes dimensiones, en un tiempo relativamente corto.
- Soporta varias bases de datos (MySQL, SQLite, Postgres, MS-SQL)
- Interfaz administrativa automática

MVC

Django implementa su propia versión del patrón MVC. El framework consta de 3 capas:

- Modelos
- Vistas
- Plantillas

Django framework

Modelos (models)

Definen la fuente y la estructura de los datos que la aplicación maneja. Esta capa se implementa en la forma de un ORM (Object Relational Mapping), con el cual se simplifica de manera considerable el acceso a los datos.

Django framework

Modelos (models)

Definen la fuente y la estructura de los datos que la aplicación maneja. Esta capa se implementa en la forma de un ORM (Object Relational Mapping), con el cual se simplifica de manera considerable el acceso a los datos.

Vistas (views)

Se comunica con los modelos del ORM de Django y entrega los datos a las plantillas. Definen cuáles datos serán mostrados al usuario.

Django framework

Modelos (models)

Definen la fuente y la estructura de los datos que la aplicación maneja. Esta capa se implementa en la forma de un ORM (Object Relational Mapping), con el cual se simplifica de manera considerable el acceso a los datos.

Vistas (views)

Se comunica con los modelos del ORM de Django y entrega los datos a las plantillas. Definen cuáles datos serán mostrados al usuario.

Plantillas (templates)

Interfaz entre el usuario y el sistema. En esta capa se define la forma en la que se muestran los datos.

Django framework

Adicionalmente, Django provee las siguientes funcionalidades:

- Clases para el manejo de formularios
- Autenticación
- Interfaz administrativa
- Internacionalización
- Cache

Django framework

Adicionalmente, Django provee las siguientes funcionalidades:

- Clases para el manejo de formularios
- Autenticación
- Interfaz administrativa
- Internacionalización
- Cache

...Y otras más.

Creando un nuevo proyecto

Para crear un proyecto en django, ejecutamos:

```
$ django-admin.py startproject libronline
```

Creando un nuevo proyecto

Para crear un proyecto en django, ejecutamos:

```
$ django-admin.py startproject libronline
```

Esto debió crear una carpeta libronline dentro de la carpeta actual, con una serie de archivos necesarios para el funcionamiento de Django.

Creando un nuevo proyecto

Para crear un proyecto en django, ejecutamos:

```
$ django-admin.py startproject libronline
```

Esto debió crear una carpeta libronline dentro de la carpeta actual, con una serie de archivos necesarios para el funcionamiento de Django.

Nota: Es importante evitar el uso de nombres que puedan generar conflictos con Python o los componentes de Django. Por ejemplo, se recomienda no utilizar palabras como “django” o “test”.

Servidor de desarrollo

El archivo `manage.py` sirve como interfaz con diversas opciones para el manejo del proyecto.

Servidor de desarrollo

El archivo `manage.py` sirve como interfaz con diversas opciones para el manejo del proyecto.

Por ejemplo, podemos ejecutar `manage.py` con la opción `runserver` para iniciar el servidor de prueba:

```
$ manage.py runserver
Validating models...

0 errors found
February 10, 2014 - 16:54:57
Django version 1.6.2, using settings 'libronline.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-C.
```

Servidor de desarrollo

El archivo `manage.py` sirve como interfaz con diversas opciones para el manejo del proyecto.

Por ejemplo, podemos ejecutar `manage.py` con la opción `runserver` para iniciar el servidor de prueba:

```
$ manage.py runserver
Validating models...

0 errors found
February 10, 2014 - 16:54:57
Django version 1.6.2, using settings 'libronline.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-C.
```

Ahora podemos probar ingresar el URL `http://127.0.0.1:8000/` en algún navegador.

Configuración

En el archivo `settings.py` se encuentran definidos todos los parámetros de configuración del proyecto.

Configuración

En el archivo `settings.py` se encuentran definidos todos los parámetros de configuración del proyecto.

Base de datos

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

Configuración

En el archivo `settings.py` se encuentran definidos todos los parámetros de configuración del proyecto.

Base de datos

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

Si se está utilizando un motor de base de datos distinto de SQLite, es necesario definir los parámetros `USER`, `PASSWORD` y `HOST`. Además, es necesario crear la base de datos previamente a esta configuración.

Parámetros locales

Para especificar la zona horaria en la que habita nuestra aplicación, configuramos el parámetro `TIME_ZONE`. En el caso de Venezuela, el valor correcto es `'America/Caracas'`.

Parámetros locales

Para especificar la zona horaria en la que habita nuestra aplicación, configuramos el parámetro `TIME_ZONE`. En el caso de Venezuela, el valor correcto es `'America/Caracas'`.

También podemos especificar el lenguaje a utilizar en la opción `LANGUAGE_CODE`, al cual le daremos el valor de `'es-VE'` para el caso de Venezuela.

Iniciando la base de datos

A continuación, ejecutaremos la siguiente instrucción:

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


Iniciando la base de datos

A continuación, ejecutaremos la siguiente instrucción:

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

La instrucción syncdb crea la tablas necesarias del proyecto en la base de datos. La primera vez que se ejecuta, el sistema preguntará al usuario si desea crear un usuario con permisos administrativos. Es recomendable hacerlo.

Iniciando la base de datos

A continuación, ejecutaremos la siguiente instrucción:

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

La instrucción syncdb crea la tablas necesarias del proyecto en la base de datos. La primera vez que se ejecuta, el sistema preguntará al usuario si desea crear un usuario con permisos administrativos. Es recomendable hacerlo.

Para explorar la base de datos SQLite, podemos utilizar el plugin de firefox SQLite Manager, o instalar sqliteman usando `apt-get install`.

Aplicaciones

Un proyecto en Django consta de un conjunto de aplicaciones, éstas no son más que paquetes de Python que siguen una convención determinada.

Aplicaciones

Un proyecto en Django consta de un conjunto de aplicaciones, éstas no son más que paquetes de Python que siguen una convención determinada.

Para crear una aplicación, ejecutamos la instrucción `startapp` a través de `manage.py`:

```
$ python manage.py startapp libros
```

Aplicaciones

Un proyecto en Django consta de un conjunto de aplicaciones, éstas no son más que paquetes de Python que siguen una convención determinada.

Para crear una aplicación, ejecutamos la instrucción `startapp` a través de `manage.py`:

```
$ python manage.py startapp libros
```

```
libros/  
  __init__.py  
  admin.py  
  models.py  
  tests.py  
  views.py
```

Aplicaciones

- `__init__.py` indica que nuestra aplicación es un paquete válido de Python.

Aplicaciones

- `__init__.py` indica que nuestra aplicación es un paquete válido de Python.
- En `admin.py` se colocará el código referente al administrador de Django para la aplicación actual.

Aplicaciones

- `__init__.py` indica que nuestra aplicación es un paquete válido de Python.
- En `admin.py` se colocará el código referente al administrador de Django para la aplicación actual.
- En `models.py` se escribirá todo el código de los modelos.

Aplicaciones

- `__init__.py` indica que nuestra aplicación es un paquete válido de Python.
- En `admin.py` se colocará el código referente al administrador de Django para la aplicación actual.
- En `models.py` se escribirá todo el código de los modelos.
- `tests.py` funciona para pruebas unitarias.

Aplicaciones

- `__init__.py` indica que nuestra aplicación es un paquete válido de Python.
- En `admin.py` se colocará el código referente al administrador de Django para la aplicación actual.
- En `models.py` se escribirá todo el código de los modelos.
- `tests.py` funciona para pruebas unitarias.
- Y en `views.py` se implementarán las vistas propias de la aplicación libros

Modelos

Los modelos representan las entidades en nuestro sistema, y tienen una correspondencia directa con la Base de Datos mediante el ORM de Django.

Modelos

Los modelos representan las entidades en nuestro sistema, y tienen una correspondencia directa con la Base de Datos mediante el ORM de Django.

Para definir nuestros modelos, editaremos el archivo `models.py` y crearemos dos clases: `Autor` y `Libro`.

```
# -*- coding: utf-8 -*-  
from django.db import models  
  
class Autor(models.Model):  
    nombre = models.CharField(max_length=200)  
  
    def __unicode__(self):  
        return self.nombre
```

Modelos

```
class Libro(models.Model):
    GENERO_CHOICES = (
        (1, 'Novela'),
        (2, 'Crónica'),
        (3, 'Ensayo'),
        (4, 'Académico'),
        (5, 'Biografía')
    )

    titulo = models.CharField(max_length=200)
    fecha_pub = models.DateField()
    autor = models.ForeignKey(Autor)
    genero = models.IntegerField(choices=GENERO_CHOICES, default=1)

    def __unicode__(self):
        return "%s - %s" % (self.titulo, self.autor.nombre)
```

Modelos

- Cada modelo se define como una clase que hereda de `django.db.models.Model`, definiendo a su vez una serie de atributos de clase.

Modelos

- Cada modelo se define como una clase que hereda de `django.db.models.Model`, definiendo a su vez una serie de atributos de clase.
- Cada campo es una instancia de la clase `django.db.models.Field`, cuyas subclasses implementan los distintos tipos de datos.

Modelos

- Cada modelo se define como una clase que hereda de `django.db.models.Model`, definiendo a su vez una serie de atributos de clase.
- Cada campo es una instancia de la clase `django.db.models.Field`, cuyas subclasses implementan los distintos tipos de datos.
- Un `ForeignKey` puede verse como una relación “uno a muchos”, ya que distintos modelos pueden definir la misma llave foránea con un modelo dado.

Modelos

- Cada modelo se define como una clase que hereda de `django.db.models.Model`, definiendo a su vez una serie de atributos de clase.
- Cada campo es una instancia de la clase `django.db.models.Field`, cuyas subclasses implementan los distintos tipos de datos.
- Un `ForeignKey` puede verse como una relación “uno a muchos”, ya que distintos modelos pueden definir la misma llave foránea con un modelo dado.
- Se han implementado además los métodos `__unicode__`. Éstos definen la representación textual de las instancias particulares de cada clase.

Tipos de campo

Django soporta los siguientes tipos de campo para los modelos:

- `AutoField`
- `BigIntegerField`
- `BinaryField`
- `BooleanField`
- `CharField`
- `CommaSeparatedIntegerField`
- `DateField`
- `DateTimeField`
- `DecimalField`

Tipos de campo

- EmailField
- FileField
- FilePathField
- FloatField
- ImageField
- IntegerField
- IPAddressField
- NullBooleanField
- PositiveIntegerField
- URLField

Tipos de campo

Existen también los tipos de campo relacionales:

- ForeignKey
- ManyToManyField
- OneToOneField

Agregando nuestra aplicación

El siguiente paso es agregar nuestra aplicación *libros* a la lista de aplicaciones instaladas en el `settings.py`:

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
  
    'libros',  
)
```

Agregando nuestra aplicación

El siguiente paso es agregar nuestra aplicación *libros* a la lista de aplicaciones instaladas en el `settings.py`:

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
  
    'libros',  
)
```

Ahora necesitamos sincronizar la base de datos con la instrucción `syncdb`:

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

El administrador de Django

Una de las características principales de Django es que viene con una interfaz administrativa que facilita mucho el trabajo. A través de esta interfaz podemos manejar todas las entidades en el sistema.

El administrador de Django

Una de las características principales de Django es que viene con una interfaz administrativa que facilita mucho el trabajo. A través de esta interfaz podemos manejar todas las entidades en el sistema.

Para habilitar un modelo en la interfaz administrativa, editamos el archivo `libros/admin.py`:

```
from django.contrib import admin
from libros.models import Libro, Autor

admin.site.register(Libro)
admin.site.register(Autor)
```


El administrador de Django

A continuación, ejecutamos el servidor de prueba con `python manage.py runserver` y abrimos el navegador con el URL `http://127.0.0.1:8000/admin/`

El administrador de Django

A continuación, ejecutamos el servidor de prueba con `python manage.py runserver` y abrimos el navegador con el URL `http://127.0.0.1:8000/admin/`

Django administration

Site administration

Auth	
Groups	 Add  Change
Users	 Add  Change
Libros	
Autors	 Add  Change
Libros	 Add  Change

El administrador de Django

Evidentemente no hay objetos creados aún, añadiremos primero un autor haciendo click en su sección Autores y luego en el botón de la esquina Añadir autor.

El administrador de Django

Evidentemente no hay objetos creados aún, añadiremos primero un autor haciendo click en su sección Autores y luego en el botón de la esquina Añadir autor.

Seguidamente introducimos el nombre del autor y hacemos click en Grabar. Ahora podemos ver el nuevo autor en el listado:



Escoja autor a modificar

Acción:	<input type="text" value="....."/>	<input type="button" value="Ir"/>	seleccionados 0 de 1
<input type="checkbox"/>	Autor		
<input type="checkbox"/>	Herman Hesse		
1 autor			

El administrador de Django

Ahora añadiremos un libro...



Añadir libro

Título:	<input type="text" value="Demian"/>
Fecha pub:	<input type="text" value="12/02/1919"/> Hoy 
Autor:	<input type="text" value="Herman Hesse"/> ▼ 
Genero:	<input type="text" value="Novela"/> ▼

El administrador de Django

Ahora añadiremos un libro...

Añadir libro

Título:	<input type="text" value="Demian"/>
Fecha pub:	<input type="text" value="12/02/1919"/> Hoy 
Autor:	<input type="text" value="Herman Hesse"/> ▼ 
Genero:	<input type="text" value="Novela"/> ▼
<input type="button" value="Guardar"/>	

Como podemos ver, es bastante sencillo manipular los objetos del sistema a través del administrador, habiendo definido únicamente los modelos.

El administrador de Django

Nombres plurales

Podemos notar que Django ha intentado inferir automáticamente el plural de `Autor`, y ha colocado `Autors` en su lugar, lo cual no es correcto. Para especificar este tipo de detalles, se define en Django una clase interna llamada `Meta`:

El administrador de Django

Nombres plurales

Podemos notar que Django ha intentado inferir automáticamente el plural de `Autor`, y ha colocado `Autors` en su lugar, lo cual no es correcto. Para especificar este tipo de detalles, se define en Django una clase interna llamada `Meta`:

```
class Autor(models.Model):  
    class Meta:  
        verbose_name_plural = "Autores"  
    # ...
```


Django desde el intérprete interactivo

Es posible tener acceso a nuestro proyecto Django desde el intérprete interactivo de Python. Para esto ejecutamos la siguiente instrucción en la raíz del proyecto:

```
$ python manage.py shell
```

Django desde el intérprete interactivo

Es posible tener acceso a nuestro proyecto Django desde el intérprete interactivo de Python. Para esto ejecutamos la siguiente instrucción en la raíz del proyecto:

```
$ python manage.py shell
```

Esto abre el intérprete interactivo habiendo cargado todos los parámetros de `settings.py`, por lo cual tendremos acceso a los modelos que hemos creado.

El API del modelo

Usemos el intérprete interactivo para jugar un poco con el API de Django:

```
>>> from libros.models import Autor, Libro  
>>> Autor.objects.all()
```

El API del modelo

Usemos el intérprete interactivo para jugar un poco con el API de Django:

```
>>> from libros.models import Autor, Libro
>>> Autor.objects.all()

[<Autor: Herman Hesse>, <Autor: Paulo Coelho>]
```

El API del modelo

Usemos el intérprete interactivo para jugar un poco con el API de Django:

```
>>> from libros.models import Autor, Libro
>>> Autor.objects.all()

[<Autor: Herman Hesse>, <Autor: Paulo Coelho>]
```

A nivel de código simplemente estamos tratando con objetos, el framework se encarga de interactuar directamente con la base de datos.

El API del modelo

Podemos acceder a los campos de los registros simplemente mediante los atributos de cada objeto:

```
>>> herman = Autor.objects.all()[0]
>>> herman.nombre
u'Herman Hesse'
```

El API del modelo

Podemos acceder a los campos de los registros simplemente mediante los atributos de cada objeto:

```
>>> herman = Autor.objects.all()[0]
>>> herman.nombre
u'Herman Hesse'
```

La `u` antes de la cadena indica que se está trabajando con una cadena “unicode”. Esto permite que las cadenas contengan caracteres especiales.

El API del modelo

Además de consultar, es posible utilizar el API completo de los modelos de Django. También podemos crear nuevos objetos y guardarlos en la base de datos:

El API del modelo

Además de consultar, es posible utilizar el API completo de los modelos de Django. También podemos crear nuevos objetos y guardarlos en la base de datos:

```
>>> serafin = Autor(nombre="Serafin Mazparrote")  
>>> serafin.save()
```

El API del modelo

Además de consultar, es posible utilizar el API completo de los modelos de Django. También podemos crear nuevos objetos y guardarlos en la base de datos:

```
>>> serafin = Autor(nombre="Serafin Mazparrote")
>>> serafin.save()

>>> lb = Libro(titulo=u"Biología de 8º grado", fecha_pub="1990-01-01",
... autor=serafin)
>>> lb.save()
```

El API del modelo

Además de consultar, es posible utilizar el API completo de los modelos de Django. También podemos crear nuevos objetos y guardarlos en la base de datos:

```
>>> serafin = Autor(nombre="Serafin Mazparrote")
>>> serafin.save()

>>> lb = Libro(titulo=u"Biología de 8º grado", fecha_pub="1990-01-01",
... autor=serafin)
>>> lb.save()

>>> lb2 = Libro(titulo="Siddhartha", fecha_pub="1922-01-01", autor=herman)
>>> lb2.save()
```

El API del modelo

rel_set

Hay distintas maneras de consultar objetos relacionados a través del API. Una de ellas es mediante el atributo `libro_set` el objeto de tipo `Autor`, éste tiene acceso a una lista con todos los libros relacionados:

```
>>> herman.libro_set.objects.all()
[<Libro: Demian - Herman Hesse>, <Libro: Siddhartha - Herman Hesse>]
```

El API del modelo

filter

Además de obtener todos los objetos, es posible filtrar resultados mediante el método `filter`:

```
>>> Libro.objects.filter(autor=herman)
[<Libro: Demian - Herman Hesse>, <Libro: Siddhartha - Herman Hesse>]
```

El API del modelo

get

Con el método `get` obtenemos un único resultado, esto es útil para buscar por `id`:

```
>>> Libro.objects.get(id=8)
Traceback (most recent call last):
...
DoesNotExist: Libro matching query does not exist. Lookup parameters were {'id': 8}
>>> L = Libro.objects.get(pk=1)
>>> L.titulo
'Demian'
```

El API del modelo

exclude

Otra función útil de búsqueda es `exclude`:

```
>>> Autor.objects.all()
[<Autor: Herman Hesse>, <Autor: Paulo Coelho>, <Autor: Serafin Mazparrote>]
>>> Autor.objects.exclude(nombre__contains="Paulo")
[<Autor: Herman Hesse>, <Autor: Serafin Mazparrote>]
```

QuerySets

- Cada vez que efectuamos una instrucción como `Autor.objects.all()` estamos lidiando con un *QuerySet*.

QuerySets

- Cada vez que efectuamos una instrucción como `Autor.objects.all()` estamos lidiando con un *QuerySet*.
- Un *QuerySet* representa una consulta a la base de datos.

QuerySets

- Cada vez que efectuamos una instrucción como `Autor.objects.all()` estamos lidiando con un *QuerySet*.
- Un *QuerySet* representa una consulta a la base de datos.
- Los *QuerySets* se evalúan “perezosamente”.

QuerySets

Para saber cuántos resultados nos devuelve un QuerySet, por ejemplo, podríamos hacer:

```
len(Autor.objects.all())
```

QuerySets

Para saber cuántos resultados nos devuelve un QuerySet, por ejemplo, podríamos hacer:

```
len(Autor.objects.all())
```

Sin embargo, esto ejecutará la consulta completa a la base de datos, la convertirá en una lista y entonces aplicará la función `len()`...

QuerySets

Para saber cuántos resultados nos devuelve un QuerySet, por ejemplo, podríamos hacer:

```
len(Autor.objects.all())
```

Sin embargo, esto ejecutará la consulta completa a la base de datos, la convertirá en una lista y entonces aplicará la función `len()`... En lugar de esto, la clase `QuerySet` nos provee el método `count`:

```
>>> Autor.objects.all().count()  
3
```

QuerySets

Para saber cuántos resultados nos devuelve un QuerySet, por ejemplo, podríamos hacer:

```
len(Autor.objects.all())
```

Sin embargo, esto ejecutará la consulta completa a la base de datos, la convertirá en una lista y entonces aplicará la función `len()`... En lugar de esto, la clase `QuerySet` nos provee el método `count`:

```
>>> Autor.objects.all().count()  
3
```

Esto se traduce internamente como una consulta en SQL que utiliza `SELECT COUNT(*)`, lo cual es mucho más eficiente.

QuerySets

De igual manera podemos utilizar *slicing* en los objetos de búsqueda:

```
>>> AlgunModelo.objects.all()[:50]
```

QuerySets

De igual manera podemos utilizar *slicing* en los objetos de búsqueda:

```
>>> AlgunModelo.objects.all()[:50]
```

Esto retornará otro QuerySet sin ejecutar, lo cual es más eficiente que ejecutar la consulta y luego obtener los primeros 50 registros.

QuerySets

También es posible encadenar los métodos descritos anteriormente:

QuerySets

También es posible encadenar los métodos descritos anteriormente:

```
>>> qs = Libro.objects.filter(autor__nombre__contains="Herman")
>>> qs.exclude(titulo="Demian").count()
1
```

Métodos de QuerySet

Además de `all()`, `get()`, `filter()` y `exclude()`, la clase `QuerySet` soporta los siguientes métodos:

`order_by`

```
>>> Libro.objects.all().order_by('autor')
[<Libro: Demian - Herman Hesse>, <Libro: Siddhartha - Herman Hesse>,
<Libro: Biología de 8º grado - Serafin Mazparrote>]
```

Métodos de QuerySet

Además de `all()`, `get()`, `filter()` y `exclude()`, la clase `QuerySet` soporta los siguientes métodos:

`order_by`

```
>>> Libro.objects.all().order_by('autor')
[<Libro: Demian - Herman Hesse>, <Libro: Siddhartha - Herman Hesse>,
<Libro: Biología de 8º grado - Serafin Mazparrote>]

>>> Libro.objects.all().order_by('-autor') # indica el orden inverso
[<Libro: Biología de 8º grado - Serafin Mazparrote>,
<Libro: Siddhartha - Herman Hesse>, <Libro: Demian - Herman Hesse>]
```

Métodos de QuerySet

values

El uso de `values()` nos retorna, diccionario con los resultados en lugar de objetos:

```
>>> Libro.objects.values()
[{'titulo': u'Demian',
  'genero': 1,
  u'id': 1,
  'fecha_pub': datetime.date(1919, 2, 13),
  'autor_id': 1},
 {'titulo': u'Biolog\xeda de 8\xto grado',
  'genero': 1,
  u'id': 2,
  'fecha_pub': datetime.date(1990, 1, 1),
  'autor_id': 3}]
```

Métodos de QuerySet

reverse

```
>>> AlgunModelo.objects.all().reverse()
```

Métodos de QuerySet

reverse

```
>>> AlgunModelo.objects.all().reverse()
```

distinct

```
>>> mi_queryset.all()
>>> mi_queryset.distinct()
```

Métodos de QuerySet

reverse

```
>>> AlgunModelo.objects.all().reverse()
```

distinct

```
>>> mi_queryset.all()
>>> mi_queryset.distinct()
```

first

```
>>> Libro.objects.first()
<Libro: Demian - Herman Hesse>
```


Métodos de QuerySet

last

```
>>> Autor.objects.last()  
<Autor: Serafin Mazparrote>
```

Métodos de QuerySet

last

```
>>> Autor.objects.last()  
<Autor: Serafin Mazparrote>
```

exists

Retorna True si existe un registro que cumple con los parámetros de búsqueda dados:

```
>>> Autor.objects.filter(nombre__contains="Neruda").exists()  
False
```

Métodos de QuerySet

create

Éste es un método que nos permite crear objetos sin tener que instanciar e invocar a `save()`

```
>>> coelho = Autor.objects.get(nombre__contains="Coelho")
>>> coelho.libro_set.create(titulo="El Alquimista", fecha_pub="1990-01-01")
<Libro: El Alquimista - Paulo Coelho>
>>> coelho.libro_set.create(titulo="Once minutos", fecha_pub="1990-01-01")
<Libro: Once minutos - Paulo Coelho>
```

Métodos de QuerySet

update

update permite hacer actualizaciones directas en el QuerySet, sin tener que obtener, modificar y luego invocar a `save()`:

```
>>> Libro.objects.filter(titulo__contains="Alquimista").update(fecha_pub="1990-01-01")
```

Métodos de QuerySet

update

update permite hacer actualizaciones directas en el QuerySet, sin tener que obtener, modificar y luego invocar a `save()`:

```
>>> Libro.objects.filter(titulo__contains="Alquimista").update(fecha_pub="1990-01-01")
```

delete

```
>>> Libro.objects.filter(autor=coelho).delete()
>>> Libro.objects.filter(autor=coelho).exists()
False
```

Parámetros de consulta

Para el uso de `get()`, `filter()` y `exclude()`, además de la igualdad exacta, se definen varios parámetros de búsqueda. Estos parámetros se concatenan con los atributos de búsqueda mediante doble underscore (`--`):

Parámetros de consulta

Para el uso de `get()`, `filter()` y `exclude()`, además de la igualdad exacta, se definen varios parámetros de búsqueda. Estos parámetros se concatenan con los atributos de búsqueda mediante doble underscore (`--`):

contains

```
>>> Autor.objects.filter(nombre__contains="Serafin")  
[<Autor: Serafin Mazparrote>]
```

icontains

```
>>> Autor.objects.filter(nombre_icontains="serafin")  
[<Autor: Serafin Mazparrote>]
```

Parámetros de consulta

in

Se utiliza para preguntar si el valor de un atributo específico se encuentra dentro de alguna colección:

```
>>> Libro.objects.filter(id__in=[1,2])  
[<Libro: Demian - Herman Hesse>, <Libro: Biología de 8º grado - Serafin Mazparrote>]
```


Parámetros de consulta

in

Se utiliza para preguntar si el valor de un atributo específico se encuentra dentro de alguna colección:

```
>>> Libro.objects.filter(id__in=[1,2])
[<Libro: Demian - Herman Hesse>, <Libro: Biología de 8º grado - Serafin Mazparrote>]
```

gt, gte, lt, lte

```
>>> Libro.objects.filter(fecha_pub__lte="1980-01-01")
[<Libro: Demian - Herman Hesse>, <Libro: Siddhartha - Herman Hesse>]
```

Parámetros de consulta

range

```
>>> import datetime
>>> start_date = datetime.date(1920, 1, 1)
>>> end_date = datetime.date(2013, 3, 31)
>>> Libro.objects.filter(fecha_pub__range=(start_date, end_date))
[<Libro: Biología de 8º grado - Serafin Mazparrote>,
 <Libro: Siddhartha - Herman Hesse>]
```

Parámetros de consulta

range

```
>>> import datetime
>>> start_date = datetime.date(1920, 1, 1)
>>> end_date = datetime.date(2013, 3, 31)
>>> Libro.objects.filter(fecha_pub__range=(start_date, end_date))
[<Libro: Biología de 8º grado - Serafin Mazparrote>,
<Libro: Siddhartha - Herman Hesse>]
```

year, month, day

```
>>> Libro.objects.filter(fecha_pub__year="1990")
[<Libro: Biología de 8º grado - Serafin Mazparrote>]
```

Parámetros de consulta

isnull

Para preguntar si un valor es nulo, se utiliza el parámetro `isnull`, de la siguiente manera:

```
>>> Libro.objects.filter(fecha_pub__isnull=True)  
[]
```

Vistas

Lo que en Django se conoce como “Vistas” representa una capa que comunica las solicitudes del usuario con los objetos del modelo. No determina cómo se ven los datos, sino cuáles datos son mostrados al usuario.

Vistas

Lo que en Django se conoce como “Vistas” representa una capa que comunica las solicitudes del usuario con los objetos del modelo. No determina cómo se ven los datos, sino cuáles datos son mostrados al usuario.

Para comenzar a entender las vistas, implementaremos primero un “hola mundo”. Para esto editaremos el archivo `views.py` que está dentro de la aplicación `libros`:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hola mundo!")
```

Mapeo de URLs

Ahora necesitamos crear dentro del directorio de la aplicación (libros), un archivo `urls.py` para definir una correspondencia entre un URL y la vista que acabamos de definir:

```
from django.conf.urls import patterns, url

from libros import views

urlpatterns = patterns('',
    url(r'^$', views.index, name='index')
)
```

Mapeo de URLs

Ahora necesitamos editar el archivo `urls.py` que se encuentra en el directorio principal de proyecto (`libronline/libronline/`), para dar acceso al módulo `libros`. El código debería quedar así:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^libros/', include('libros.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```


Mapeo de URLs

Ahora necesitamos editar el archivo `urls.py` que se encuentra en el directorio principal de proyecto (`libronline/libronline/`), para dar acceso al módulo `libros`. El código debería quedar así:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^libros/', include('libros.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

Guardamos entonces el archivo y ahora podemos probar iniciando nuevamente el servidor con `python manage.py runserver` e introduciendo `http://127.0.0.1:8000/libros/` en el navegador.

Mapeo de URLs

En los archivos de `urls.py` se definen expresiones regulares que describen los patrones de URL que se asociarán a determinadas vistas, o a los archivos de “mapeo” de URLs de aplicaciones específicas. De esta forma se pueden mantener organizadas de manera modular todas las rutas de las vistas del proyecto.

Mapeo de URLs

En los archivos de `urls.py` se definen expresiones regulares que describen los patrones de URL que se asociarán a determinadas vistas, o a los archivos de “mapeo” de URLs de aplicaciones específicas. De esta forma se pueden mantener organizadas de manera modular todas las rutas de las vistas del proyecto.

Para mayor información sobre la sintaxis de expresiones regulares, referirse a la documentación oficial del módulo “re” de Python.

Vistas

Ahora modificaremos `libros.views.index` para convertirla en una vista que realmente haga algo:

```
from django.shortcuts import render
from libros.models import Autor, Libro

def index(request):
    autores = Autor.objects.exclude(libro__isnull=True)
    context = {'autores_list': autores}
    return render(request, 'libros/index.html', context)

def autor_detail(request, autor_id):
    try:
        autor = Autor.objects.get(pk=autor_id)
    except Autor.DoesNotExist:
        raise Http404
    return render(request, 'libros/autor_detail.html', {'autor': autor})
```

Vistas

Ahora debemos definir el mapeo de URLs para acceder a la vista de detalle de autor. Nuestro archivo `libros/urls.py` debería lucir de esta forma:

```
from django.conf.urls import patterns, url
from libros import views

urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^autor/(?P<autor_id>\d+)/$', views.autor_detail, name='autor_detail')
)
```

Vistas

Ahora debemos definir el mapeo de URLs para acceder a la vista de detalle de autor. Nuestro archivo `libros/urls.py` debería lucir de esta forma:

```
from django.conf.urls import patterns, url
from libros import views

urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^autor/(?P<autor_id>\d+)/$', views.autor_detail, name='autor_detail')
)
```

La expresión `(?P<autor_id>\d+)` simplemente corresponde a una serie de dígitos numéricos, y asigna el valor encontrado a un argumento con el nombre “autor_id”, el cual será recibido por la vista.

Plantillas

Como podemos notar, esta vez nuestras vistas redirigen a unos templates (`index.html` y `autor_detail`) que aún no existen. Vamos a crear estas plantillas, y a configurar todo lo necesario para que funcionen.

Plantillas

Como podemos notar, esta vez nuestras vistas redirigen a unos templates (`index.html` y `autor_detail`) que aún no existen. Vamos a crear estas plantillas, y a configurar todo lo necesario para que funcionen.

Lo primero que haremos será crear una carpeta `templates` que se encuentre en la raíz del proyecto. Dentro crearemos otra carpeta con el nombre de la aplicación (`libros`):

```
$ mkdir templates
$ cd templates
$ mkdir libros
```


Plantillas

Ahora crearemos `index.html`, con el siguiente código:

```
<h1>Autores destacados</h1>
{% if autores_list %}
    <ul>
        {% for autor in autores_list %}
            <li><a href="/libros/autor/{{ autor.id }}">{{ autor.nombre }}</a></li>
        {% endfor %}
    </ul>
{% else %}
    <p>No hay autores definidos en el sistema.</p>
{% endif %}
```

Plantillas

Ahora crearemos un archivo `autor_detail.html` en el mismo directorio, con el siguiente código:

```
<h1>{{ autor.nombre }}</h1>
```

Obras escritas:

```
<ul>
{% for libro in autor.libro_set.all %}
    <li>{{ libro.titulo }} ({{ libro.fecha_publicacion|date:"Y" }})</li>
{% endfor %}
</ul>
```

```
<a href="/libros/">Volver al inicio</a>
```

Configuración

Es necesario hacer algunos ajustes para que Django encuentre nuestras plantillas. Abramos el archivo `settings.py` e insertemos al final los siguientes valores:

```
TEMPLATE_LOADERS =(
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader'
)

TEMPLATE_DIRS = (
    os.path.join(BASE_DIR, 'templates'),
)
```

Probando nuestro sitio

Si hemos hecho todo bien hasta ahora, deberíamos poder probar nuestro sitio a través de la URL `http://127.0.0.1:8000/libros/`.

Referencias útiles

- <http://www.djangoproject.com> (sitio oficial)
- <http://lightbird.net/db/index.html> (tutorial con ejemplos)
- <http://effectivedjango.com/> (guía avanzada)
- <http://docs.python.org/2/howto/regex.html> (Expresiones regulares en Python)

Muchas gracias!!