Django

Introducción al desarrollo de aplicaciones web

Martin Riva
martin@devsar.com

Martin Alderete
martin.a@devsar.com

Sebastian Serrano
sebastian@devsar.com

Francisco Silvera
francisco@devsar.com

Quienes Somos?

- DevsAr es una empresa platense de desarrollo de software Mobile y Web con alcance global.
- Tenemos un vasto portfolio de Clientes principalmente de USA, Brasil, Europa y Argentina.
- Promovemos y utilizamos herramientas de software libre.
- Algunas de las tecnologías que más usamos son Python, Django, Ajax / JavaScript, JQuery, backbone.js, Android y tecnologías de computación en la nube como Google App Engine, Amazon Web Services y Heroku.

Contenido del Curso

- HTTP Concepts
- Introducción al Framework (Características, MVC)
- Modelos (Models)
- Vistas (Views)
- URL Dispatcher
- Plantillas (Templates)
- Formularios (Forms)
- Middlewares and Context Processors
- Configuración del Ambiente de Trabajo
- Configuración de un Proyecto Django (Settings)

Modalidad del Curso

- Clases Teórico/Prácticas
- Trabajo Práctico Final Opcional

Pre-requisitos del Curso

- Manejo básico de Linux
- Manejo de Lenguaje Python
- Conocimientos básicos de HTML/CSS
- Noción de Base de Datos

Pequeña Encuesta

Cuántos de ustedes ....

- saben Python?
- han usado Base de Datos? Cuales?
- han usado SQL?
- tienen experiencia en desarrollo app webs?
- han usado HTML/Css/JQuery?
- tienen experiencia laboral?

Algunos Links ...

- python: http://www.python.org
- django: http://www.djangoproject.com
- pip: http://www.pip-installer.org
- sqlite3 : http://www.sqlite.org
- virtualenv: http://pypi.python.org/pypi/virtualenv

Introducción al Framework

Que es Django?

- Es un framework open source hecho en python para el desarrollo ágil de aplicaciones web.

- Leimotiv - Un framework para perfeccionistas con deadlines.
- **DRY -** Do not Repeat Yourself.
- **Explícito** mejor que **Implícito** (filosofía Python)
- Enfoque basado en la Eficiencia, Seguridad, Flexibilidad y Simplicidad

Django - Características

Que nos provee el framework?

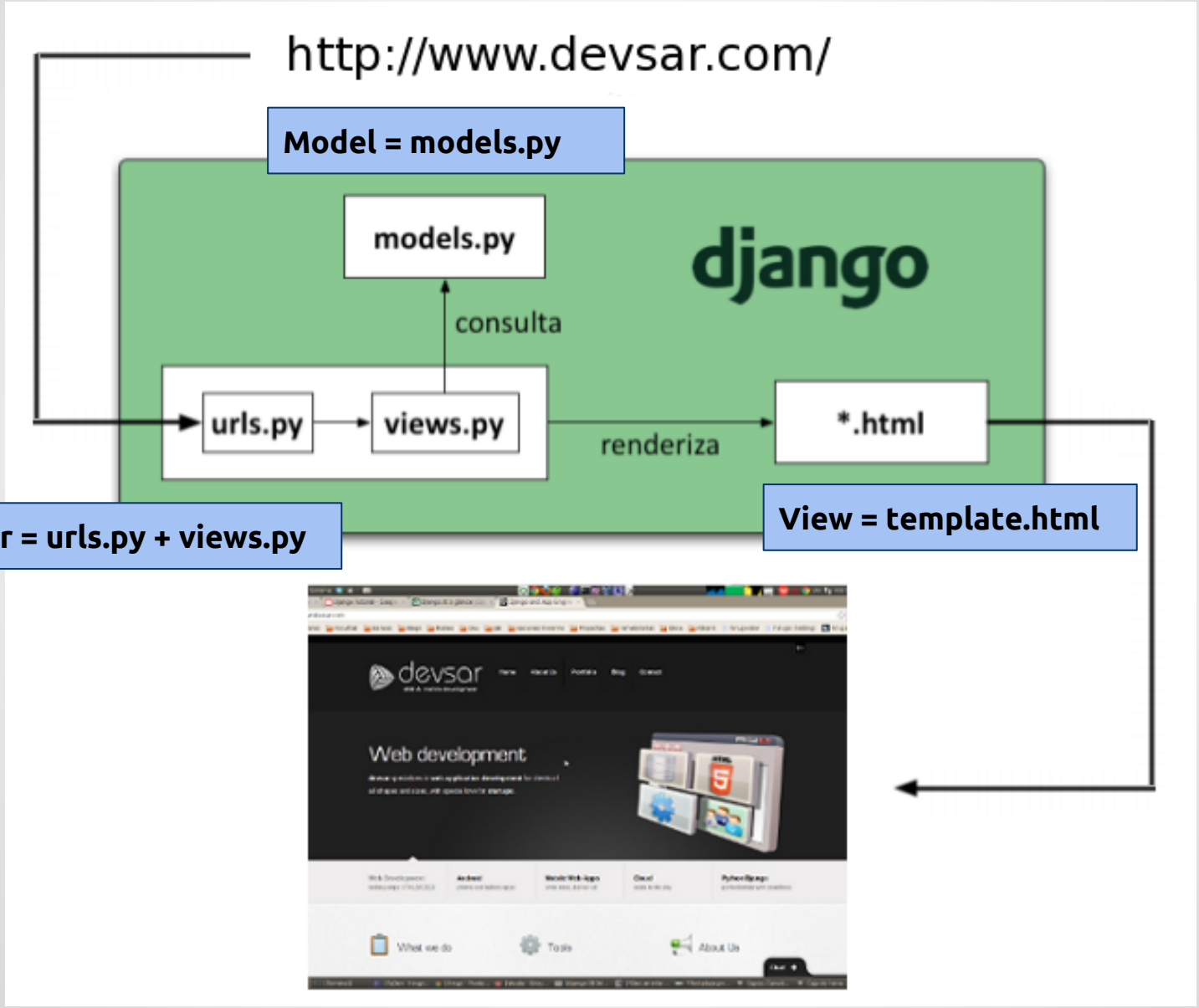
- ORM - Mapeador de Objetos-Relacional
- Motor de renderizado de Templates
- Clases para el Manejo de Formularios
- URL Mapper
- Sistema de Internacionalización
- Sistema de Autenticación
- Provee una Interfaz de Administración automática

MVC (o MTV?)

Es un patrón de arquitectura que separa los modelos de datos (models), la interfaz de usuario (views) y el manejador de lógica de negocios (controller) de manera tal de permitir el reemplazo de alguno de ellos sin necesidad de cambiar el resto. Beneficios:

- Los diseñadores pueden trabajar separadamente de los desarrolladores sin necesidad de preocupaciones respecto de cómo se modelan los datos.
- Los desarrolladores pueden abocarse a la lógica de negocios sin necesidad de preocuparse por los detalles de la presentación.

Flow


http://www.devsar.com/
Model = models.py
models.py
django
consulta
urls.py
views.py
renderiza
*.html
Controller = urls.py + views.py
View = template.html
devsar
Web development

Blog App - Django Request Handler

> http://www.miblog.com/posts/

Cuando Django recibe una petición, crea un objeto HttpRequest que la representa.

> url(r'posts/$', 'apps.blog.views.posts', name='blog_posts')

Resuelve la URL, seleccionando aquella función de la view que será responsable en la creación del response (HttpResponse).

```
> def posts(request):
>     ... do something with your models ...
>     return HttpResponse(...)
```

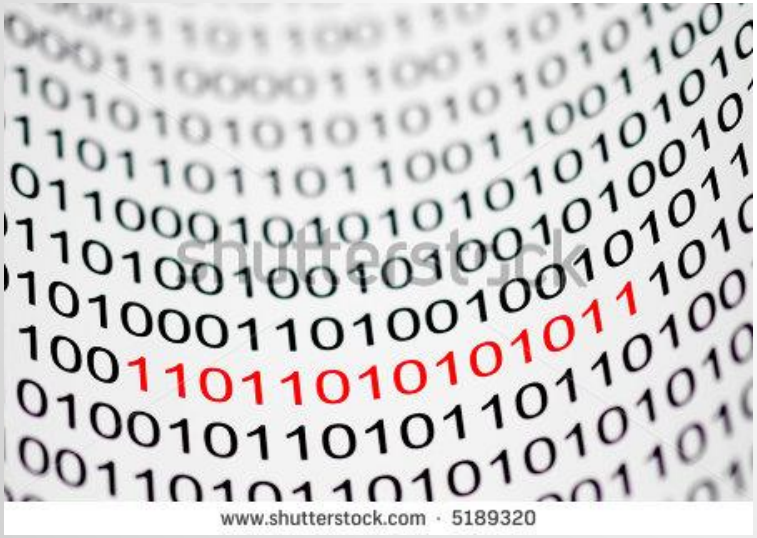
Se ejecuta la vista, que retornara el HttpResponse correspondiente...

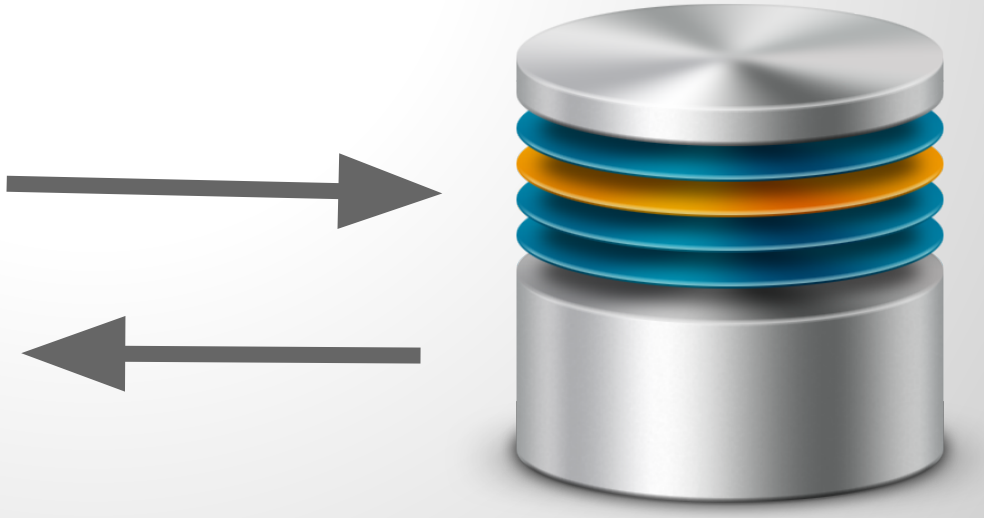
Intro a modelos: ORM

En las aplicaciones los datos generalmente van **desde y hacia** la **base de datos**.

Las aplicaciones utilizan **Orientación a Objetos.**
Las base de datos generalmente son **Relacionales.**

Tenemos un problema =(!!!





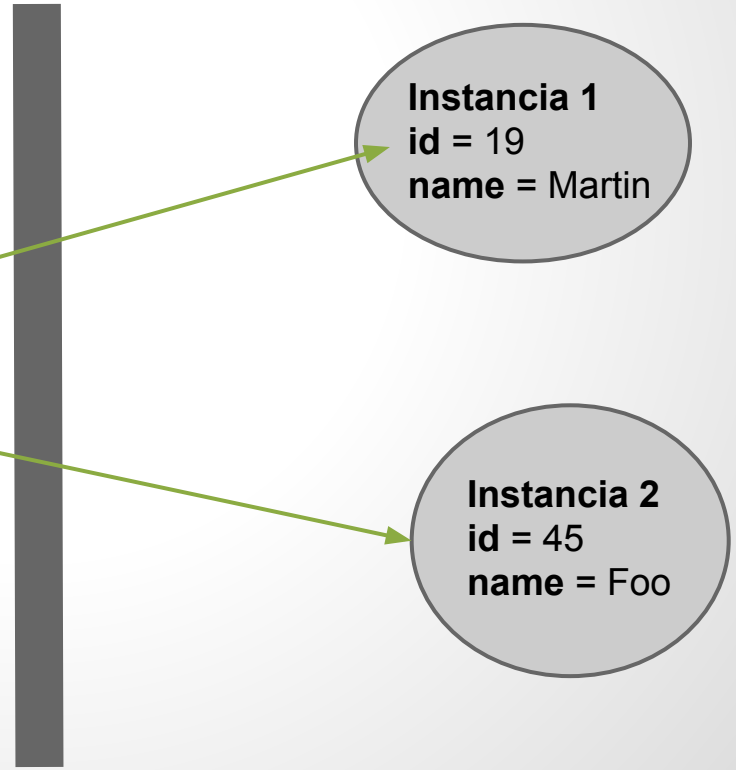
Intro a modelos: ORM

SELECT * FROM users;

Base de datos

id	name
19	Martin
45	Foo

Memoria



Modelos (Model)

- Un modelo es una descripción de los datos de nuestra BD, representada mediante código Python. Es nuestra capa de datos.
- Django utiliza los modelos para ejecutar código SQL por detrás y retornar estructuras de datos convenientes que representan cada fila de nuestra DB.
- En resumen, define los campos y el comportamiento de los datos que estamos guardando.
- Se declara mediante una clase Python que hereda de django.db.models.Model.
- Cada uno de nuestros **modelos** se mapea con una única **tabla** de nuestra base de datos.

Modelos (Model)

- Cada **atributo** de un modelo representa una **columna** de una tabla.
- El nombre de la tabla en la base de datos se derivará automáticamente del nombre del modelo y del nombre de la aplicación. Pero puede ser definido por el usuario a través de los metadatos.
- Y lo más importante ... INDEPENDIENTE del SGBD subyacente ...!!!

Modelos (Model)

Primer aproximación

```
from django.shortcuts import render
import MySQLdb

def posts_list(request):
    db = MySQLdb.connect(
             user='me', db='mydb', passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM posts ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render(request, 'posts_list.html', {'names': names})
```

Modelos (Model)

... No seria mas facil hacer algo de este estilo...

```
from django.shortcuts import render
from apps.blog.models import Post

def posts_list(request):
    posts = Post.objects.order_by('name')
    return render(request, 'posts_list.html', {'posts': posts})
```

Modelos (Model)

```
# models.py
from django.db import models
from django.contrib.auth.models import User

class Post(models.Model):
    author = models.ForeignKey(User, blank=True, null=True)
    title = models.CharField(blank=False, max_length=100)
    content = models.TextField(blank=False, max_length=40)
    datetime = models.DateTimeField(blank=True, auto_now_add=True)
    visits = models.IntegerField(default=0)
```

```
CREATE TABLE "blog_post" (
    "id" integer NOT NULL PRIMARY KEY,
    "author_id" integer NOT NULL REFERENCES "a
    "title" varchar(100) NOT NULL,
    "content" text NOT NULL,
    "datetime" datetime NOT NULL,
    "visits" integer NOT NULL
);
```

```
# settings.py

INSTALLED_APPS = (
    #...
    'apps.blog,
)
```

Cuando agregamos una nueva aplicación hay que insertar una nueva entrada en la lista de **INSTALLED_APPS** y a posteriori ejecutar el comando

>> **python manage.py syncdb**

Modelos (Model) - Field Types

- AutoField
- BigIntegerField
- BooleanField
- CharField
- DateField
- DateTimeField
- DecimalField
- EmailField
- FileField

Útil para usar cuando se requiere que el usuario suba archivos a nuestra app.

Modelos (Model) - Field Types

- FloatField
- ImageField
- IntegerField
- IPAddressField
- SlugField
- SmallIntegerField
- TextField
- TimeField
- URLField

Un CharField para la representación de urls

Modelos (Model) - Field Properties

null : Si se define en True, Django guardará valores vacíos (NULL) en la base de datos, por defecto es False.

blank : Seteado en True, Django permitirá dejar los fields en blanco en el form. El default es False.

choices : Lista de tuplas (of 2-tuples) para usar como opciones de selección en un listbox. Si se setea, Django reemplazará el widget del form por un Select con esa lista de opciones en lugar del texto por defecto.

```
class User(models.Model):
    GENDER_CHOICES = (
        ('M', 'Man'),
        ('W', 'Woman'),
        ('U', 'Unknown'),
    )
    gender = models.CharField(max_length=2, choices=GENDER_CHOICES)
```

Modelos (Model) - Field Properties

db_column : Permite definir el nombre de la columna en la base de datos. Por defecto Django usa el nombre del Field.

db_index : Si es True, django-admin.py sqlindexes <sqlindexes> creara un indice nuevo para ese Field.

default : El valor por defecto que tomara el Field. Puede ser un valor o bien una función. Si usa una función, dicha función se ejecutará cada vez que se cree un objeto nuevo.

editable : Si se setea en False, el campo no se mostrará en el admin ni en ningún otro ModelForm. Default es True.

error_messages : Esta propiedad permite redefinir el mensaje de error utilizado para ese field.

Modelos (Model) - Field Properties

help_text : Extra "help" text para ser mostrado en el widget del formulario.

primary_key : Si esta en True, este será el campo PRIMARY KEY del modelo.

Si el desarrollador no setea ningún primary key django agregara un IntegerField automáticamente (primary_key=True implica null=False & unique=True). Solo se permite un primary key por modelo.

unique : Si se setea en True, este campo deberá ser único a través de todas las instancias del modelo. Si se trata de guardar 2 objetos con un mismo valor en un campo con unique=True, django levantara la exception django.db. IntegrityError.

validators : Una lista con validaciones a ejecutar para este campo

Modelos (Model) - Relaciones

OneToOneField

```
class Car(models.Model):
  motor = OneToOneField(Motor)

class Motor(models.Model):
  serie = models.CharField()
```

Un Car tiene un y solo un Motor, un Motor pertenece a un y solo un Car

```
>>> c = Car.get(id=3)
>>> c.motor
<Motor: Motor object>
>>> m = Motor.get(id=5)
>>> m.car
<Car: Car object>
```

Modelos (Model) - Relaciones

ForeignKeyField

```
class Post(models.Model):
  title = models.CharField()

class Comment(models.Model):
  post = ForeignKeyField(Post)
```

Un Post tiene varios Comments, un Comment pertenece a un y solo un Post

```
>>> c = Comment.get(id=3)
>>> c.post
<Post: Post object>
>>> p = Post.get(id=3)
>>> p.comment_set.all() #p.comments (si defini related_name="comments")
[<Comment: Comment object>, ...]
```

Modelos (Model) - Relaciones

ManyToManyField

```
class Post(models.Model):
  tags = models.ManyToManyFields(Tag)

class Tag(models.Model):
  ...
```


```
>>> p = Post.get(id=3)
>>> p.tags.add(tag1)
>>> p.tags.add(tag2)
>>> p.tags.all()
[<Tag: Tag object>, <Tag: Tag object>]

>>> t.post_set.add(p1, p2)
>>> t.post_set.all()
[<Post: Post object>, ...]
```

Un Post tiene cero o varios Tags, y un mismo Tag puede estar presente en varios Post

Modelos (Model)

```
# models.py

import datetime
from django.db import models
from django.contrib.auth.models  import User

class Post(models.Model):
    author = models.ForeignKey(User)
    title = models.CharField(blank=False, max_length=100)
    content = models.TextField(blank=False, max_length=40)
    datetime = models.DateTimeField(blank=True,default=datetime.datetime.
now())
    visits = models.IntegerField(default=0)

class Comment(models.Model):
    author = models.ForeignKey(User)
    post = models.ForeignKey(Post)
    content = models.CharField(blank=False, max_length=100)
```

Modelos - Operaciones

- Creación

```
>> post = Post()
>> post.author = User.objects.get(id=3) #ex: Martin
>> post.title = "Comenzamos el Curso!"
>> post.content= "A partir de noviembre comienzan las clases! ..."
>> post.save()
```

- Edición / Update

```
>> post = Post.objects.get(id=1)
>> post.title = "Comenzamos el Curso de Django!"
>> post.save()
```

- Borrado

```
>> post = Post.objects.get(id=1)
>> post.delete()
```

Modelos - Operaciones (Managers)

- **Un Manager** es una interface a las **operaciones** que los modelos de Django realizan en la **base de datos**.
- **Existe al menos un Manager** por cada modelo en Django. El default se llama **“objects”**
- Se pueden agregar Manager propios a los modelos y también customizar los manager a gusto.

Como luce esto:

```
ModelClass.<manager>.manager_method()
```

```
Post.objects.all() (Llama al metodo “all” del “default manager”)
```

Utilizamos los Managers para interactuar con la base de datos

Modelos - Consultas

- Como hago para realizar consultas sobre los modelos? (**ModelClass.objects.all**)

```
>> posts = Post.objects.all()
```

- Y para obtener una instancia particular de un modelo? (**ModelClass.objects.get**)

```
>> post5 = Post.objects.get(id=5)
```

- Filtrar aquellos posts que se crearon entre d1 y d2? (**ModelClass.objects.filter**)

```
>> posts = Post.objects.filter(datetime__gt=d1, datetime__lt=d2)
```

- Lista los 10 post más visitados, ordenados por cantidad de visitas.

```
>> most_visited = Post.objects.order_by('-visits')[:10]
```

https://docs.djangoproject.com/en/1.4/topics/db/queries/#making-queries

Modelos - Consultas Relaciones

Cuando definimos relaciones (ForeignKey, OneToOneField o ManyToManyField), las **instancias de un modelo** pueden acceder a sus **relaciones**.

Django setea automáticamente un **Manager** para esto.

OneToOne

```
c = Car.objects.get(pk=1)
c.motor
```

```
m = Motor.objects.get(pk=15)
m.car
```

OneToOne se accede utilizando el atributo en ambas instancias.

https://docs.djangoproject.com/en/1.4/topics/db/queries/#related-objects

Modelos - Consultas Relaciones

Cuando definimos relaciones (ForeignKey, OneToOneField o ManyToManyField), las **instancias de un modelo** pueden acceder a sus **relaciones**.

Django setea automáticamente un **Manager** para esto.

ForeignKey

```
c = Comment.objects.get(pk=1)
c.post
```

Al revés

```
p = Post.objects.get(pk=1)
p.comment_set.all()
```

ForeignKey
El Modelo que tiene la ForeignKey utiliza el “atributo”.
Ejemplo: “c.**post**”

El otro Modelo utiliza el nombre del modelo mas “_set”.
Ejemplo: **“**p.**comment_set**.all()**”**

https://docs.djangoproject.com/en/1.4/topics/db/queries/#related-objects

Modelos - Consultas Relaciones

Cuando definimos relaciones (ForeignKey, OneToOneField o ManyToManyField), las **instancias de un modelo** pueden acceder a sus **relaciones**.

Django setea automáticamente un **Manager** para esto.

ManyToMany

```
p = Post.objects.get(pk=1)
p.tags.all()
```

Al revés

```
t = Tag.objects.get(pk=10)
t.post_set.all()
```

Similar a **ForeignKey**, cambia la nomenclatura de nombres:
El Modelo que tiene la ManyToMany utiliza el “atributo”.
Ejemplo: “p.**tags**.all()”

El otro Modelo utiliza el nombre del modelo mas “_set”.
Ejemplo: **“**t.**post_set**.all()**”**

https://docs.djangoproject.com/en/1.4/topics/db/queries/#related-objects

URLs / Controlador (Controller)

- El archivo **urls.py** actua como puerta de entrada y contiene todas las posibles peticiones que nuestra aplicación "sabe" responder.
- Toda petición a la aplicación se realiza a través de un HttpRequest.
- En pocas palabras, definimos QUE recurso se quiere ver y quien se va a encargar de manejarlo.
- Las URLs se construyen en base a expresiones regulares.
- Django se encarga de mapear cada url con la función de la vista especificada en el archivo **views.py**

URLs / Controlador (Controller)

```
# urls.py
urlpatterns = patterns('',
   (r'^articles/2003/$', 'news.views.special_case_2003'),
   (r'^articles/(?P<year>\d{4})/$', 'news.views.year_archive'),
   (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', 'news.views.month_archive'),
   (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$', 'news.views.article_detail'),
)
```

Muchas tuplas!!

(pattern, view_callback)

Ejemplos:

articles/2003/ --> **news.views.special_case_2003(request)**

articles/2010/ --> **news.views.year_archive(request, year='2010')**

articles/2013/04/ --> **news.views.month_archive(request, year='2013', month='04')**

articles/2013/04/26/ --> **news.views.article_detail(request, year='2013', month='04', day='26')**

articles/2003/04/veinte/ --> **¿Qué pasa con esto?**

URLs "reversing" --> reverse()

```
# urls.py
from django.conf.urls import url
urlpatterns = patterns('',
   url(r'^post/create/$', 'apps.blog.views.create_post',
            name='blog_create_post'),
   url(r'^post/edit/(?P<post_id>\d+)/$', 'apps.blog.views.edit_post',
            name='blog_edit_post'),
)
```

Sin tuplas! Agregamos url(....)
Atención!! ES UNA FUNCIÓN.

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

Ejemplos:

reverse('blog_create_post') ---> **'post/create/'**

reverse('blog_edit_post', kwargs={'post_id': 10}) ---> **'post/edit/10/'**

reverse('blog_crear_post') ---> **¿Qué pasa con esto?**

Vistas (Views)

- En el archivo **views.py** definiremos las funciones que se encargarán de atender los requests.
- Cada vista es, básicamente, una función que recibe como parámetro un objeto HttpRequest (y demás parámetros provenientes de la url) y devuelve **SIEMPRE** un objeto HttpResponse.
- Es en donde definimos:
  - a que modelos vamos a acceder y/o manipular
  - que template será el encargado de mostrar los datos o bien quien será el responsable de seguir atendiendo el request.

Plantillas (Templates)

- Los templates (***.html**) definen COMO vamos a mostrar nuestros datos a los usuarios.
- HTML Potenciado! HTML + tags + filters + estructuras de control + herencia + ...
- Archivos independientes y **LENGUAJE** independiente.

Un objeto **Template()** contiene el string de salida que queremos devolver en el HttpResponse (normalmente HTML), pero incluyendo etiquetas especiales de Django.

Un objeto **Context()** contiene un diccionario con los valores que dan contexto a una plantilla, los que deben usarse para renderizar un objeto **Template()**.

Plantillas (Templates)

```
# views.py

from django.http import HttpResponse
from django.template import Template, Context
from datetime import datetime

AWFUL_TEMPLATE = """
<html><body>
     Son las {{ hora }}.
</body></html>
"""

def hora_actual(request):
     now = datetime.now()
     t = Template(AWFUL_TEMPLATE)
     c = Context({'hora': now})
     html = t.render(c)
     return HttpResponse(html)
```

Ven algo raro aca?

- Código HTML dentro de un archivo .py!!!

- No nos permite desacoplar la lógica de negocio de su representación

- Los diseñadores saben HTML, no Python! Difícilmente un diseñador pueda entender como funciona esto.

- No es posible extender los templates, al menos no de forma intuitiva!

Plantillas (Templates)

```
# views.py

from django.http import HttpResponse
from django.template import Template, Context
from django.shortcuts import render_to_response
from datetime import datetime

def hora_actual(request):
  now = datetime.now()
  return render_to_response('hora.html', {'hora': now})

# hora.html
<html>
<body>
    Son las {{ hora }}.
</body>
</html>
```

Plantillas (Templates) - Tags

Inline Tags

```
{% cycle 'row1' 'row2' %}
* Se utiliza para ciclar entre N valores posibles

{% extends "base.html" %}
* Se utiliza para extender otro template

{% now "jS o\f F" %}
* Renderiza la fecha actual (7th of November)

{% include "foo/bar.html" %}
* Carga el template bar.html renderizado con el contexto actual
```

Plantillas (Templates) - Tags

Block Tags

```
{% comment %} This line will not be rendered {%  endcomment %}

{% for element in some_list %}
    <div>Este es el elemento {{ element }}</div>
{% empty %}
    <div>La lista esta vacia</div>
{% endfor %}

{% if athlete_list %} //operadores validos ==, !=, <, >, <=, >= and in
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}

{% with total=business.employees.count %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

Plantillas (Templates) - Filters

```
a_list = [1,2,3]
value = "esto es un ejemplo"
url_value = "http://www.example.org/"

{{ value|capfirst }} -> "Esto es un ejemplo"

{{ value|cut:" " }} -> "estoesunejemplo"

{{ a_date_value|date:"D d M Y" }} : Fri 07 Nov 2012

{{ value|default:"nothing" }} : Si value es "" o None "nothing"

{{ a_list|first }} -> 1

{{ value|join:" // " }} -> "esto // un // ejemplo"

{{ a_list|length }} -> 3
```

Plantillas (Templates) - Filters

```
{{ a_list|slice:":2" }} -> [1,2]

{{ value|slugify }} -> "esto-es-un-ejemplo"

{{ value|title }} -> "Esto Es Un Ejemplo"

{{ value|upper }} -> "ESTO ES UN EJEMPLO"

{{ url_value|urlencode:"" }} -> "http%3A%2F%2Fwww.example.org%2F"

{{ value|wordcount }} -> 4

{{ value|floatformat }} -> 34.26000 then 34.3

{{ value|make_list|slice:":4" }} ->  [u'e', u's', u't', u'o']
```

Plantillas (Templates) - Extends

>> base.html

```
<html>
<head>
  <title>Mi página personal</title>
</head>
<body>
  {% block content %}
  Contenido por defecto.
  {% endblock %}
</body>
</html>
```

>> home.html

```
{% extends "base.html" %}
{% block content %}
Hola desde la portada.
{% endblock %}
```

Middleware

El Middleware es un mecanismo que nos permitirá modificar e intervenir en el proceso interno del manejo de requests/responses que realiza Django. Para qué utilizamos los middlewares?

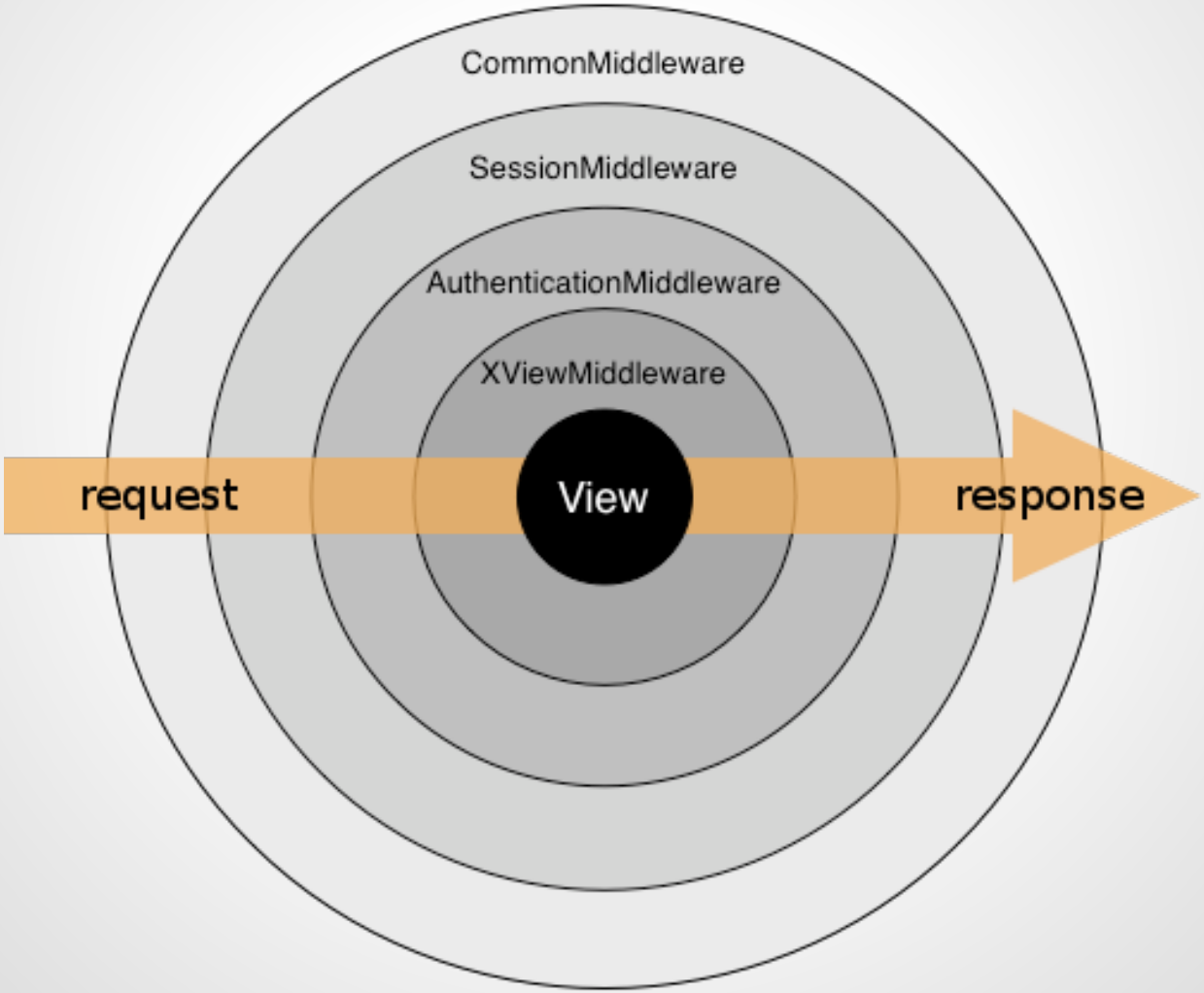
- En caso de que queramos ejecutar un mismo bloque de código antes/después de un request y/o response.
- Para mantener el estado de las sesiones y de los usuarios.
- Django mismo utiliza varios Middlewares para dar soporte a varios de los mecanismos que nos facilitan la vida.

La ejecución se realiza en dos etapas:

1) Fase Request, en donde se ejecutan los métodos:
    - process_request()
    - process_view()

2) Fase Response, se llama a los métodos:
    - process_response()
    - process_exception()

Middleware


CommonMiddleware
SessionMiddleware
AuthenticationMiddleware
XViewMiddleware
request
View
response

Middleware

Django provee varios puntos diferentes en los que permite ejecutar clases middleware, previamente definidas en el archivo de configuración (settings.py). Una misma clase puede ejecutarse en más de un punto, estas son las opciones:

- **Request middleware**: se ejecuta después de crear el objeto HttpRequest, pero antes de resolver la URL (es decidir, qué vista ejecutar), permitiendo modificar el objeto request o devolver un respuesta propia antes de que el resto de la aplicación ejecute.
- **View middleware**: es ejecutado después de la resolución de la URL, pero antes de ejecutar la vista correspondiente. Permite ejecutar operaciones antes y después de la ejecución de la vista. La vista podría llegar a no ejecutarse en absoluto.
- **Response middleware**: se ejecuta al finalizar la vista, después de que el objeto response haya sido creado y antes de entregarlo al cliente. Utilizado para realizar las modificaciones finales.
- **Exception middleware**: se ejecuta si la view lanza una exception.

Middleware

process_request() siempre retorna None o un objeto HttpResponse
None: Continúa el procesamiento del request.
HttpResponse: Se corta el pipeline y retorna dicha response.

```
# apps/core/middleware.py
class LogglyLogger(object):
    def process_request(self, request):
        # Extra logging we wonly want with non-async calls
        if not request.path.startswith('/async'):
            Loggly.log(request, tags=['middleware', 'hinge'])
        return None


# settings.py
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'apps.core.middleware.LogglyLogger',
    'apps.core.middleware.UsageTrack',
)
```

Middleware

process_response() siempre retorna un objeto HttpResponse. Puede alterar el response original, o bien retornar uno totalmente nuevo.

```
# apps/core/middleware.py
import gc
class UsageTrack(object):
    def process_response(self, request, response):
        memory = runtime.memory_usage()
        logging.info("memory_usage_current=%s" % memory.current())
        logging.info("memory_usage_average1m=%s" % memory.average1m())
        logging.info("memory_usage_average10m=%s" % memory.average10m())
        cpu_usage = runtime.cpu_usage()
        logging.info("cpu_usage_total=%s" % cpu_usage.total())
        logging.info("cpu_usage_rate1m=%s" % cpu_usage.rate1m())
        logging.info("cpu_usage_rate10m=%s" % cpu_usage.rate10m())
        if memory.current() > 70:
            gc.collect()
            memory = runtime.memory_usage()
            logging.info("memory_after_collect=%s" % memory.current())
        return response
```

Middleware

process_view() siempre retorna None or un objeto HttpResponse

```
# apps/core/middleware.py
class FacebookGathering(object):
    def process_view(self, request, view_func, view_args, view_kwargs):
        fbid = view_kwargs.get('fbid')
        if fbid:
          # Query facebook, get the profile and save it in the request
           request.fbdata = facebooklib.get_profile(fbid)
        return None


# settings.py
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'apps.core.middleware.LogglyLogger',
    'apps.core.middleware.UsageTrack',
    'apps.core.middleware.FacebookGathering',
)
```

Context Processors

- Los procesadores de contexto son funciones
- Reciben un request (HttpRequest) como parámetro y devuelven un diccionario con datos que serán accesibles desde los templates.
- Muy útiles cuando hay un dato que es utilizado en muchas vistas (DRY).

```
# my_context_processor.py
def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

# settings.py
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.contrib.messages.context_processors.messages",
    "apps.core.my_context_processor.ip_address_processor",
)
```

Context Processors

```
# views.py
from django.shortcuts import render_to_response
from django.template import RequestContext

def show_user_ip(request):
    ctx = {'name': 'Martin'}
    return render_to_response(
        'my_template.html',
        ctx,
        context_instance=RequestContext(request)
    )

# my_template.html
<html>
<body>
{{ name }}, your ip is: {{ ip_address }}
</body>
</html>
```

Forms: django.forms

Es una librería (amigable) para trabajar con formularios.

Que nos permite hacer?

- Mostrar formularios HTML con sus widgets generados automáticamente.
- Validar los datos enviados del cliente aplicando reglas de validación.
- Re-dibujar formularios en caso de contener errores de validación.
- Convertir los datos enviados del cliente en tipos de datos de Python.

Un Form encapsula una secuencia de **form.fields** y una colección de reglas que deben pasar para que el form sea aceptado.

Los forms heredan de django.forms.Form y son creados de forma **declarativa** como los **Models**.

Forms

Fields

- Representan el tipo de datos junto con las restricciones de validación.
- Tienen asociado un Widget para renderizarse.
- Realizan validaciones y retornan valores del tipo apropiado.
- Pueden ser arbitrariamente específicos o bien genéricos.

BooleanField

CharField

ChoiceField

DateField

EmailField

FileField

IntegerField

MultipleChoiceField

URLField

ModelChoiceField

Forms

Validación

```
from django import forms

> f = forms.EmailField()
> f.clean('foo@example.com')
> f.clean('invalid email')

# my_validators.py
def without_asterix(value)
  if "*" in value:
     #Mensaje customr
     raise ValidationError("Value can't contain an asterix char")

# form.py
name = forms.CharField(validators=[without_asterix])
```

Forms

Widgets

- Objetos que representan los elementos de un Formulario
- Hay uno por cada tipo de elemento de un Form: TextArea, PasswordInput
- Pueden contener incluso codigo CSS o JS para ser usado en el renderizado
- Saben como renderizar HTML

Métodos

render(self, name, value, attrs=None) -> retorna un HTML con la representacion de dicho Widget

@attrs: es un diccionario de atributos HTML

@value: no es necesariamente valido!

value_from_datadict(self, data, files, name) -> retorna el valor ingresado en el widget.

@data: diccionario con datos provenientes del POST o GET del request.

Forms

Widgets

- **TextInput** — `<input type='text' ...>`
- **PasswordInput** — `<input type='password' ...>`
- **Textarea** — `<textarea>...</textarea>`
- **Select**

```
<select>
    <option ...>...</option>
</select>
```

- **RadioSelect**

```
<ul>
  <li><input type='radio' ...></li>
  ...
</ul>
```

- **FileInput** — `<input type='file' ...>`

Forms

Error Messages

- Por defecto cada field contiene su propio mensaje de error. Sin embargo estos pueden ser customizados.

```
errors = {
  'invalid': "Este valor no es valido",
  'required': "Dale che, llena este campo!"
}

field = forms.CharField(error_messages=errors)
```

Forms

Fields Attributes

- **required**(boolean)
- **label**(string)
- **help_text**(info extra sobre los requerimientos del campo)
- **error_messages(dict)**
- **widget (forms.widgets.Widget)**

Fields para Models (relationships)

ForeignKey/ManyToMany/OneToOne => ModelChoiceField, ModelMultipleChoiceField

* Aceptan un QuerySet desde donde toman los datos para llenar el Widget

* Retornan el objeto seleccionado.

Binding Data

```
# desde datos enviados a traves de POST/GET
form = MyForm(data=request.POST)
# desde una instancia
a post = Post.objects.get(id=3)
```

ModelForm: django.forms.ModelForm

ModelForm: Helper que nos permite crear Form partiendo de un Modelo.

Es la clase base para que creemos nuestros propios ModelForms.

Son creados de forma **declarativa** como los **Models** y **los Forms**.

Por qué?

Porque muchas veces las aplicaciones están "dirigidas" por la DB.

Aplicamos la filosofía DRY y no dupliquemos la definición!

Caso clásico ABM!

```
# forms.py
from django import forms
from app.blog.models import Post

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
```

Forms

```
from django import forms

# forms.py

# Primer aproximación
class PostForm(forms.Form):
    title = forms.CharField()
    public = forms.BooleanField(widget=forms.Select)
    content = forms.CharField(widget=forms.TextArea
(rows=20, cols=30))

# Segunda aproximación usando ModelForms
class PostForm(forms.ModelForm):
    class Meta:
        model = Post
```

Forms: Uso básico

BaseForm Validation

```
form = MyForm(data=request.POST)
if form.is_valid():
        data = form.cleaned_data() #dict with GOOD data!
        model = MyModel(**data)
        model.save()
```

ModelForm Validation

```
form = MyModelForm(data=request.POST)
if form.is_valid():
        inst = form.save(commit=False)
        # do some extra logic with inst if needed
         inst.save()
```

Forms

- **Creación**

```
# views.py
def create_post(request):

    form = PostForm(request.POST or None)
    if form.is_valid(): # All validation rules pass
        post = form.save(commit=False)
        # do some extra logic with post if needed
        post.save()
        return redirect(reverse("blog_posts"))

    return render(request, "create.html", {'form': form})
```

Forms

- **Edición**

```
# views.py
def edit_post(request, post_id):

    a_post = get_object_or_404(Post, id=post_id)
    form = PostForm(request.POST or None, instance=a_post)
    if form.is_valid(): # All validation rules pass
        post = form.save()
        # do something
        post.save()
        return redirect(reverse("blog_posts"))

    return render(request, "edit.html", {'form': form})
```

Forms

- **Renderizado**

 # my_template.html

```
<h3>Create Post</h3>

<div class="form_container">
    <form method="post" action=".">{% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Create"/>
    </form>
</div>
```

Forms

- **Customización**

 # my_template.html

```
<form action="." method="post">
    {{ form.non_field_errors }}
    <div class="fieldWrapper">
        {{ form.title.errors }}
        <label for="id_title">Title:</label>
        {{ form.title }}
    </div>
    <div class="fieldWrapper">
        {{ form.content.errors }}
        <label for="id_content">Post Content:</label>
        {{ form.content }}
    </div>
    <p><input type="submit" value="Save Post" /></p>
</form>
```

Forms

- **Filtrar propiedades del modelo (Meta Class)**

```
class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        exclude = ('author','visits') #op1
        fields = ('title', 'content') #op2
```

- **Sobreescribir el widget por default**

```
class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        widgets = {
            'name': Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

Forms

- **Sobreescribir el widget de un campo**

```
class ArticleForm(ModelForm):
    pub_date = MyDateFormField()
    class Meta:
        model = Article
```

- **Sobreescribir el label de un campo**

```
class ArticleForm(ModelForm):
    pub_date = DateField(label='Publication date')
    class Meta:
        model = Article
```

Forms

- Por defecto el orden de los campos del formulario sigue el mismo orden en el que fueron definidas las propiedades en el modelo.
- Si se define el **Meta.fields**, entonces los campos del form serán renderizados en el orden en que aparecen definidos allí.
- Cuando se sobreescribe el widget de un formulario, django asume que se quiere redefinir el comportamiento y por ende no tiene en cuenta los valores seteados en el modelo.
- Al usar **Meta.fields** o **Meta.excluded**, dichos campos no serán usados para inicializar la instancia al ejecutar el método save(). Si esas propiedades son requeridas, la ejecución fallara.

Wizards

Los **Wizards** son Interfaces de usuarios que se muestran como una serie bien definida de steps (pasos).

Generalmente vemos estas clases de interfaces de usuarios en los instaladores, asistentes de configuración, etc.

Principio de **"Divide and Conquer"**

Para qué sirven?

Cuando tenemos Forms "grandes" como para usarlos en una página es mejor dividirlo en "varios" forms más chicos e ir mostrando uno a la vez.

Primero solicitar los datos más importantes y luego los menos importantes.

En el mundo de la programación web generalmente hacer esto es tedioso!!!

django.contrib.formtools

Aplicación incluida en Django para trabajar con WIZARDS, es decir, dividir forms a través de varias páginas.

Las vistas están basadas en clases.

Es una aplicación muy robusta y configurable.

Dos tipos de Backends para el Wizard:

SessionWizardView (Django Session)

CookieWizardView (Browser Cookie)

Como Usarlo:

1) Definir un **número de forms** (forms de Django) uno por página.
2) Crear una subclase de **WizardView** que posee el comportamiento.
3) Crear algun(os) template(s).
4) Agregar **django.contrib.formtools** a INSTALLED_APPS
5) Agregar una url que sea resuelta por **WizardView.as_view**

django.contrib.formtools

#views.py

```
from django.contrib.formtools.wizard.views import SessionWizardView

class ContactWizard(SessionWizardView):
        template_name = 'contact_wizard_form.html'
        def done(self, form_list, **kwargs):
                do_something_with_the_form_data(form_list)
                return HttpResponseRedirect('/page-to-redirect-to-when-done/')
```

#urls.py

```
from apps.blog.forms import ContactForm1, ContactForm2
from apps.blog.views import ContactWizard

contact_view = ContactWizard.as_view([ContactForm1, ContactForm2])

urlpatterns = patterns('',
    (r'^contact/$',  contact_view),
)
```

django.contrib.formtools

Conclusión

- Es una aplicación grande.
- Es muy personalizable.
- Hay que probarla para aprender a usarla.
- Leer la documentación.
  https://docs.djangoproject.com/en/1.4/ref/contrib/formtools/form-wizard/
- Leer el fuente (**django.contrib.formtools.wizard)**
- Template por default (**django/contrib/formtools/templates/)**
- Uso Avanzado
  https://docs.djangoproject.com/en/1.4/ref/contrib/formtools/form-wizard/#advanced-wizardview-methods

django.contrib.auth

Que contiene el paquete contrib.auth?

- User Model
- Permissions: Binarios (si/no). Indica si un usuario puede realizar tal o cual tarea.
- Groups: Una manera genérica de aplicar "etiquetas" y "permisos" a más de un usuario.
- Un sistema de hashing configurable (para el resguardo de contraseñas)
- Forms, Views, Decorators útiles para logear/deslogear usuarios o restringir acceso a ciertos contenidos

Que cosas NO vienen incluidas en este paquete?

- Chequeo de fortaleza de contraseña
- Limitación de intentos de conexión
- Autenticación con cuentas de terceros (ex: OAuth)

django.contrib.auth

Instalación

```
#settings.py
INSTALLED_APPS += (
    'django.contrib.auth',        # contains the core of the authentication
                                  # framework, and its default models.
    'django.contrib.contenttypes'  # is the Django content type system, which
                                  # allows permissions to be associated with
                                  # models you create.
)

MIDDLEWARE_CLASSES += (
    #SessionMiddleware manages sessions across requests.
    'django.contrib.sessions.middleware.SessionMiddleware',
    # AuthenticationMiddleware associates users with requests using sessions.
    'django.contrib.auth.middleware.AuthenticationMiddleware',
)
#> python manage.py syncdb : crea las tablas necesarios para los modelos de auth
```

django.contrib.auth

Las siguientes vistas YA están implementadas en django, por ende NO es necesario reescribir nada.

- django.contrib.auth.views.login
- django.contrib.auth.views.logout
- django.contrib.auth.views.password_reset
- django.contrib.auth.views.password_reset_done
- django.contrib.auth.views.password_reset_confirm
- django.contrib.auth.views.password_reset_complete
- django.contrib.auth.views.password_change
- django.contrib.auth.views.password_change_done

 # urls.py

```
url(r'^login/$', 'django.contrib.auth.views.login',
        {
          'template_name': 'profile/auth/login.html',
          #'authentication_form': MyAuthenticationForm, podríamosos cambiar el formulario
         },   name="login"),

url(r'^logout/$', 'django.contrib.auth.views.logout', {'next_page': '/'}, name="logout"),
```

django.contrib.auth

 # templates/profile/auth/login.htm

```
<div class="grid_34">
        <form action="." method="post" class="form-horizontal">
                {% csrf_token %}
                {{ form|crispy }}
                <input type="hidden" value="{{ next }}" name="next"/>
                <input type="submit" value="{% trans 'Login' %}"/>
                <a href="{% url profile_password_reset %}">{% trans "Olvidé mi password" %}</a>
        </form>
</div>
```

 # views.py

En las vistas muchas veces vamos a querer filtrar usuarios que no esten autenticados. Para esto vamos a usar el decorador **@login_required**

```
from django.contrib.auth.decorators import login_required


@login_required
def my_secret_view(request):
        render(request, "my_template.html", {})
```

Si el usuario no está autenticado este decorador redirecciona al usuario a la vista de login envía al login con esta url en el parámetro next.

django.contrib.auth

Otra forma de preguntar si un usuario está logueado es mediante el metodo **is_authenticated** que nos proveen los objetos de la clase User

 # views.py

```
def my_secret_view(request):
    if request.user.is_authenticated():
        render(request, "my_template.html", {})
```

 # my_template.html

```
{{ user.is_authenticated }}
{% if user.is_authenticated %}
```

Estructura de un Proyecto

```
<projects-folder>
      <your-project>
            ENV/
            src/
                  apps/
                        posts/
                        comments/
                  config/
                        env/
                              production.py
                        app.py
                  libs/
                        facebook.py
                  static/
                        ccs/
                        js/
                  templates/
                  app.yaml
                  cron.yaml
                  index.yaml
                  indexes.py
                  manage.py
                  settings.py
                  urls.py
```

Setup del ambiente de Trabajo

 # Instalamos un ambiente virtual

>> virtualenv ENV

Lo activamos

>> source ENV/bin/activate

 # Instalamos django

>> pip install django<1.5

>> django-admin.py --help

>> django-admin.py startproject "blog"

 # Creamos un par de aplicaciones

>> cd blog

>> django-admin.py startapp "posts"

>> django-admin.py startapp "comments"

Virtualenv

- Total Independencia de las librerías del sistema.
- Evita problemas de incompatibilidad de versiones.
- Cada proyecto tiene su propio juego de librerías con su versión correspondiente.

PIP

- Gestor integral de paquetes
- Permite especificar un archivo de dependencias/requerimientos necesarios para la ejecución de nuestra aplicación

Setup del ambiente de Trabajo

 # Iniciamos el servidor de desarrollo

>> python manage.py runserver

 # Chequeamos que todo este funcionando

browser: http://127.0.0.1:8000/

Buenas Prácticas y TIPS

En general el codigo deberia ser limpio, conciso y legible (PEP8).

1. https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/
2. http://www.python.org/dev/peps/pep-0008/

Models

- Se aconseja el uso de modelos densos/pesados. Es decir, modelos con BL (lógica de negocios) incluida.
- Evitar usar logica de negocios en las vistas (views.py) o al menos usar lo justo y necesario para atender el requerimiento.
- De ser necesario usar Managers: Esto evita código repetido y hace que nuestro código sea más testeable, legible y, no menos importante, debbugeable.
- Se recomienda el uso de South como app para la migración de schemas.

Managers

- Se recomienda el uso de Managers para evitar incurrir en consultas repetitivas y/o encapsular filtros o tendencias comunes sobre un conjunto de datos de un Modelo.

Buenas Prácticas y TIPS

Deployment

- Durante el desarrollo se recomienda el uso de ambientes aislados (ISOLATED). Esta práctica facilitará el deploy de nuestras apps y resolverá el problema de dependencias de libs de terceros. Se recomienda el uso de un ambiente por proyecto. NOTA: Ver herramientas como virtualenv, virtualenvwrapper, PIP.
- Web Server: Nginx + gunicorn + Supervisor (para mantenerlos vivos)
- Static Server: Nginx / CDN

Templating

- Los templates NO deberían contener BL, sino sólo aquella lógica necesaria para presentar los datos al usuario.

Debugging

- Utilizar django-debug-toolbar, útil para mejorar la performance de las de las vistas y la optimización de los modelos/queries.

Buenas Prácticas y TIPS

Code/Misceláneas

- Utilizar un archivo de configuración por ambiente (settings.py)
- Usar nombres cortos, obvios, preferentemente de una /dos palabras.
- Muchas pequeñas apps en vez de una única y grande.
- No reinventar la rueda! Primero verificar que el problema que tenemos que no resolver no haya sido ya resuelto por alguien más!

Third-party Apps

django-tastypie: Creaciónn de APIs Restfull

url: https://github.com/toastdriven/django-tastypie

django-debug-toolbar: Herramienta de desarrollo que permite visualizar datos acerca de cada request/response

url: https://github.com/django-debug-toolbar/django-debug-toolbar

sentry: Monitoreo de errores en tiempo real.

url: https://github.com/getsentry/sentry

celery: Sistema distribuido de procesamiento de mensajes simple, seguro y flexible. Es un sistema de colas de tareas de procesamiento en tiempo-real.

url: http://docs.celeryproject.org/en/latest/