
django

Introducción

Introducción

- Django es un framework principalmente centrado en el desarrollo web escrito en Python
- Filosofía DRY (Don’t Repeat Yourself)
- Puede ser utilizado también para el desarrollo de procesos de servidor
- Desde la versión 1.5.X es compatible con Python 3
- Debe su nombre a un guitarrista de jazz de origen gitano: Django Reinhardt
- Soportado por la Django Software Foundation

Introducción

- Utilizado por empresas como:
    - Instagram (adquirida por Facebook en 2012 por 1.000M $ USD)
    - Pinterest
    - Bitbucket
    - Reddit
    - NASA
    - Disqus

Cosas que lo hacen tan maravilloso

Cosas que lo hacen tan maravilloso

- Utiliza el patrón MVC (un poco de aquella manera)
- Dispone de ORM (Object Relational Mapping)
- Mecanismos del patrón Publisher-Subscriber
- Auto-generación de código
- Interfaz web de administración (automágicamente)

Cosas que lo hacen tan maravilloso

- Proporciona un Workflow: una manera de trabajar
- Muy buena documentación
- Un gran soporte por parte de la comunidad
- Infinidad de plugins/módulos compatibles que nos solucionan muchas tareas
- Sistema de middleware que permite desarrollar una capa entre framework y aplicaciones (ideal para cachés, compresores, etc.)

Cosas que lo hacen tan maravilloso

- Protección CSRF
- Soporte de sesiones
- Internacionalización
- Sistema de diseño de URLs amigables
- Incluye un servidor web para desarrollar

¡Manos a la obra!

El entorno virtual

El entorno virtual

Frikr$ virtualenv env

Frikr$ source env/bin/activate

(env)Frikr$ pip install django

El entorno virtual

En Windows

C:/…/Frikr> env/Scripts/activate

(env)C://.../Frikr/> pip install django

Iniciar el proyecto

Iniciar el proyecto

(env)user$ django-admin.py startproject frikr

Iniciar el proyecto

En Windows

(env)C://.../Frikr/> python env/Scripts/django-admin.py startproject frikr

Estructura del proyecto

frikr/ paquete principal de nuestro proyecto

__init__.py indica que es un paquete Python

settings.py archivo de configuración del proyecto

urls.py declaración de las URLs disponibles

wsgi.py para puesta en producción con servidores

manage.py utilidades de Django. Superútil.

Arrancando el servidor de desarrollo

Iniciar el servidor de desarrollo

(env)frikr$ python manage.py runserver

(env)C:/…/frikr> python manage.py runserver

Iniciar el servidor de desarrollo

Validating models...

0 errors found

Jan 29, 1985 - 21:45:00

Django version 1.6.2, using settings 'frikr.settings'

Starting development server at http://127.0.0.1:8000/

It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Configurando Django

El archivo settings.py

Configurando Django

- DEBUG: indica si estamos o no en modo debug
- INSTALLED_APPS: aplicaciones instaladas
- DATABASES: configuración de bases de datos
- LANGUAGE_CODE: idioma por defecto
- TIME_ZONE: zona horaria por defecto
- STATIC_URL: url para servir archivos estáticos (CSS, JavaScript, Imágenes)
- MEDIA_URL: url para servir estáticos que genera la aplicación (fotos, audios, videos, etc.)
- ADMIN: Tupla de tuplas con los nombres e e-mail de los administradores. En caso de errors, se envía un e-mail con las trazas a los admin.

Ejemplo con MySQL

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': ‘<database_name>',
        'USER': ‘<database_user>',
        'PASSWORD': ‘<database_password>',
        'HOST': ‘<database_host>',
        'PORT': ‘<database_port>',
    }
}
```

Nuestra app photos

Nuestra app

Para crear una app, debemos usar el comando:

(env)frikr$ python manage.py **startapp photos**

(env)C:/…/frikr> python manage.py **startapp photos**

Estructura de una app

photos/ paquete de la app

__init__.py indica que es un paquete Python

admin.py configuración para el admin de Django

models.py descripción de los modelos

tests.py archivo para escribir los tests de la aplicación

views.py aquí escribiremos nuestros controladores

¿Controladores en views.py?

- Sí, según los creadores, es porque su enfoque es distinto al MVC tradicional

- Según ellos, Django es un framework MTV (Model-Template-View)

- En realidad, funciona como un MVC

- Podríamos llamarlo controllers.py sin problema

MVC/MVT

El patrón Modelo-Vista-Controlador
o Modelo-Vista-Template

MVC

- Es un patrón de diseño muy común en tecnologías web

- También se utiliza en el desarrollo para móviles

MVT

- En Django, llaman vistas a lo que serían los controladores (de hecho se suelen crear en un archivo views.py).

- La presentación (vista en MVC) la delegan en plantillas.

- Por tanto, es un MVC, pero llamando vista a los controladores MVC y templates a las vistas MVC.

Modelo

- Representa las entidades de nuestra aplicación
    - Usuarios
    - Mensajes entre usuarios
    - Posts de un blog
    - etc.

Vista/Template (Django)

- La vista se encarga de presentar los datos que el controlador le indica en el formato que le indica
  - Una página HTML
  - Datos en JSON
  - Datos en XML

Controlador/Vista (Django)

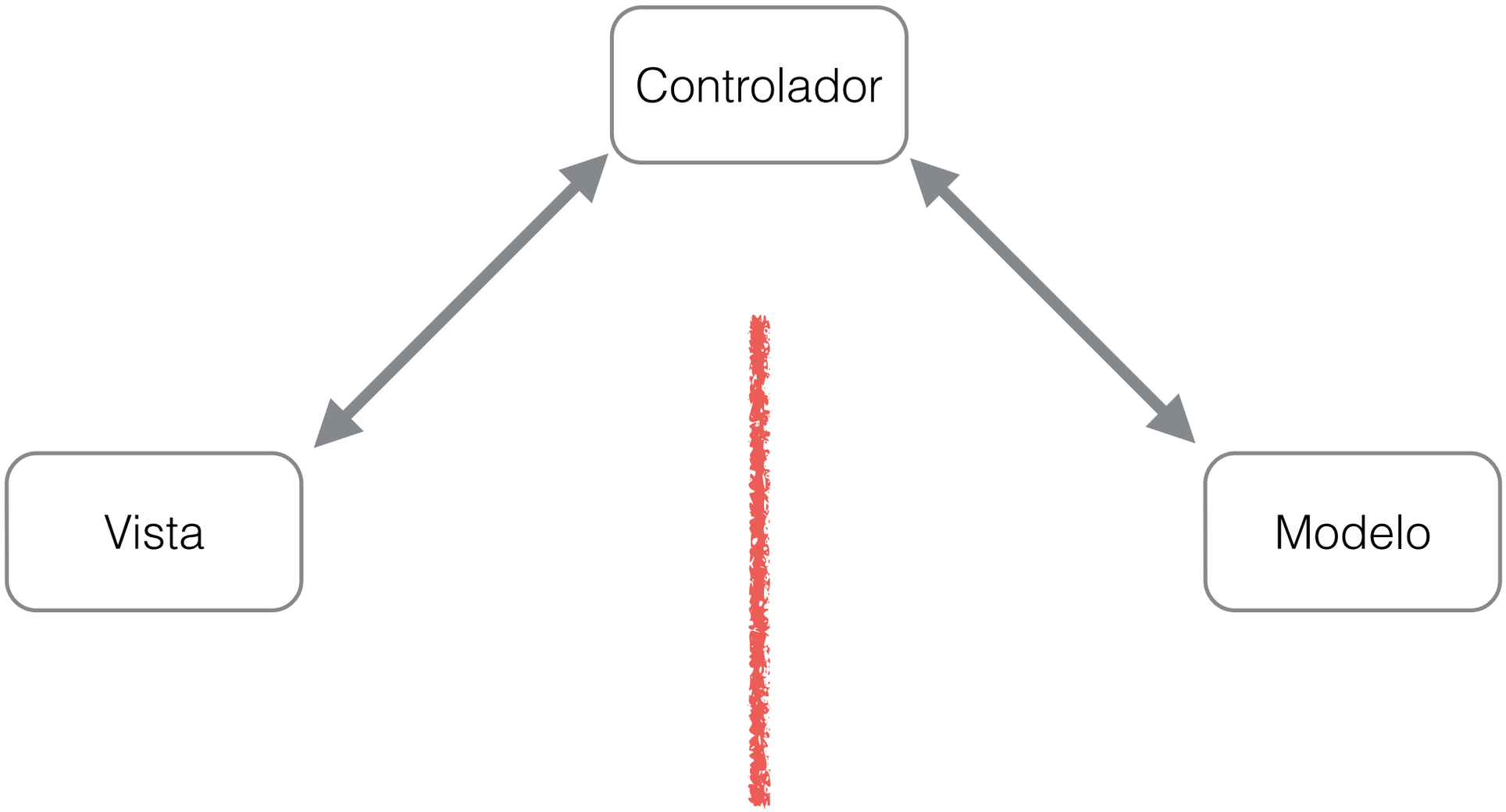
- Se encarga de de gestionar los modelos y cómo serán éstos presentados (las vistas)

- Es el nexo de unión del patrón

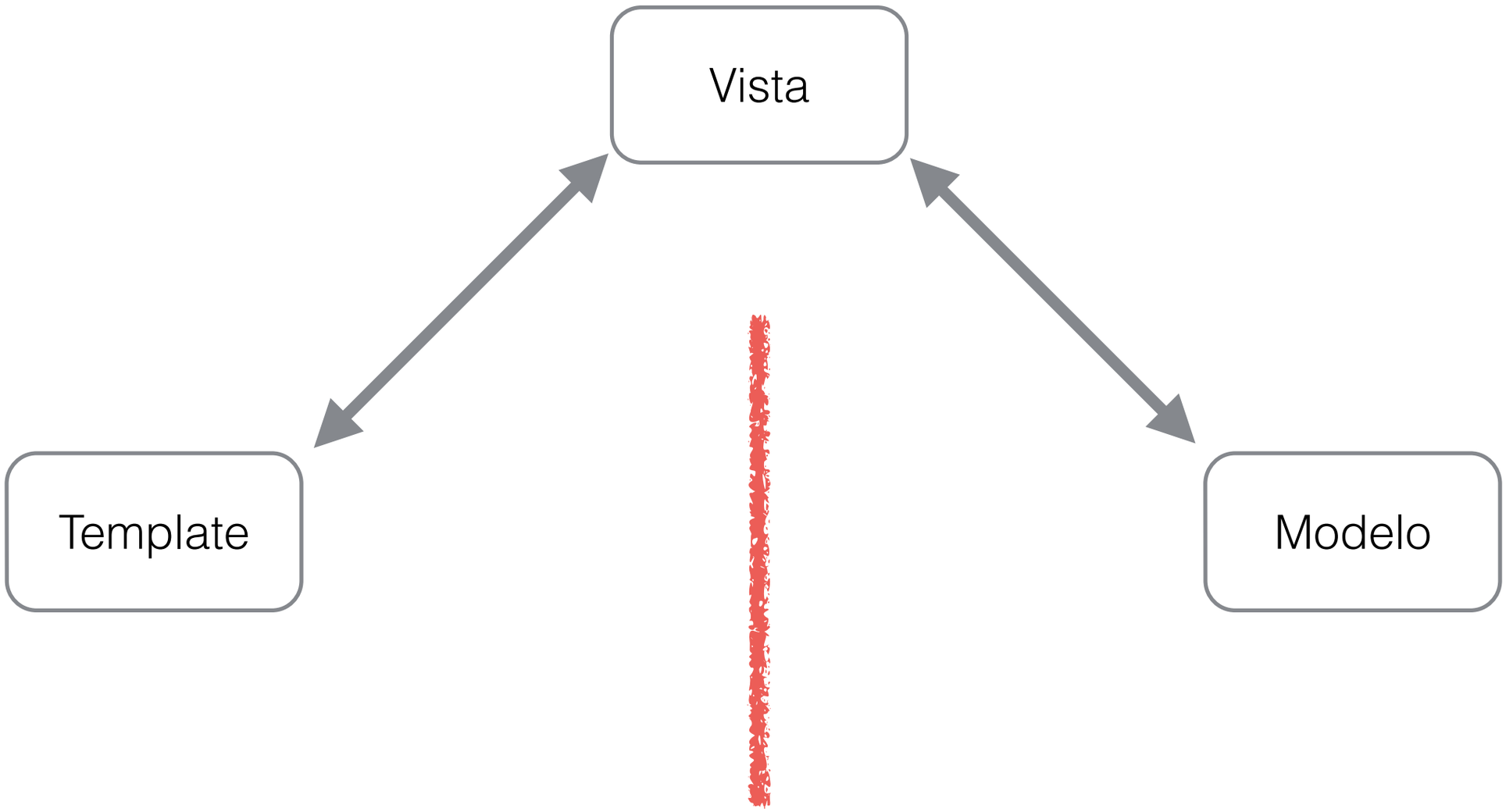
- Un modelo y una vista nunca “se tocan”

- En Django, los controladores manejan las URLs que solicita el navegador (o cliente) y se encarga de las gestiones necesarias con los modelos para presentarlos como se solicita (con la vista).

MVC



MVT



Las vistas Django

Los “controladores” de nuestra app

Las vistas Django (controladores en MVC)

- Los vistas en Django pueden ser funciones o métodos de una clase

- A través de las URL, indicamos qué controlador maneja cada URL

- Reciben siempre un objeto HttpRequest con datos de la petición HTTP: parámetros, método HTTP, datos de autenticación, etc.

- Deben devolver un objeto HttpResponse (o una subclase de HttpResponse) con el contenido (la vista)

Las vistas Django (controladores en MVC)

- Los parámetros GET podemos recuperarlos utilizando el atributo GET del objeto HttpRequest

- Los parámetros POST podemos recuperarlos utilizando el diccionario POST del objeto HttpRequest

- El objeto HttpRequest contiene también información de las cabeceras y del usuario autenticado (si existe)

El objeto HttpRequest

El objeto HttpRequest

- Este objeto lo reciben todas las vistas de Django
- Contienen información de la petición HTTP:
  - parámetros GET en el atributo *GET*
  - parámetros POST en el atributo *POST*
  - información del usuario autenticado en el atributo *user*
    - Si no está autenticado será un objeto de la clase AnonymousUser
    - Si está autenticado, será un objeto de la clase User

Los modelos

El alma de nuestra app

Los modelos

- Deben heredar de **django.db.models.Model** para ser compatibles con el motor ORM

- Los campos que definamos, deberán ser objetos derivados de **django.db.models.Field** (https://docs.djangoproject.com/en/1.6/ref/models/fields/)

- Definiendo tan sólo los campos, Django se encarga de gestionar las operaciones de bases de datos sin que tengamos que hacer nada

Los modelos

- Es recomendable crear un método **__unicode__** que devuelva una cadena unicode que describa el objeto (por ejemplo: de un usuario que devuelva su nombre y apellidos)

- Los modelos pueden relacionarse entre sí a través de campos **OneToOneField**, **OneToManyField** y **ManyToManyField**

- No hace falta que generemos un campo id, Django siempre genera uno por defecto

Los modelos

- Al heredar de la clase Modelo de Django, se implementan automáticamente los métodos *save* y *delete* que crean/actualizan y eliminan un objeto de la base de datos.

- Las clases Modelo de Django, tienen un atributo objects, el cual es un ModelManager que utilizaremos para hacer peticiones a la base de datos.

Peticiones

- Recuperar todos los objetos:
  - Model.objects.all()
- Recuperación filtrada:
  - Model.objects.filter(<campo>=<valor_filtrado>)
- Recuperación de un sólo elemento:
  - Model.object.get(<campo>=<valor_filtrado>)
  - *OJO: Esto lanza una excepción Model.DoesNotExist cuando no encuentra ningún registro y MultipleObjectsReturned cuando hay más de uno*

Validación de modelos

- Django proporciona un método de validación de modelos que se ejecuta siempre antes del guardado de un modelo en la base de datos.

- El método *clean* permite validar un modelo antes de ser guardado a través del método *save*

- También permite modificar algunos datos antes de guardar un objeto

- Las validaciones fallidas deben lanzar una excepción django.core.exceptions.ValidationError

Validación de modelos

```
class Article(models.Model):

    def clean(self):
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError('Draft entries may not have a
publication date.')
        if self.status == 'published' and self.pub_date is None:
            self.pub_date = datetime.date.today()
```

Herencia de modelos

No es buena idea

Herencia de modelos

- En general, la herencia de modelos en Django no es buena idea por razones de rendimiento

- Un modelo que hereda de otro, genera dos tablas en la base de datos, una para la clase madre y otra para la clase hija (con una relación 1-1).

- Si quieres usar herencia, puedes jugar con la herencia múltiple de Django desde clases abstractas.

Herencia de modelos

```
class CommonInfo(models.Model):

    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:

        abstract = True
```

Creando la base de datos

Con la estructura de nuestros modelos

Creando la base de datos

- El lenguaje SQL
- Claves únicas, claves foráneas
- CREATE
- SELECT
- INSERT
- UPDATE
- DELETE

Con Django, olvídate del SQL

Creando la base de datos

¡¡¡ Añadir nuestra app a las INSTALLED_APPS del settings.py !!!

Para crear la base de datos:

(env)frikr$ python manage.py **syncdb**

(env)C:/…/frikr> python manage.py **syncdb**

El admin de Django

Abracadabra, ¡un backend de la manga!

El admin de Django

- Django nos proporciona un interfaz web de administración basado en los modelos de nuestras apps

- Con un poco de código, podemos hacer que nuestras apps sean fácilmente gestionables desde una web que Django crea por nosotros

El admin de Django

- Debemos añadir **django.contrib.admin** al INSTALLED_APPS de nuestro settings.py

- Hacer el syncdb correspondiente (si procede)

- Añadir las URLs del sistema de administración a las URLs del proyecto

- Registrar nuestra aplicación al sistema de administración (en el admin.py de nuestra app)

Registrar nuestra app en el admin

- Debemos crear una clase que hereda de django.contrib.admin.ModelAdmin donde definimos cómo queremos que se comporte en el admin

- Luego debemos enlazar nuestro modelo al ModelAdmin con la función admin.site.register

Registrar nuestra app en el admin

```
# admin.py
from django.contrib import admin
from models import Planet

admin.site.register(Planet)
```


Caballeros…antes tenían mi curiosidad.
Ahora tienen mi atención.

Las plantillas

El aspecto de nuestra app

Las plantillas

- Las plantillas en Django es el código HTML (XML, JSON, YAML, etc.) que devolvemos desde un controlador

- Definen cómo se presenta la información

- En el caso de HTML, Django proporciona un potente sistema de plantillas

Las plantillas

- Estas plantillas se almacenan en una carpeta “templates” de nuestra app

- Se recomienda incluir dentro de la carpeta “templates” de cada app, una carpeta con el nombre de la app para evitar temas de espacio de nombres.

- Un proyecto puede tener su propia carpeta templates, para sobreescribir la presentación de las apps que lo componen

- El sistema de plantillas permite la reutilización (header, footer, etc.) y utiliza una sintaxis muy simple

Las plantillas

- El trabajo sucio debe hacerlo el controlador, la vista cuanto menos lógica tenga, mejor.

- Las plantillas reciben datos en un diccionario al que se puede acceder directamente a las claves

- Siempre tienen acceso a un objeto user, que identifica al usuario de la aplicación en el momento de la aplicación (cuando no está autenticado es un objeto de la clase AnonymousUser)

¿Cómo cargamos una plantilla desde el controlador?

```
from django.shortcuts import render

context = { ‘fighter_one’ : …, ‘fighter_two’ : … }

render(request, ‘<template_name>’, context)
```

Sintaxis de pantillas

Sintaxis de plantillas

- Django utiliza un sistema de plantillas para el que no se requiere conocimientos de Python.

- La idea es que los maquetadores de frontend puedan maquetar sin saber backend.

- Por defecto, escapa el HTML para evitar inyección de datos.

Impresión de datos

{{ “Mi nombre es Django” }}

Pinta la string “Mi nombre es Django”

{{ cowboy }}

Pinta el valor de la variable “cowboy”

{{ cowboy.name }}

Pinta el atributo “name” del objeto “cowboy”

Sintaxis para instrucciones

{% INSTRUCCIONES %}

Las instrucciones van entre {% y %}

Instrucciones IF

```
[…]
<body>

{% if <CONDICIONES> %}

<div> {{ something }} </div>

{% elif <OTRAS_CONDICIONES> %}

<div> {{ something_else }} </div>

{% else %}

<div> {{ a_different_thing }} </div>

{% endif %}

</body>
[…]
```

Bucles for

[...]

{% for <ITEM> IN <ITEMS_LIST> %}

 {{ <ITEM> }}

{% endfor %}

[...]

Bloques

- Los bloques permiten extender una plantilla base y reescribir diferentes partes.

- Muy útiles para webs donde tenemos que mantener un mismo header o footer, o que tengan que tener diferentes partes en común.

Bloques

base.html

```
<html>
<head>
<title> {% block title %} Forcebook {% endblock %} </title>
</head>
<body>
<header>
{% block header %} header default content {% endblock %}
</header>
<section>
{% block section %} section default content {% endblock %}
</section>
<footer>
{% block footer %} footer default content {% endblock %}
</footer>
</body>
</html>
```

Bloques

{% extends base.html %}

{% block title %} Forcebook rocks! {% endblock %}

{% block header %} Welcome to Forcebook {% endblock %}

{% block section %} Forcebook it's the most geek-based social network {% endblock %}

{% block footer %} Powered by Django {% endblock %}

Bloques

```
<html>
<head>
<title> Forcebook Rocks! </title>
</head>
<body>
<header>
Welcome to Forcebook
</header>
<section>
Forcebook it’s the most geek-based social network
</section>
<footer>
Powered by Django
</footer>
</body>
</html>
```

Incluyendo otras plantillas

[...]

{% include “menu.html” %}

 {{ main_body }}

{% include “footer_menu.html” %}

[...]

Evitando el escapado de HTML

```
[...]

{% autoescape off %}
<p> {{ django.html_description }} </p>
{% end autoescape %}

[...]
```

Filtros

{{ variable|default:”Default value” }}

Devuelve valor por defecto si variable no existe

{{ variable|length }}

Pinta el valor de la variable “cowboy”

{{ variable_with_HTML|striptags }}

Elimina los tags HTML

Django Forms

Formularios HTML fáciles

Django Forms

- En la web, el envío de formularios es algo básico e imprescindible.

- Django nos proporciona una manera de trabajar con formularios de una manera muy sencilla

- Podemos trabajar con formularios como si fueran modelos que tienen unos campos

Django Forms

```
from django import forms


class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Django Forms

- Pasándolos a una plantilla podemos imprimirlos usando la llamada:
    - form.as_p: campos en etiquetas <p> de HTML
    - form.as_table: presentado como una tabla HTML
    - form.as_ul: presentado como una <ul> HTML
- También podemos personalizar la presentación de sus campos como queramos

Django Forms

https://docs.djangoproject.com/en/1.6/ref/forms/api/

ModelForms

- Los ModelForms son formularios que se crean “automágicamente” a partir de un modelo dado

- Tan sólo tenemos que crear una clase que hereda de django.forms.ModelForm e indicar en una clase el modelo que gestiona y los campos que debe presentar

ModelForms

```
from django.forms import ModelForm

from myapp.models import Article


class ArticleForm(ModelForm):

    class Meta:

        model = Article

        fields = ['pub_date', 'headline', 'content', 'reporter']
```

Validación de formularios

- Al igual que los modelos, los formularios se validan mediante un método clean.

- Los errores de validación, deben lanzar una excepción django.forms.ValidationError

Validación de formularios

```
class ContactForm(forms.Form):

    […]

    def clean(self):

        cleaned_data = super(ContactForm, self).clean()

cc_myself = cleaned_data.get("cc_myself")

subject = cleaned_data.get("subject")

if cc_myself and subject:

    if "help" not in subject:

        raise forms.ValidationError("Did not send for 'help' in "

                "the subject despite CC'ing yourself.")
```

Vistas basadas en clases

Vistas basadas en clases

- Podemos utilizar también vistas basadas en clases

- La clase debe implementar métodos con nombres de los métodos HTTP que soporte (get, post)

Vistas basadas en clases

```
from django.views.generic import View
from django.http import HttpResponse

class PlanetList(View):
    def get(self, request, format=None):
        […]
        return HttpResponse(…)

    def post(self, request, format=None):
        […]
        return HttpResponse(…)
```

Vistas basadas en clases

En el fichero de URLs, debemos asociar la clase usando el método estático .as_view().

Ejemplo:

```
urlpatterns = patterns('',

    url(r’^planets/$', views.PlanetList.as_view()),

)
```

Generic Views

Haciendo todavía menos

Generic Views

- Es habitual en las webs tener vistas para listar entidades de un modelo o para ver el detalle de una de las entidades (Ejemplo: Flickr tiene un listado de fotos y cada foto su detalle)

- Django nos proporciona una serie de vistas genéricas que nos ayudan a implementar más rápido este tipo de vistas

- Tan sólo tenemos que heredar de ellas y configurar un par de atributos para que funcionen mágicamente

Generic Views

- TemplateView: renderiza una plantilla
- RedirectView: realiza una redirección
- DetailView: renderiza el detalle de un modelo con una plantilla
- ListView: renderiza el listado de unos modelos con una plantilla
- FormView: renderiza un formulario
- CreateView: crea un objeto basado en un modelo
- UpdateView: actualiza un objeto basado en un modelo
- DeleteView: borra un objeto basado en un modelo

Generic Views

En views.py:

```
class ArticleDetailView(DetailView):

    model = Article
    template_name = ‘article-detail.html’
```

En las urls.py:

```
urlpatterns = patterns('',

    url(r'^(?P<pk>[-_\w]+)/$', ArticleDetailView.as_view() ),

)
```

Archivos estáticos

Sirviendo desde Django CSS, JavaScript, imágenes, etc.

Archivos estáticos

- Cada app debe tener sus archivos estáticos para que puedan ser realmente reusables.

- En cada app, debemos crear una carpeta static, donde colocaremos los archivos estáticos específicos sólo de esa aplicación

- También el proyecto puede tener su propia carpeta de statics, pero tenemos que configurar que esté accesible en el settings utilizando STATICFILES_DIRS

Admin tunning

Jugando con el panel de administración

Admin tunning

- En los ModelAdmin podemos utilizar atributos para modificar la forma en la que se visualizan en el administrador.

Admin tunning

list_display = (‘field_1’, …, ‘field_N’,)

list_filter = (‘field_1’, …, ‘field_N’,)

search_fields = [‘field_1’, …, ‘field_N’]

Admin tunning

```
fieldsets = (
    (None, {
        'classes': ('wide',),
        'fields' : ('name',)
    }),
    ('System and sector', {
        'classes': ('extrapretty'),
        'fields' : (('system', 'sector'),)
    }),
    ('Coordinates', {
        'classes': ('collapse', 'wide',),
        'fields' : ('galactic_latitude', 'galactic_longitude',),
        'description' : 'Please, use intergalactic unit metrics'
    }),
)
```

Settings personalizados

Settings, settings everywhere

Settings personalizados

- En las apps, es habitual tener una serie de variables que parametrizan su comportamiento

- Estas variables, deben poder configurarse desde el settings.py del proyecto, para que se configure el comportamiento de las apps para el mismo.

Settings personalizados

```
from django.conf import settings

MY_SETTING = getattr(settings, 'MY_SETTING', ‘DEFAULT_VALUE’)

LOGIN_URL = getattr(settings, 'LOGIN_URL', ‘/login‘)
```

Usuarios con datos extra

Usuarios con datos extra

- Generalmente, el modelo de usuario no es suficiente para almacenar todos los datos que tiene un usuario

- Se suele necesitar almacenar información como la edad, el país de procedencia, etc.

- Lo primero que podemos pensar es en hacer herencia, pero como ya sabemos, la herencia en modelos no es buena idea

Usuarios con datos extra

- Heredar del usuario no es buena idea, pues el objeto que está accesible desde los objetos HttpRequest de Django es de la clase User de Django, no la que nosotros creamos personalizada

- Lo mejor es crear un modelo que actúe como información extra de usuario y relacionarlo con una clave foránea con el usuario

- De esta manera podremos acceder a los datos a través de un atributo *profile*

Usuarios con datos extra

```
from django.db import models

from django.contrib.auth.models import User

class Profile(models.Model):

    user = models.OneToOneField(User)

    age = models.IntegerField()

    city = models.CharField(max_length=“150”)
```

Usuarios con datos extra

De esta manera podremos acceder a los datos de perfil de esta usuario desde el atributo user de un objeto HttpRequest:

- request.user.profile.age
- request.user.profile.city

Enviando e-mails

Enviando e-mails

- Django nos proporciona una función para enviar e-mails fácilmente

- Podemos establecer el mecanismo de envío de e-mail desde nuestro settings.py (por ejemplo, utilizando una cuenta de Gmail)

- Incluso, podemos depurar el envío de e-mails sin tener un servidor SMTP

Enviando e-mails

```
from django.core.mail import send_mail

subject = u“Bienvenido a Django!”

message = u“Te damos la bienvenida a Django forastero!”

from_message = “django@agbotraining.com”

to_list = [“acasero@agbotraining.com”, “tarantino@django.com”]

fail_silently = True # si falla el envío, no lanza excepción error 500

send_mail(subject, html, from_message, to_list, fail_silently= fail_silently)
```

Enviando e-mails

Para configurar el envío de e-mails, podemos utilizar las siguientes variables en el settings.py del proyecto:

EMAIL_USE_TLS # True/False si tiene que usar conexión TLS

EMAIL_HOST # host con el servidor SMTP

EMAIL_HOST_USER # usuario de acceso al servidor SMTP (si aplica)

EMAIL_HOST_PASSWORD # password de acceso al servidor SMTP

EMAIL_PORT # puerto de escucha SMTP

Enviando e-mails

Si no tenemos ningún servidor SMTP desde el que probar, podemos ejecutar un servidor de escucha con Python en nuestra consola con:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Y en settings.py:

```
EMAIL_HOST = 127.0.0.1

EMAIL_PORT = 1025
```

Usando diferentes settings

Pa staging, pa producción, pa ti, pa mi

Usando diferentes settings

- Es habitual tener diferentes archivos de settings para diferentes entornos: producción, staging, desarrollo, pruebas, etc.

- Podemos tener tantos settings como queremos, sólo tenemos que indicar cual queremos utilizar al llamar a manage.py

Usando diferentes settings

python manage.py runserver —settings=frikr.settings_produccion

python manage.py syncdb —settings=frirk.settings_testing