Sistemas Web Desconectados

Release 1

van Haaster, Diego Marcos; Defossé, Nahuel

August 20, 2009

Índice general

Índice:

Parte I

Tecnologías del servidor

CAPÍTULO 1

CGI

CGI, *Common Gateway Interface* [1] es un estándar de comunicación entre un servidor web y una aplicación, que permite que un a través de un navegador, se invoque un programa en el servidor y se recuperen resultados de éste.

CGI fue la primera estandarización de un mecanismo para generar contenido dinámico en la web.

En el estandar CGI, el servidor web intercambia datos con la aplicación medinate variables de entorno y los flujos de entrada y salida.

Los parámetros HTTP (como la URL, el método (GET, POST, PUSH, etc.), nombre del servidor puerto, etc.) e información sobre el servidor son trasferidos a la aplicación CGI como variables de entorno.

Si existiese un cuerpo en la petición HTTP, como por ejemplo, el contenido de un formulario, bajo el método POST, la aplicación CGI accede a esta como entrada estandard.

El resultado de la ejecución de la aplicación CGI se escribe en la salida estándard, anteponiendo las cabeceras HTTP respuesta, para que el servidor responda al clinete. En los encabezados de respuesta, el tipo MIME determina como interpreta el clinete la respuesta. Es decir, la invocación de un CGI puede devolover diferentes tipos de contenido al clinete (html, imágenes, javascript, contenido multimedia, etc.)

Dentor de las varaibles de entorno, la Wikipedia [WikiCGI2009] menciona:

- **QUERY_STRING** Es la cadena de entrada del CGI cuando se utiliza el método GET sustituyendo algunos símbolos especiales por otros. Cada elemento se envía como una pareja Variable=Valor. Si se utiliza el método POST esta variable de entorno está vacía.
- **CONTENT_TYPE** Tipo MIME de los datos enviados al CGI mediante POST. Con GET está vacía. Un valor típico para esta variable es: Application/X-www-form-urlencoded.
- **CONTENT_LENGTH** Longitud en bytes de los datos enviados al CGI utilizando el método POST. Con GET está vacía.
- **PATH_INFO** Información adicional de la ruta (el "path") tal y como llega al servidor en el URL.
- **REQUEST_METHOD** Nombre del método (GET o POST) utilizado para invocar al CGI.
- **SCRIPT_NAME** Nombre del CGI invocado.
- **SERVER_PORT** Puerto por el que el servidor recibe la conexión.
- **SERVER_PROTOCOL** Nombre y versión del protocolo en uso. (Ej.: HTTP/1.0 o 1.1)

Variables de entorno que se intercambian de servidor a CGI:

[1] A veces traducido como pasarela común de acceso.

- **SERVER_SOFTWARE** Nombre y versión del software servidor de www.
- **SERVER_NAME** Nombre del servidor.
- **GATEWAY_INTERFACE** Nombre y versión de la interfaz de comunicación entre servidor y aplicaciones CGI/1.12

Debido a la popularidad de las aplicaciones CGI, los servidores web incluyen generalmente un directorio llamado **cgi-bin** donde se albergan estas aplicaciones.

Nota: Faltan referencias sobre la popularidad de los lenguajes

Históricamente las aplicaciones CGI han sido escritas en lenguajes interpretados, sinedo muy popular Perl y más recientemente el lenguaje PHP.

CAPÍTULO 2

Lenguajes interpreteados

2.1 PHP

2.2 Ruby

2.3 Python

Python es un lengauje interpretado. .. Escribir sobre el concepto de modulos y algo de programacion modular, ya que da soporte a protopy http://es.wikipedia.org/wiki/Programaci%C3%B3n_modular

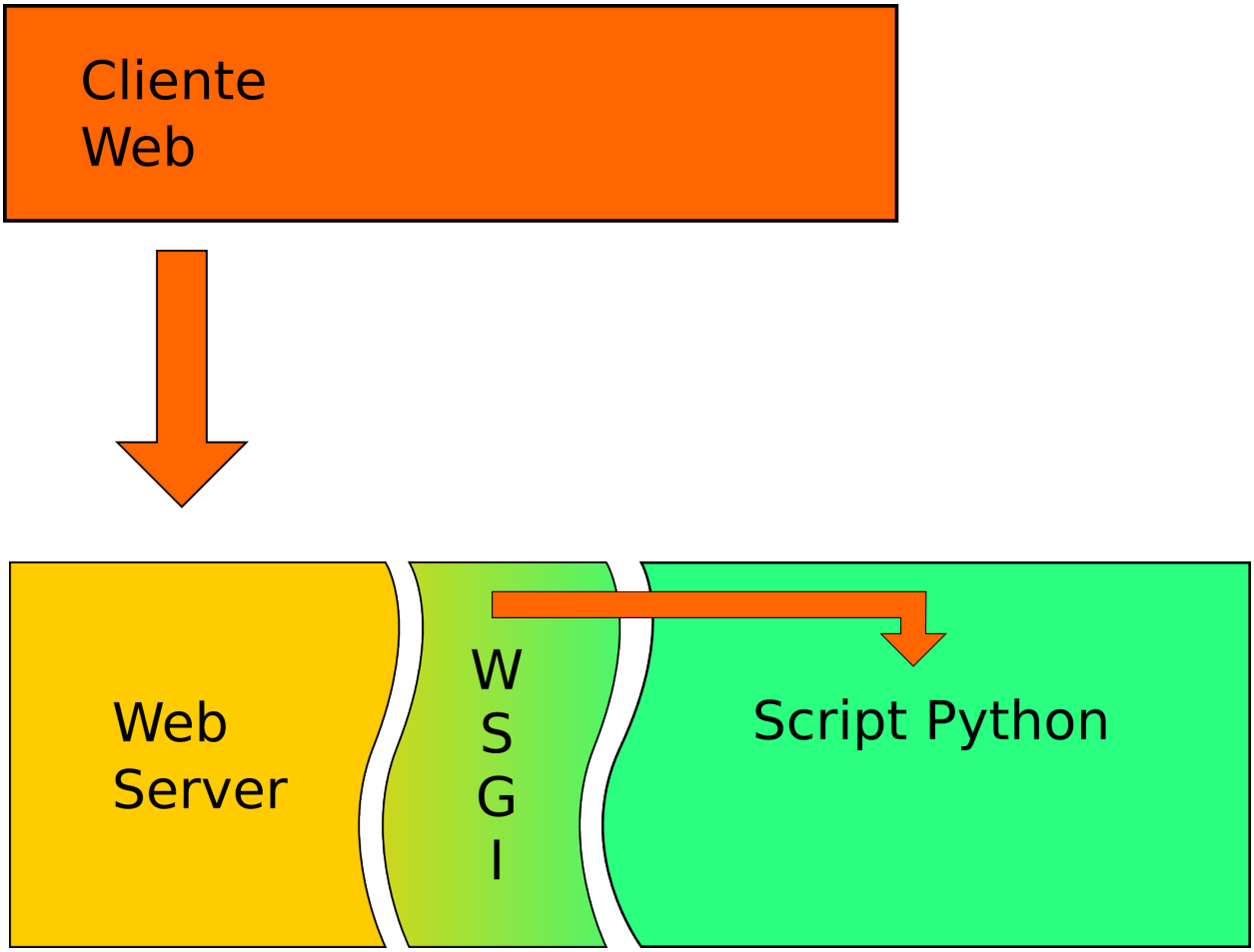
2.3.1 WSGI

WSGI o Web Server Gateway Interfase es una especidicación para que un web server y una aplicación se comuniquen. Es un estándar del lenguaje Python, descrito en el PEP [1] 333. Si bien WSGI es similar en su concepción a CGI, su objetivo es estandarizar la aparición de estructras de software cada vez más complejas (frameworks *servidor-frameworks_*)

Python, son albergados en el sitio oficial http://www.python.org

WSGI propone que una aplicación es una función que recibe 2 argumentos. Como primer argumento, un diccionario con las variables de entorno, al igual que en CGI, y como segundo argumento una función (u *python_callables_*) al cual se invoca para iniciar la respuesta.

[1] PEP *Python Enhancement Proposals* son documentos en los que se proponen mejoras para el lenguaje


Cliente
Web
Web
Server
W
S
G
I
Script Python

CAPÍTULO 3

Frameworks

Según la la wikipedia [WIK001] un framework de software es *una abstracción en la cual un código común, que provee una funcionalidad genérica, puede ser personalizado por el programador de manera selectiva para brindar una funcionalidad específica*.

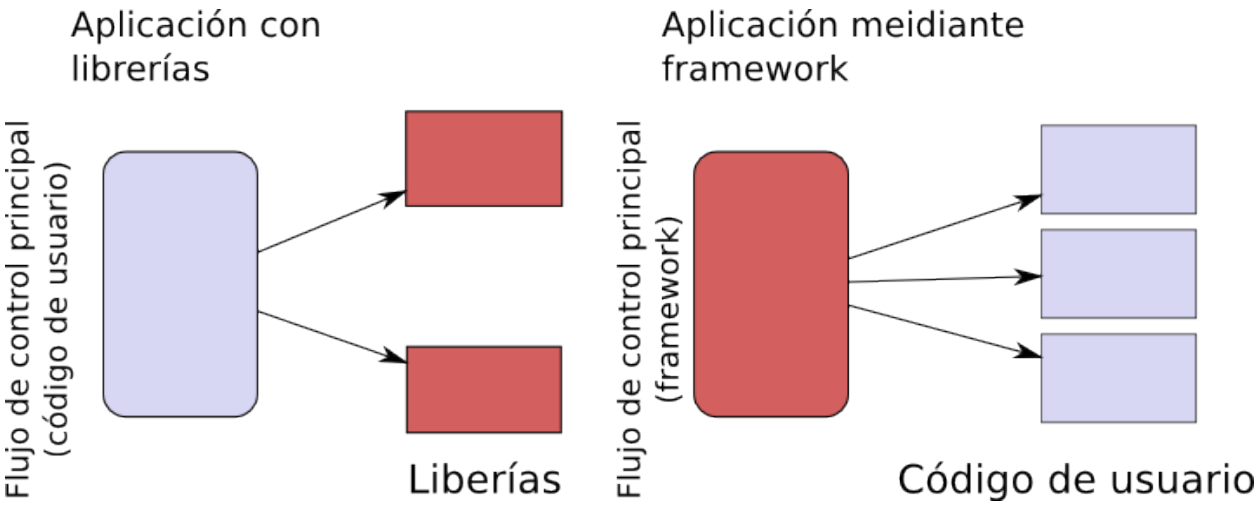
Además agrega que los frameworks son similares a las bibliotecas de software (a veces llamadas librerías) dado que proveen abstracciones reusables de código a las cuales se accede mediante una API bien definida. Sin embargo, existen ciertas características que diferencian al framework de una librería o aplicaciones normales de usuario:

- **Inversion de control** Al contrario que las bibliotecas en las aplicaciones de usuario, en un framework, el flujo de control no es manejado por el llamador, sino por el framework. Es decir, cuando se utilizan bibliotecas o programas de usuario como soporte para brindar funcionalidad, estas son llamados o invocados en el código de aplicación principal que es definido por el usuario. En un framework, el flujo de control principal está definido por el framework.
- **Comportamiento por defecto definido** Un framework tiene un comportamiento por defecto definido. En cada componete del framework, existe un comportamineto genérico con alguna utilidad, que puede ser redefinido con funcionalidad del usuario.
- **Extensibiliad** Un framework suele ser extendido por el usuario mediante redefinición o especialización para proveer una funcionalidad específica.
- **No modificabilidad del código del framework** En general no se permite la modificación del código del framework. Los programadores pueden extender el framework, pero no modificar su código.

Los diseñadores de frameworks tienen como objetivo facilitar el desarrollo de software, permitiendo a los programadores enfocarse en cumplimentar los requermientos del análisis y diseño, en vez de dedicar tiempo a resolver los detalles comunes de bajo nivel. En general la utilización de un framework reduce el tiempo de desarrollo.

Por ejemplo, en un equipo donde se utiliza un framework web para desarrollar un sitio de banca electrónica, los desarrolladores pueden enfocarse en la lógica necesaria para realizar las extraciones de dinero, en vez de la mecánica para preserver el estado entre las peticiones del navegador.

Sin embargo, se suele argumentar que los frameworks pueden ser una carga, debido a la complejidad de sus APIs o la incertidumbre que generar la existencia de varios frameworks para un mismo tipo de aplicación. A pesar de tener como objetivo estandarizar y reducir el tiempo de desarrollo, el aprendizaje de un framework suele requerir tiempo extra en el desarrollo, que debe ser tenido en cuenta por el quipo de desarrollo. Trás completar el desarrollo en un framework, el equipo de desarrllo no debe volver a invertir tiempo en aprendizaje en sucesivos desarrollos.



3.1 Framework Web

Nota: Ver diferencia entre sitio y aplicación

Un framework web, es un framework de software que permite implementar aplicaciones web brindando soporte para tareas comunes como.

En Wikipeida [WIKI002]

- Seguridad
- Mapeo de URLs
- Sistema de plantillas
- Caché
- AJAX
- Configuración mínima y simplificada

CAPÍTULO 4

Model View Controler

Antes de profundizar en más código, tomémonos un momento para considerar el diseño global de una aplicación Web Django impulsada por bases de datos.

Como mencionamos en los capítulos anteriores, Django fue diseñado para promover el acoplamiento débil y la estricta separación entre las piezas de una aplicación. Si sigues esta filosofía, es fácil hacer cambios en un lugar particular de la aplicación sin afectar otras piezas. En las funciones de vista, por ejemplo, discutimos la importancia de separar la lógica de negocios de la lógica de presentación usando un sistema de plantillas. Con la capa de la base de datos, aplicamos esa misma filosofía para el acceso lógico a los datos.

Estas tres piezas juntas – la lógica de acceso a la base de datos, la lógica de negocios, y la lógica de presentación – comprenden un concepto que a veces es llamado el patrón de arquitectura de software *Modelo-Vista-Controlador* (MVC). En este patrón, el "Modelo" hace referencia al acceso a la capa de datos, la "Vista" se refiere a la parte del sistema que selecciona qué mostrar y cómo mostrarlo, y el "Controlador" implica la parte del sistema que decide qué vista usar, dependiendo de la entrada del usuario, accediendo al modelo si es necesario.

Django sigue el patrón MVC tan al pie de la letra que puede ser llamado un framework MVC. Someramente, la M, V y C se separan en Django de la siguiente manera:

- *M*, la porción de acceso a la base de datos, es manejada por la capa de la base de datos de Django, la cual describiremos en este capítulo.
- *V*, la porción que selecciona qué datos mostrar y cómo mostrarlos, es manejada por la vista y las plantillas.
- *C*, la porción que delega a la vista dependiendo de la entrada del usuario, es manejada por el framework mismo siguiendo tu URLconf y llamando a la función apropiada de Python para la URL obtenida.

Debido a que la "C" es manejada por el mismo framework y la parte más emocionante se produce en los modelos, las plantillas y las vistas, Django es conocido como un *Framework MTV*. En el patrón de diseño MTV,

- *M* significa "Model" (Modelo), la capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.
- *T* significa "Template" (Plantilla), la capa de presentación. Esta capa contiene las decisiones relacionadas a la presentación: como algunas cosas son mostradas sobre una página web o otro tipo de documento.
- *V* significa "View" (Vista), la capa de la lógica de negocios. Esta capa contiene la lógica que accede al modelo y la delega a la plantilla apropiada: puedes pensar en esto como un puente entre el modelos y las plantillas.

Si estás familiarizado con otros frameworks de desarrollo web MVC, como Ruby on Rails, quizás consideres que las vistas de Django pueden ser el "controlador" y las plantillas de Django pueden ser la "vista". Esto es una confusión desafortunada a raíz de las diferentes interpretaciones de MVC. En la interpretación de Django de MVC, la "vista"

describe los datos que son presentados al usuario; no necesariamente el *cómo* se mostrarán, pero si *cuáles* datos son presentados. En contraste, Ruby on Rails y frameworks similares sugieren que el trabajo del controlador incluya la decisión de cuales datos son presentados al usuario, mientras que la vista sea estrictamente el *cómo* serán presentados y no *cuáles*.

Ninguna de las interpretaciones es más "correcta" que otras. Lo importante es entender los conceptos subyacentes.

Mapeador Objeto-Relacional

En las aplicaciones web modernas, la lógica arbitraria a menudo implica interactuar con una base de datos. Detrás de escena, un *sitio web impulsado por una base de datos* se conecta a un servidor de base de datos, recupera algunos datos de esta, y los muestra con un formato agradable en una página web. O, del mismo modo, el sitio puede proporcionar funcionalidad que permita a los visitantes del sitio poblar la base de datos por su propia cuenta.

Muchos sitios web más complejos proporcionan alguna combinación de las dos. Amazon.com, por ejemplo, es un gran ejemplo de un sitio que maneja una base de datos. Cada página de un producto es esencialmente una consulta a la base de datos de productos de Amazon formateada en HTML, y cuando envías una opinión de cliente (*customer review*), esta es insertada en la base de datos de opiniones.

Así como en el **'Capítulo 3'_** detallamos la manera "tonta" de producir una salida con la vista (codificando *en duro*) el texto directamente dentro de la vista), hay una manera "tonta" de recuperar datos desde la base de datos en una vista. Esto es simple: sólo usa una biblioteca de Python existente para ejecutar una consulta SQL y haz algo con los resultados.

En este ejemplo de vista, usamos la biblioteca `MySQLdb` (disponible en http://www.djangoproject.com/r/python-mysql/) para conectarnos a una base de datos de MySQL, recuperar algunos registros, y alimentar con ellos una plantilla para mostrar una página web:

```
from django.shortcuts import render_to_response
import MySQLdb

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

Este enfoque funciona, pero deberían hacerse evidentes inmediatamente algunos problemas:

- Estamos codificando *en duro* (*hard-coding*) los parámetros de la conexión a la base de datos. Lo ideal sería que esos parámetros se guardarsen en la configuración de Django.
- Tenemos que escribir una cantidad de código estereotípico: crear una conexión, un cursor, ejecutar una sentencia, y cerrar la conexión. Lo ideal sería que todo lo que tuviéramos que hacer fuera especificar los resultados que queremos.

- Nos ata a MySQL. Si, en el camino, cambiamos de MySQL a PostgreSQL, tenemos que usar un adaptador de base de datos diferente (por ej. `psycopg` en vez de `MySQLdb`), alterar los parámetros de conexión y – dependiendo de la naturaleza de las sentencia de SQL – posiblemente reescribir el SQL. La idea es que el servidor de base de datos que usemos esté abstraído, entonces el pasarnos a otro servidor podría significar realizar un cambio en un único lugar.

Rails

Symfony

CAPÍTULO 6

Django

Acá tenemos que justificar por que django

Django es un framework web escrito en Python el cual sigue vagamente el concepto de Modelo Vista Controlador. Ideado inicialmente como un adminsitrador de contenido para varios sitios de noticias, los desarrolladores encontraron que su CMS era lo sufcientemente genérico como para curbir un ámbito más aplio de aplicaciones.

En honor al músico Django Reinhart, fue liberado el código base bajo la licencia *BSD* en Julio del 2005 como Django Web Framework. El slogan del framework fue "Django, Él framework para perfeccionistas con fechas límites" [1].

En junio del 2008 fue anuncidada la cereación de la Django Software Fundation, la cual se hace cargo hasta la fecha del desarrollo y mantenimiento.

Los orígenes de Django en la administración de páginas de noticias son evidentes en su diseño, ya que proporciona una serie de características que facilitan el desarrollo rápido de páginas orientadas a contenidos. Por ejemplo, en lugar de requerir que los desarrolladores escriban controladores y vistas para las áreas de administración de la página, Django proporciona una aplicación incorporada para administrar los contenidos que puede incluirse como parte de cualquier proyecto; la aplicación administrativa permite la creación, actualización y eliminación de objetos de contenido, llevando un registro de todas las acciones realizadas sobre cada uno (sistema de logging o bitácora), y proporciona una interfaz para administrar los usuarios y los grupos de usuarios (incluyendo una asignación detallada de permisos).

Con Django también se distribuyen aplicaciones que proporcionan un sistema de comentarios, herramientas para sindicar contenido via RSS y/o Atom, "páginas planas" que permiten gestionar páginas de contenido sin necesidad de escribir controladores o vistas para esas páginas, y un sistema de redirección de URLs.

Django como framework de desarrollo consiste en un conjunto de utilidades de consola que permiten crear y manipular proyectos y aplicaciones.

6.1 Estructuración de un proyecto en Django

Durante la instalación del framework en el sistema del desarrollador, se añade al PATH un comando con el nombre django-admin.py. Mediante este comando se crean proyectos y se los administra.

Un proyecto se crea mediante la siguiente orden:

```
$ django-admin.py startproject mi_proyecto # Crea el proyecto mi_proyecto
```

Un proyecto es un paquete Python que contiene 3 módulos:

[1] Del ingles "The Web framework for perfectionists with deadlines"

- **manage.py** Interfase de consola para la ejecución de comandos
- **urls.py** Mapeo de URLs en vistas (funciones)
- **settings.py** Configuración de la base de datos, directorios de plantillas, etc.

En el ejemplo anterior, un listado gerárquico del sistema de archivos mostraría la siguiente estructura:

```
mi_proyecto/
    __init__.py
    manage.py
    settings.py
    urls.py
```

El proyecto funciona como un contenedor de aplicaciones que ser rigen bajo la misma base de datos, los mismos templates, las mismas clases de middleware entre otros parámetros.

Analicemos a continuación la función de cada uno de estos 3 módulos.

6.1.1 Módulo settings

Este módulo define la configuración del proyecto, siendo sus atributos principales la configuración de la base de datos a utilizar, la ruta en la cual se encuentran los médios estáticos, cuál es el nombre del archivo raíz de urls (generalmente urls.py). Otros atributos son las clases middleware, las rutas de los templates, el idioma para las aplicaciones que soportan *i18n*, etc.

Al ser un módulo del lengauje python, la confguración se puede editar muy facilmente a diferencia de configuraciones realizadas en XML, además de contar con la ventaja de poder configurar en caliente algunos parametros que así lo requieran.

Un parametro fundamental es la lista denominada INSTALLED_APPS que contiene los nombres de las aplicaciones instaladas en le proyecto.

6.1.2 Módulo manage

Esta es la interfase con el framework. Éste módulo es un script ejecutable, que recibe como primer argumento un nombre de comando de django.

Los comandos de django permiten entre otras cosas:

- **startapp <nombre de aplicación>** Crear una aplicación
- **runserver** Correr el proyecto en un servidor de desarrollo.
- **syncdb** Generar las tablas en la base de datos de las aplicaciones instaladas

6.1.3 Módulo urls

Este nombre de módulo aparece a nivel proyecto, pero también puede aparecer a nivel aplicación. Su misión es definir las asociaciones entre URLs y vistas, de manera que el framework sepa que vista utilizar en función de la URL que está requiriendo el clinete. Las URLs se ecriben mediante expresiones regulares del lenguaje Python. Este sistema de URLs aprovecha muy bien el modulo de expreciones regulares del lenguaje permitiendo por ejemplo recuperar gurpos nombrados (en contraposición al enfoque ordinal tradicional).

La asociación url-vistas se define en el módulo bajo el nombre *urlpatterns*. También es posible derivar el tratado de una parte de la expresión reuglar a otro módulo de urls. Generalmente esto ocurre cuando se desea delegar el tratado de las urls a una aplicación particular.

Ej: Derivar el tratado de todo lo que comience con la cadena personas a al módulo de urls de la aplicación personas.

```
(r'^personas', include('mi_proyecto.personas.urls'))
```

6.2 Mapeando URLs a Vistas

Con la estructura del proyecto así definida y las herramientas que provee Django, es posible ya ver resultados en el navegador web corriendo el servidor de desarrollo incluido en el framework para tal fin.

Es posible tambien en este momento definir modulos de vistas dentro del proyecto que otrogen determinada funcionalidad al sitio. Las vistas son invocadas por Django y deben retornar una página HTML que contenga los resultados procesados para el cliente. Lo importante de este punto es como decirle a Django que vista ejecutar ante determinada url, es en este punto donde surgen las *URLconfs*.

La *URLconf* es como una tabla de contenido para el sitio web. Básicamente, es un mapeo entre los patrones URL y las funciones de vista que deben ser llamadas por esos patrones URL. Es como decirle a Django, "Para esta URL, llama a este código, y para esta URL, llama a este otro código".

En el apartado de modulos del proyecto se observo el modulo sobre el cual el objeto URLconf es creado automáticamente: el archivo `urls.py`, este modulo tiene como requisito indispensable la definicion de la variable `urlpatterns`, la cual Django espera encontrar en el módulo `ROOT_URLCONF` definido en `settings`. Esta es la variable que define el mapeo entre las URLs y el código que manejan esas URLs.

6.3 Estructura de una aplicación Django

Una aplicación es un paquete python que consta de un módulo models y un módulo views. .. Hacer referencia al comando de startapp del modulo manager El resultado de el comando **startapp** en el ejemplo anterior genera el siguiente resultado:

```
mi_proyecto/
    mi_aplicacion/
        __init__.py
        models.py
        views.py
    __init__.py
    manage.py
    settings.py
    urls.py


mi_proyecto/
    mi_aplicacion/
        __init__.py
        models.py
        views.py
    ...
```

6.3.1 Módulo models

Cada vez que se crea una aplicación, se genera un módulo models.py, en el cual se le permite al programador definir modelos de objetos, que luego son transformados en tablas relacionales [2].

6.3.2 Módulo views

Cada aplicacion posee un módulo views, donde se definen las funcinoes que atienden al cliente y son activadas gracias a el mapeo definido en el módulo urls del proyecto o de la aplicación.

Las funciones que trabajan como vistas deben recibir como primer parámetro el request y opcionalmente parámetros que pueden ser recuperados del mapeo de urls.

Dentro del módulo de urls

```
# Tras un mapeo como el sigunete
(r'^persona/(?P<id_persona>\d)/$', mi_vista)
# la vista se define como
def mi_vista(request, id_persona):
    persona = Personas.objects.get(id = id_persona)
    datos = {'persona':persona, }
    return render_to_response('plantilla.html', datos)
```

6.4 El cliclo de una petición

Cada vez que un browser realiza una petición a un proyecto desarrollado en django, la petición HTTP pasa por varias capas.

Inicialmente atraviesa los Middlewares, en la cual, el middleware de Request, empaqueta las variables del request en una instancia de la calse Request.

Luego de atravesar los middlewares de request, mediante las defincione de URLs, se selecciona la vista a ser ejecutada.

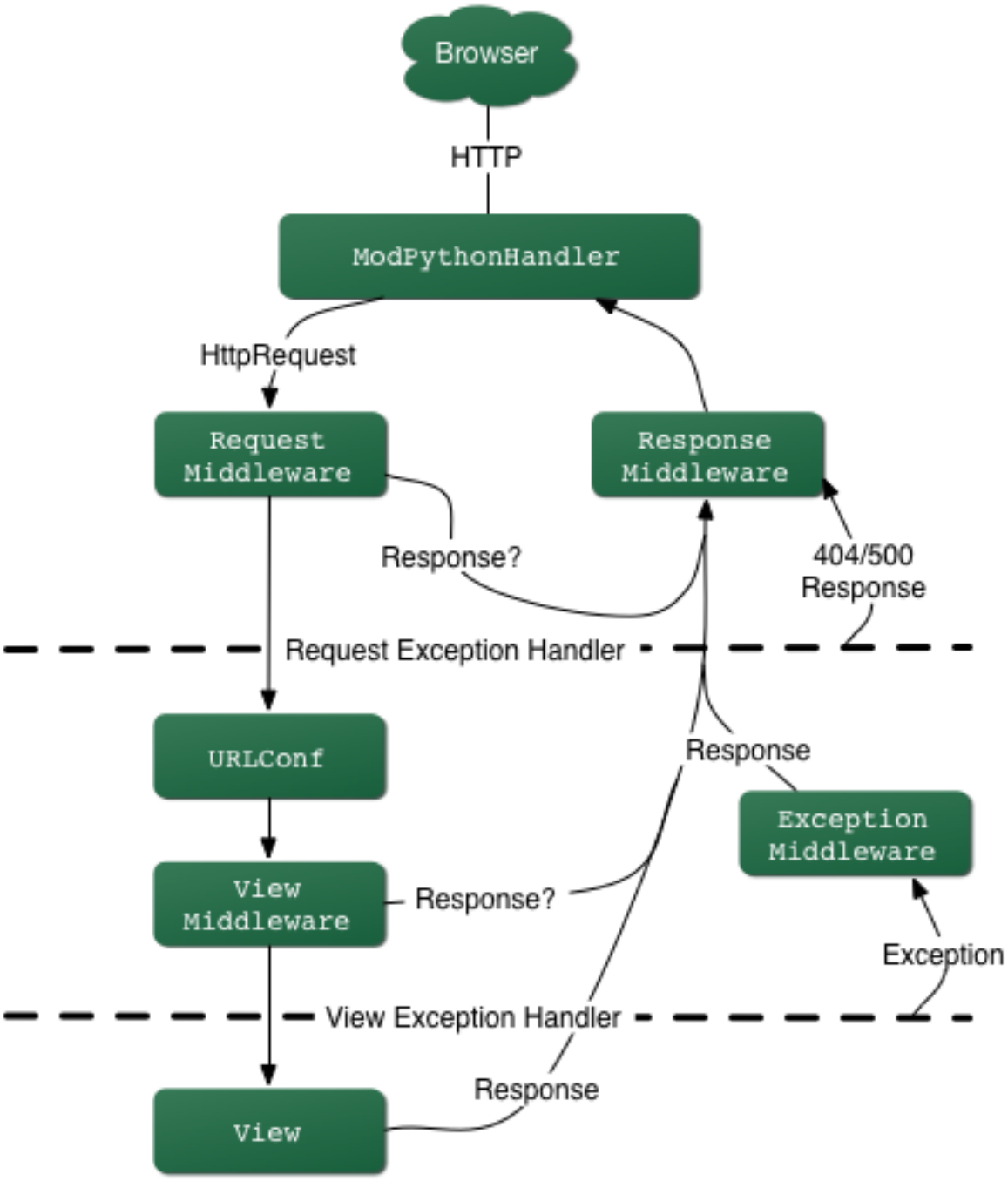
Una vista es una función que recibe como primer argumento el request y opcionalmente una serie de parámetros que puede recuperar de la propia URL.

Dentro de la vista se suelen hacer llamadas al ORM, para realizar consultas sobre la base de datos. Una vez que la vista a completado la lógica, genera un mapeo que es transferido a la capa de templates.

El template rellena sus comodines en función de los valores del mapeo que le entrega la vista. Un template puede poseer lógica muy básica (bifurcaciones, bucles de repetición, formateo de datos, etc).

El template se entrega como un HttpResponse. La responsabilidad de la vista es entregar una instancia de esta clase.

[2] Mediante el comando syncdb del módulo manage del proyecto



6.5 Interactuar con una base de datos

Django incluye una manera fácil pero poderosa de realizar consultas a bases de datos utilizando Python.

Una ves configurada la conexión a la base de datos en el módulo de configuración *Módulo settings* se esta condiciones de comenzar a usar la capa del sistema de Mapeo Objeto-Relacional del framework.

Si bien existen pocas reglas estrictas sobre cómo desarrollar dentro de Django, existe un requisito respecto a la con-

vención de la aplicación: “si se va a usar la capa de base de datos de Django (modelos), se debe crear una aplicación de Django. Los modelos deben vivir dentro de una aplicaciones”. Para crear una aplicación se debe proceder con el procedimiento ya mencionado en *Módulo manage*.

6.6 Modelos

Un modelo de Django es una descripción de los datos en la base de datos, representada como código de Python.

Esta es la capa de datos – lo equivalente a sentencias SQL – excepto que están en Python en vez de SQL, e incluye más que sólo definición de columnas de la base de datos. Django usa un modelo para ejecutar código SQL detrás de las escenas y retornar estructuras de datos convenientes en Python representando las filas de las tablas base de datos. Django también usa modelos para representar conceptos de alto nivel que no necesariamente pueden ser manejados por SQL.

Django define los modelos en Python por varias razones:

- **La introspección requiere *overhead* y es imperfecta. Django necesita** conocer la capa de la base de datos para porveer una buena API de consultas y hay dos formas de lograr esto. Una opción sería la introspección de la base de datos en tiempo de ejecución, la segunda y adoptada por Django es describir explícitamente los datos en Python.
- **Escribir Python es divertido, y dejar todo en Python limita el número de** veces que el cerebro tiene que realizar un “cambio de contexto”.
- **El codigo que describe a los modelos se puede dejar fácilmente bajo un** control de versiones.
- **SQL permite sólo un cierto nivel de metadatos y tipos de datos basicos,** mientras que un modelo puede contener tipos de datos especializado. La ventaja de un tipo de datos de alto nivel es la alta productividad y la reusabilidad de código.
- SQL es inconsistente a través de distintas plataformas.

Una contra de esta aproximación, sin embargo, es que es posible que el código Python quede fuera de sincronía respecto a lo que hay actualmente en la base. Si se hacen cambios en un modelo Django, se necesitara hacer los mismos cambios dentro de la base de datos para mantenerla consistente con el modelo.

Finalmente, Django incluye una utilidad que puede generar modelos haciendo introspección sobre una base de datos existente. Esto es útil para comenzar a trabajar rápidamente sobre datos heredados.

Este modelo de ejemplo define una `Persona` que encapsula los datos correspondientes al `nombre` y el `apellido`.

```
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length = 30)
    apellido = models.CharField(max_length = 30)
```

`nombre` y `apellido` son atributos de clase

```
CREATE TABLE miapp_persona (
    "id" serial NOT NULL PRIMARY KEY,
    "nombre" varchar(30) NOT NULL,
    "apellido" varchar(30) NOT NULL
);
```

En el ejemplo presentado se observa que un modelo es una clase Python que hereda de `django.db.models.Model` y cada atributo representa un campo requerido por el módelo de datos de la aplicación. Con esta informacion Django genera automaticamente la *API* de acceso a los datos en la base.

6.6.1 Usando la API - Consultas

Luego de crear los modelos y sincronizar la base de datos generando de esta manera el SQL correspondiente se esta en condiciones de usar la API de alto nivel en Python que Django provee para accesar los datos:

```
>>> from models import Persona
>>> p1 = Persona(nombre='Pablo', apellido='Perez')
>>> p1.save()
>>> personas = Persona.objects.all()
```

En estas lineas se ven algunos detalles de la interaccion con los modelos:

- **Para crear un objeto, se importa la clase del modelo apropiada y se crea** una instancia pasándole valores para cada campo.
- Para guardar el objeto en la base de datos, se usa el método `save()`.
- Para recuperar objetos de la base de datos, se usa `Persona.objects`.

Internamente Django traduce todas las invocaciones que afecten a los datos en secuencias `INSERT`, `UPDATE`, `DELETE` de SQL

6.6.2 Administradores de consultas

Estos objetos representan la interfase de comunicacion con la base de datos. Cada modelo tiene por lo menos un administrador para acceder a los datos almacenados.

Cada entidad presente en el modelo de una aplicación django (de aquí en adelante, simplemente modelo), tiene al menos un *Manager*. Este *Manager* encapsula en una semántica de objetos las opreaciones de consulta (*query*) de la base de datos [3] . Un *Manager* consiste en una instancia de la clase *django.db.models.manager.Manager* donde se definen, entre otros métodos, *all()*, *filter()*, *exclude()* y *get()*.

Cada uno de éstos métodos genera como resultado una instancia de la calse *QuerySet*. Un *QuerySet* envuelve el "resultado" de una consulta a la base de datos. Se dice que envuelven el "resultado" porque la estrategia de acceso a la base de datos es *evaluación retardada* [4], es decir, que la consulta que representa el *QuerySet* no será evaluada hasta que no sea necesario acceder a los resultados.

El siguiente ejemplo utiliza el manager *objects* que agrega de manera automática el ORM al modelo Usuario. En este caso, se consulta por todas las instancias de la entidad usuario.

```
Usuario.objects.all()
```

El ORM se encarga de transformar la invocación al método *all()* por el SQL siguiente.

```
SELECT * FROM aplicacion_usuarios
```

Un *QuerySet*, además de presentar la posibilidad de ser iterado, para recuperar los datos, también posee una colección de métodos orientados a conuslta, como *all()*, *filter()*, *exclude()* y *get()*. Cada uno de estos métodos, al igual que en un mánager, devuelven instancias de *QuerySet* como resultado. Gracias a esta característica recursiva, se pueden generar consultas medinate encadenamiento.

```
# datetime.now() devuelve un objeto datetime con la fecha y hora actual
r = Publicaciones.objects.filter( encabezado__stastswith = "Impuesto")
    .exclude( fecha_publicacion__lte = "2009/03/02" )
print r
```

[3] En el lenguaje SQL, las consultas se realizan miediante la sentecia SELECT.
[4] También conocida como *Lazy Evaluation*

es equivalente a: .. Esto no se si está bien

```
SELECT * FROM aplicacion_publicaciones WHERE encabezado LIKE "Impuesto%"
AND NOT fecha_publicacion >= "2009/03/02"
```

6.7 API de consultas

Dado la siguiente definición de modelo

```
class Persona(models.Model):
    nombre = models.CharField( max_length = 80 )
    apellido = models.CharField( max_length = 140 )
    fecha_de_nacimiento = models.DateField()


class Automovil(models.Model):
    ''' Un automovil posee un propietario '''

    modelo = models.IntegerField()
    marca = models.ForeignKey( Marca )

    propietario = models.ForeignKey( Persona )

class Marca(models.Model):
    nombre = models.CharField( max_length = 40 )
```

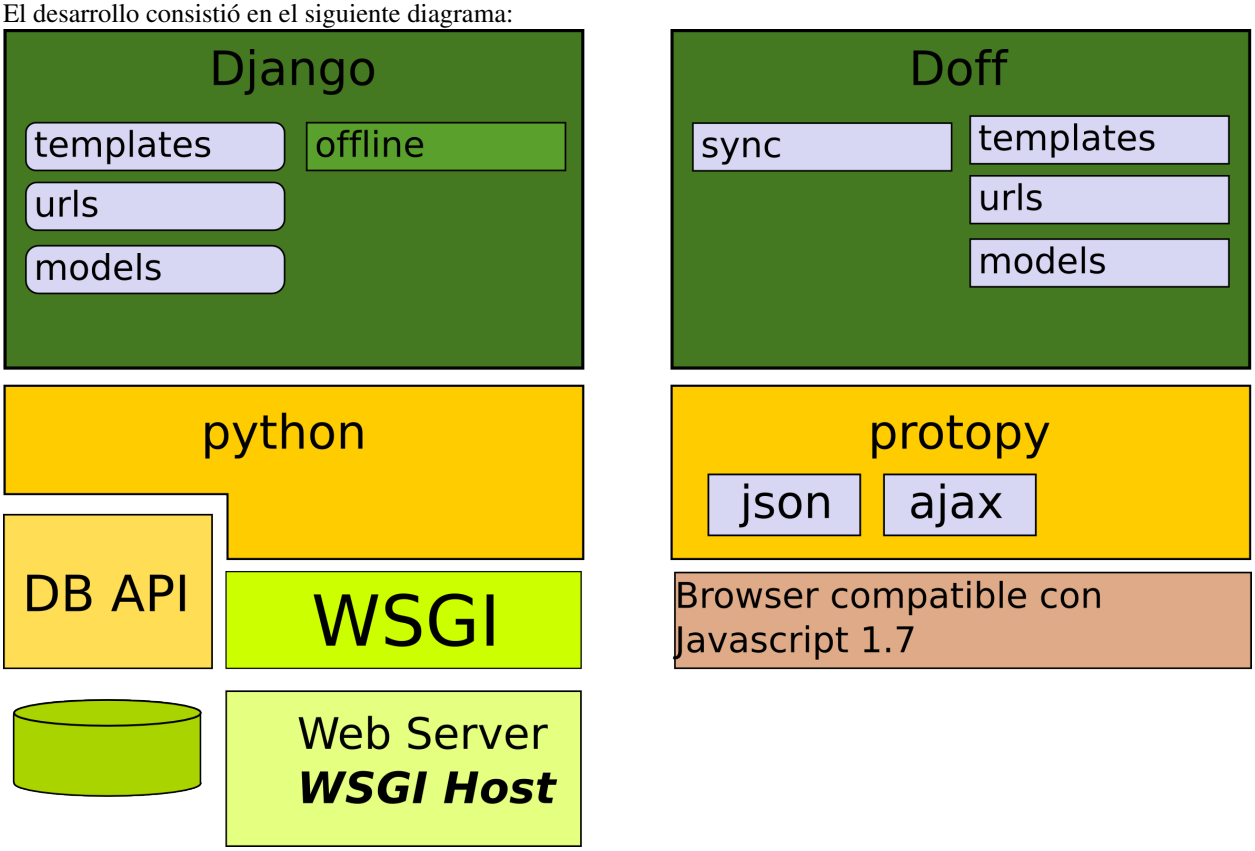
el ORM agrega a cada modelo la propiedad *objects*:

```
>>> Persona.objects
<django.db.models.manager.Manager object at 0x9e5d44c>
>>> Marca.objects
<django.db.models.manager.Manager object at 0x7efc002>
>>> Automovil.objects
<django.db.models.manager.Manager object at 0xa0edb07>
```

Parte II

Desarrollo

El desarrollo consistió en el siguiente diagrama:



Parte III

Una biblioteca en JavaScript

La idea de diseñar y desarrollar un framework en que funcione en el hambiente de un navegador web, como es Firefox, deja entrever muchos aspectos que no resultan para nada triviales al momento de codificar.

- Se requieren varias lineas de codigo para implementar un framework.
- Como llega el codigo al navegador y se inicia su ejecucion.
- La cara visible o vista debe ser fasilmente manipulable por la aplicacion de usuario.
- Como los datos generados en el cliente son informados al servidor.
- El framework debe brindar soporte a la aplicacion de usuario de una forma natural y transparente.
- Se debe promover al reuso y la extension de funcionalidad del framework.
- Como se ponen en marcha los mecanimos o acciones que la aplicacion de usuario define.

En este capitulo se introducen las ideas principales que motivaron la creacion de una libreria en JavaScript, que brinde el soporte necesario al framework y a buena parte de los items expuestos.

Si bien el desarrollo de la libreria se mantuvo en paralelo a la del framework, existen aspectos basicos a los que esta brinda soporte y permiten presentarla en un apartado separado como una ‘’Libreria JavaScript’‘, esta constituye la base para posteriores construcciones y auna herramientas que simplifican el desarrollo client-side.

proto‘type + ‘py‘thon = ‘protopy

‘‘La creación nace del caos’‘, la libreria ‘‘Protopy” no escapa a esta afirmacion e inicialmente nace de la integracion de Prototype con las primeras funciones para lograr la modularizacion; con el correr de las lineas de codigofootnote{Forma en que los informaticos miden el paso del tiempo} el desarrollo del framework torna el enfoque inicial poco sustentable, requiriendo este de funciones mas Python-compatibles se desecha la libreria base y se continua con un enfoque más ‘‘pythonico’‘, persiguiendo de esta forma acercar la semántica de JavaScript 1.7 a la del lenguaje de programacion Python.

No es arbitrario que el navegador sobre el cual corre Protopy sea Firefox y mas particularmente sobre la version 1.7 de JavaScript. El proyecto mozilla esta hacercando, con cada nueva versiones del lenguaje, la semantica de JavaScript a la de Python, incluyendo en esta version generadores e iteradores los cuales son muy bien explotados por Protopy y el framework.

CAPÍTULO 7

Protopy

Protopy es una libreria JavaScript para el desarrollo de aplicaciones web dinamicas. Aporta un enfoque modular para la inclusión de código, orientación a objetos, manejo de AJAX, DOM y eventos.

Para una referencia completa de la API de Protopy remitase al apandice *Protopy*

CAPÍTULO 8

Organizando el codigo

Como ya se vio en *cliente-javascript*, una de las formas tradicionales y preferida de incluir funcionalidad JavaScript en un documento HTML es mediante el tag `script″ haciendo una referencia en el atributo ``src` a la url del archivo que contiene el codigo que sera interpretado por el cliente web al momento de mostrar el documento.

Este enfoque resulta sustentable para pequeños proyectos donde el lenguaje brinda mayormente soporte a la interacción con el usuario (validacion, accesibilidad, etc) y los fragmentos de codigo que se pasan al cliente son bien conocidos por el desarrollador; en proyectos que implican mayor cantidad de funcionalidad JavaScript con grandes cantidades de codigo este enfoque resulta complejo de mantener y evolucionar en el tiempo.

Es por lo expuesto hasta aquí que se busco una forma de organizar y obtener el codigo que resulte sustentable y escalable; similar al concepto de ‘‘modulo” en Python *servidor-lenguajes-python*, se enfocó el desarrollo de Protopy en pequeñas unidades de codigo manejables, que puedan ser fasilmente obtenidas e interpretadas por el cliente y sumadas entre si cumplan una determinada tarea. Esta tecnica no es nueva en programacion y basicamente implica llevar el concepto de “divide y vencerás” ó “análisis descendente (Top-Down)” al hambito de JavaScript.

Un módulo en Protopy resuelve un problema especifico y define una interfaz de comunicación para accesar y utilizar la funcionalidad que contiene. Por más simple que resulte de leer, esto implica que existe una manera de **obtener** un modulo y una manera de **publicar** la funcionalidad de un modulo, logrando de esta forma que interactuen entre ellos para resolver una determinada tarea.

En su forma mas pedestre un módulo es un archivo que contiene definiciones y sentencias de JavaScript. El nombre del archivo es el nombre del módulo con el sufijo .js pegado y dentro de un módulo, el nombre del módulo (como una cadena) está disponible como el valor de la variable __name__.

8.1 Obtener un módulo

Cuando un módulo llamado spam es importado, Protopy busca un archivo llamado spam.js en la url base, de no encontrar el archivo... Otra forma de obtener modulos es usando nombres de **paquetes** asociados a urls, de forma similar a la anteriro cuando un módulo llamado paquete.spam es importado, Protopy busca en el objeto sys.paths si existe una url asociada a paquete, de encontrar la relacion el archivo spam.js es buscado en esta, por otra parte si sys.paths no contiene una url asociada el archivo paquete/spam.js es buscado en la url base de Protopy. El uso del objeto sys.paths permite a los modulos de JavaScript que saben lo que están haciendo al modificar o reemplazar el camino de búsqueda de módulos. Nótese que es importante que el script no tenga el mismo nombre que un módulo estandar. Ver el apendice *Módulos estandar* para más información.

8.2 Publicar un módulo

Un módulo puede contener sentencias ejecutables y definición de funciones también. La intención de éstas sentencias es inicializar el módulo. Ellas son ejecutadas sólo la primera vez que el módulo se importa a alguna parte. Cada módulo tiene su propia tabla de símbolos privada, que es usada como la tabla global de símbolos por todas las funciones definidas el módulo. Así, el autor de un módulo puede usar variables globales en el módulo sin preocuparse por chocar con las variables globales de un usuario.

Los módulos pueden importar otros módulos. Se acostumbra pero no se requiere poner todas las sentencias import al comienzo de un módulo (o script, para el evento). Los nombres de los módulos importados son puestos en la tabla global de símbolos del módulo importador.

Hay incluso una variante para importar todos los nombres que un módulo define:

Ésto importa todos los nombres excepto aquellos que empiecen con un underscore (__).

de codigo define un modulo, este puede ser importado por otro fragemento de codigo y cada uno representa su propio ambito de nombres. Existen en Protopy dos tipos de modulos, los modulos integrados y los modulos organizados en archivos JavaScript. %poner mas sobre los archivos js que representan modulos %Esquema de nombrado Para acceder a los modulos es necesario establecer un esquema de nombrado ldots %Completar sobre esquema de nombres. Con los modulos y un sistema de nombrado para el acceso a los mismos, la responsabilidad de la carga del codigo se deja en manos del cliente y del propio codigo que requiera determinada funcionalidad provista por un modulo.

Undo de los principales inconvenientes a los que Protopy da solucion es a la inclucion dinamica de funcionalidad bajo demanda, esto se logra con los ‘‘modulos’‘.

Las funciones principales para trabajar con los modulos son ‘‘require” para cargar un modulo en el hambito de nombres local y ‘‘publish” para que los modulos publiquen o expongan la funcionalidad.

Creando tipos de objeto

%Semántica de objetos, dentro de la cual se hace una adaptación de En la programación basada en prototipos las ‘‘clases” no están presentes, y la re-utilización de procesos se obtiene a través de la clonación de objetos ya existentes. Protopy agrega el concepto de clases al desarrollo, mediante un contructor de ‘‘tipos de objeto’‘. De esta forma los objetos pueden ser de dos tipos, las clases y las instancias. Las clases definen la disposición y la funcionalidad básicas de los objetos, y las instancias son objetos ‘‘utilizables” basados en los patrones de una clase particular. ...

Como ya se menciono anteriormente Protopy explota las novedades de JavaScript 1.7, para los iteradores el constructor de tipos prevee el metodo verb|__iter__| con la finalidad de que los objetos generados en base al tipo sean iterables.

Los primeros tipos que surgen para la organizacion de datos dentro de la librerias con los ‘‘Sets” y los ‘‘Diccionarios’‘, hambos aproximan su estructura a las estructuras homonimas en python, brindando una funcionalidad similar. Si bien la estructura ‘‘hasheable” nativa a JavaScript en un objeto, los diccionarios de Protopy permiten el uso de objetos como claves en lugar de solo cadenas.

CAPÍTULO 10

Extendiendo el DOM

Si bien el *DOM* ofrece ya una *API* muy completa para acceder, añadir y cambiar dinámicamente el contenido estructurado en el documento HTML.

CAPÍTULO 11

Manejando los eventos

%event.connect %event.

CAPÍTULO 12

Envolviendo a gears

%Almacenamiento en la base de datos local %LocalServer para guradar codigo

CAPÍTULO 13

Auditando el codigo

%Logger

Interactuando con el servidor

%HttpRequest

Soporte para json

%JSON REF http://www.json.org/ %Esto es algo sobre json, quiza no valla aca JSONbrinda un buen soporte al intercambio de datos, resultando de fásil lectura/escritura para las personas y de un rapido interpretacion/generacion para las maquinas. Se basa en un subconjunto del lenguaje de programación JavaScript, estándar ECMA-262 3ª Edición - Diciembre de 1999. Este formato de texto es completamente independiente del lenguaje de programacion, pero utiliza convenciones que son familiares para los programadores de lenguajes de la familia ‘‘C’‘, incluyendo C, C + +, C #, Java, JavaScript, Perl, Python y muchos otros.

JSON se basa en dos estructuras: * Una colección de pares nombre / valor. En varios lenguajes esto se

representa mediante un objeto, registro, estructura, diccionario, tabla hash, introducido lista o matriz asociativa.

- Una lista ordenada de valores. En la mayoría de los lenguajes esto se representa como un arreglo, matriz, vector, lista, o secuencia.

Estas son estructuras de datos universales. Prácticamente todos los lenguajes de programación modernos las soportan de una forma u otra. Tiene sentido que un formato de datos que es intercambiable con los lenguajes de programación también se basan en estas estructuras.

Para mas informacion sobre JSONhttp://www.json.org/

%Ahora vemos que hace protopy, la necesidad Mientras que un cliente se encuentre sin conexion con el servidor web, es capaz de generar y almacenar datos usando su base de datos local. Al reestablecer la conexion con el servidor web, estos datos deben ser transimitos a la base de datos central para su actulizacion y posterior sincronizacion del resto de los clinetes. La transferencia de datos involucra varios temas, uno de ellos y que compete a este apartado, es el formato de los datos que se deben pasar por la conexcion; este formato debe ser ‘‘comprendido” tanto por el cliente como por el servidor. Desde un primer momento se penso en JSONcomo el formato de datos a utilizar, es por esto que Protopy incluye un modulo para trabajar con el mismo.

%Porque no xml? No existe una razon concreta por la cual se deja de lado el soporte en Protopy para XML como formato de datos; aunque se puede mencionar la simplicidad de implementacion de un parser JSONcontra la implementacion de uno en XML. Para el leector interesado agregar el sporte para XML en Protopy consta de escribir un modulo que realize esa tarea y agregarlo al paquete base.

%El como El soporte para JSONse encuentra en el modulo ‘‘json” entre los modulos estandar de Protopy. Este brinda soporte al pasaje de estructuras de datos JavaScript a JSON y viceversa. Los tipos base del lenguje JavaScript estan soportados y tienen su reprecentacion correspondiente, object, array, number, string, etc. pero este modulo interpreta ademas de una forma particular a aquellos objetos que implementen el metodo verb|__json__|, dejando de este modo en manos del desarrollador la reprecentacion en JSON de determinado objetos. La inclusion del metodo

verb|__json__| resulta de especial impotancia a la hora de pasar a JSON los objetos creados en base a tipos definidos por el desarrollador mediante el contructor ‘‘type’‘.

Con el soporte de datos ya establecidos en la libreria, el framework solo debe limitarce a hacer uso de él y asegurar la correcta sincronizacion de datos entre el cliente y el servidor web, este tema se retomara en el capitulo de sincronizacion. %TODO: retomar este tema para no ser un mentiroso :)

Ejecutando codigo remoto

%JSON-RPC http://json-rpc.org/ %XML-RPC http://www.xmlrpc.com/ El RPC (del inglés Remote Procedure Call, Llamada a Procedimiento Remoto) es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos. El protocolo es un gran avance sobre los sockets usados hasta el momento. De esta manera el programador no tenía que estar pendiente de las comunicaciones, estando éstas encapsuladas dentro de las RPC.

Las RPC son muy utilizadas dentro del paradigma cliente-servidor. Siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando éste de vuelta el resultado de dicha operación al cliente.

Hay distintos tipos de RPC, muchos de ellos estandarizados como pueden ser el RPC de Sun denominado ONC RPC (RFC 1057), el RPC de OSF denominado DCE/RPC y el Modelo de Objetos de Componentes Distribuidos de Microsoft DCOM, aunque ninguno de estos es compatible entre sí. La mayoría de ellos utilizan un lenguaje de descripción de interfaz (IDL) que define los métodos exportados por el servidor.

Hoy en día se está utilizando el XML como lenguaje para definir el IDL y el HTTP como protocolo de red, dando lugar a lo que se conoce como servicios web. Ejemplos de éstos pueden ser SOAP o XML-RPC. XML-RPC es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos y HTTP como protocolo de transmisión de mensajes.[1]

Es un protocolo muy simple ya que sólo define unos cuantos tipos de datos y comandos útiles, además de una descripción completa de corta extensión. La simplicidad del XML-RPC está en contraste con la mayoría de protocolos RPC que tiene una documentación extensa y requiere considerable soporte de software para su uso.

Fue creado por Dave Winer de la empresa UserLand Software en asociación con Microsoft en el año 1998. Al considerar Microsoft que era muy simple decidió añadirle funcionalidades, tras las cuales, después de varias etapas de desarrollo, el estándar dejó de ser sencillo y se convirtió en lo que es actualmente conocido como SOAP. Una diferencia fundamental es que en los procedimientos en SOAP los parámetros tienen nombre y no interesan su orden, no siendo así en XML-RPC.[2

Parte IV

Glosario

API Application-Programming-Interface; conjunto de funciones y procedimientos (o métodos, si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

DOM Document-Object-Model; interfaz de programación de aplicaciones que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos.

JSON JavaScript-Object-Notation; formato ligero para el intercambio de datos.

RPC Remote-Procedure-Call; es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos.

field An attribute on a *model*; a given field usually maps directly to a single database column.

generic view A higher-order *view* function that abstracts common idioms and patterns found in view development and abstracts them.

model Models store your application's data.

MTV hola

MVC Model-view-controller; a software pattern.

project A Python package – i.e. a directory of code – that contains all the settings for an instance of Django. This would include database configuration, Django-specific options and application-specific settings.

property Also known as "managed attributes", and a feature of Python since version 2.2. From the property documentation:

> Properties are a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls. [...] You could only do this by overriding `__getattr__` and `__setattr__`; but overriding `__setattr__` slows down all attribute assignments considerably, and overriding `__getattr__` is always a bit tricky to get right. Properties let you do this painlessly, without having to override `__getattr__` or `__setattr__`.

queryset An object representing some set of rows to be fetched from the database.

slug A short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs. For example, in a typical blog entry URL:

```
http://www.djangoproject.com/weblog/2008/apr/12/spring/
```

the last bit (`spring`) is the slug.

template A chunk of text that separates the presentation of a document from its data.

view A function responsible for rending a page.

BSD ve ese de

i18n La internacionalización es el proceso de diseñar software de manera tal que pueda adaptarse a diferentes idiomas y regiones sin la necesidad de realizar cambios de ingeniería ni en el código. La localización es el proceso de adaptar el software para una región específica mediante la adición de componentes específicos de un locale y la traducción de los textos, por lo que también se le puede denominar regionalización. No obstante la traducción literal del inglés es la más extendida.

Parte V

Indices, glosario y tablas

- *Índice*
- *Índice de Módulos*
- *Glosario*

Parte VI

Referencia sobre el lenguaje Python

Modularidad

A.1 Ámbito de nombres

TODO

A.2 Módulos

Un módulo en Python es un archivo con código python. Usualmente con la extensión .py. Un módulo puede ser importado en el ámbito de nombres local mediante la sentecia **import**.

Por ejemplo, consideremos el módulo ***funciones.py***

```
# coding: utf-8

def media(lista):
        return float(sum(lista)) / len(lista)

def media_geo(lista):
        # La raíz puede expresarse como potencia > 1
        return reduce(lambda x, y: x*y, lista) ** 1.0/len(lista)
```

Para importar el módulo al ámbito de nombres local se puede utilizar la sentencia **import**.

```
>>> import funciones
>>> dir(funciones)
```

A.3 Paquete

Un paquete es una colección de uno o más módulos, contenidos en una directorio. Para que un directorio sea tratado como paquete debe crearse un módulo con el nombre __init__.py.

El módulo __init__ puede contener código que será evaluado si se realiza una import con el nombre del paquete como argumento, o si se realiza la importación de todos los símbolos:

```
from mi_paquete import * # Se evalua __init__.py
```

A.4 Módulo de expresiones regulares “re”

El módulo de expresiones regulares de Python permite recuperar gurpos nombrados.

```
r'persona/(?P<nombre>\w+)/(?P<edad>\d{2,3})'
```

En la expresión regular anterior se pueden recuperar el grupo **nombre**, que es un gurpo de uno o más letras, y el grupo **edad**, que es un entero de 2 o 3 cifras.

```
>>> import re
>>> expresion = re.compile(r'persona/(?P<nombre>\w+)/(?P<edad>\d{2,3})')
>>> match = expresion.search('persona/nahuel/25')
>>> match.group('nombre')
'nahuel'
>>> match.group('edad')
'25'
```

A.5 Metaprogramación mediante metaclases

Se puede definir la estructura de una clase mediante otra clase que herede de type.

A.6 Objetos llamables o *callables*

Un objeto callable

Bibliografía

[WikiCGI2009] *Interfaz de entrada común*, Wikipedia, 2009, último acceso Agosto 2009, http://es.wikipedia.org/wiki/Common_Gateway_Interface#Intercambio_de_informaci.C3.B3n:_Variables_de_entorno

[WIK001] *Software Framework*, Wikipedia, 2009, http://en.wikipedia.com/software_framework, última visita Agosto de 2009.

[WIKI002] *Web Framework*, Wikipedia, 2009, http://en.wikipedia.org/wiki/Web_application_framework, última visita Agosto de 2009.

Parte VII

Referencia sobre Django

Instalación de Django

La mayoría de la gente querrá instalar el lanzamiento oficial más reciente de http://www.djangoproject.com/download/. Django usa el método `distutils` estándar de instalación de Python, que en el mundo de Linux es así:

1. Baja el tarball, que se llamará algo así como *Django-0.96.tar.gz*
2. `tar xzvf Django-*.tar.gz`
3. `cd Django-*`
4. `sudo python setup.py install`

En Windows, recomendamos usar 7-Zip para manejar archivos comprimidos de todo tipo, incluyendo `.tar.gz`. Puedes bajar 7-Zip de http://www.djangoproject.com/r/7zip/.

Cambia a algún otro directorio e inicia `python`. Si todo está funcionando bien, deberías poder importar el módulo `django`:

```
>>> import django
>>> django.VERSION
(0, 96, None)
```

APÉNDICE C

Comandos del módulo manage

C.1 El comando syncdb

El comando syncdb busca los modelos de todas las aplicaciones instaladas. Por cada modelo, genera el SQL necesario para crear las tablas relacionales y mediante la configuración definida en el módulo settings, se connecta con la base de datos y ejecuta las secuencia SQL, creando así las tablas del modelo que no existan.

C.2 El comando runserver

Este comando lanza el servidor de desarrollo. Generalmente se ejecuta en el puerto 8000.

C.3 El comando validate

Este comando recibe puede no recibir argumentos o una lista de aplicaciones que validar. Realiza una verficación de sintaxis

APÉNDICE D

Comandos de usuario

Django permite

Parte VIII

Referencia de la definición de modelos

El *doff-modelos* se explica lo básico de la definición de modelos. Existe un enorme rango de opciones disponibles que no se han cubierto en otro lado. Este apéndice explica toda opción disponible en la definición de modelos.

Campos

La parte más importante de un modelo – y la única requerida – es la lista de campos de la base de datos que define.

Restricciones en el nombre de los campos

Existen solo dos restricciones en el nombre de los campos:

1. Un nombre de campo no puede ser una palabra reservada, porque eso ocasionaría un error de sintaxis, por ejemplo:

```
var Ejemplo = type('Ejemplo', [ models.Model ], {
    var = new models.IntegerField() // 'var' es una palabra reservada!
});
```

1. Un nombre de campo no puede contener dos o más guiones bajos consecutivos, debido a la forma en que trabaja la sintaxis de las consultas de búsqueda, por ejemplo:

```
var Ejemplo = type('Ejemplo', [models.Model], {
    foo__bar = new models.IntegerField() // 'foo__bar' tiene dos guiones bajos!
});
```

Estas limitaciones se pueden manejar sin mayores problemas, dado que el nombre del campo no necesariamente tiene que coincidir con el nombre de la columna en la base de datos. Ver *db_column*, más abajo.

Las palabras reservadas de SQL, como `join`, `where`, o `select`, *son* permitidas como nombres de campo, dado que se "escapean" todos los nombres de tabla y columna de la base de datos en cada consulta SQL subyacente.

Cada campo en el modelo debe ser una instancia del tipo de campo apropiado. Los tipos de `Field` son utilzados para determinar algunas cosas:

- El tipo de columna de la base de datos (ej.:, `INTEGER`, `VARCHAR`).
- El widget a usar en la generacion de formularios (ej., `<input type="text">`, `<select>`).
- Los requerimientos mínimos de validación.

A continuación, una lista completa de los campos, ordenados alfabéticamente. Los campos de relación (`ForeignKey`, etc.) se tratan en la siguiente sección.

E.1 AutoField

Un `IntegerField` que se incrementa automáticamente de acuerdo con los IDs disponibles. Normalmente no es necesario utilizarlos directamente ya que se agrega un campo de clave primaria automáticamente al modelo si no se especifica una clave primaria.

E.2 BooleanField

Un campo Verdadero/Falso.

E.3 CharField

Un campo string, para cadenas cortas o largas. Para grandes cantidades de texto, usar `TextField`.

`CharField` requiere un argumento extra, `max_length`, que es la longitud máxima (en caracteres) del campo. Esta longitud máxima es reforzada a nivel de la base de datos y en la validación.

E.4 DateField

Un campo de fecha. `DateField` tiene algunos argumentos opcionales extra, como se muestra en la *Tabla*.

Cuadro E.1: Argumentos opcionales extra de DateField

Argumento	**Descripción**
`auto_now`	Asigna automáticamente al campo un valor igual al momento en que se salva el objeto. Es útil para las marcas de tiempo "última modificación". Observar que *siempre* se usa la fecha actual; no es un valor por omisión que se pueda sobreescribir.
`auto_now_add`	Asigna automáticamente al campo un valor igual al momento en que se crea el objeto. Es útil para la creación de marcas de tiempo. Observar que *siempre* se usa la fecha actual; no es un valor por omisión que se pueda sobreescribir.

E.5 DateTimeField

Un campo de fecha y hora. Tiene las mismas opciones extra que `DateField`.

E.6 EmailField

Un `CharField` que chequea que el valor sea una dirección de email válida. No acepta `max_length`; su `max_length` se establece automáticamente en 75.

E.7 FileField

Advertencia: No implementado actualmente ver bien la doc.

Un campo de captura de archivos. Tiene un argumento *requerido*, como se ve en la *Tabla*.

Cuadro E.2: Opciones extra de FileField

Argumento	Descripción
`upload_to`	Una ruta del sistema de archivos local que se agregará a la configuración de `MEDIA_ROOT` para determinar el resultado de la función de ayuda `get_<fieldname>_url()`.

Esta ruta puede contener formato `strftime`, que será reemplazada por la fecha y hora de la captura del archivo (de manera que los archivos capturados no llenen una ruta dada).

El uso de un `FileField` o un `ImageField` en un modelo requiere algunos pasos:

1. En el archivo de configuración (settings), es necesario definir `MEDIA_ROOT` con la ruta completa al directorio donde quieras que Django almacene los archivos subidos. (Por performance, estos archivos no se almacenan en la base de datos.) Definir `MEDIA_URL` con la URL pública base de ese directorio. Asegurarse de que la cuenta del usuario del servidor web tenga permiso de escritura en este directorio.
2. Agregar el `FileField` o `ImageField` al modelo, asegurándose de definir la opción `upload_to` para decirle a Django a cual subdirectorio de `MEDIA_ROOT` debe subir los archivos.
3. Todo lo que se va a almacenar en la base de datos es la ruta al archivo (relativa a `MEDIA_ROOT`). Seguramente preferirás usar la facilidad de la función `get_<fieldname>_url` provista por Django. Por ejemplo, si tu `ImageField` se llama `mug_shot`, puedes obtener la URL absoluta a tu image en un plantilla con `{{object.get_mug_shot_url }}`.

Por ejemplo, digamos que tu `MEDIA_ROOT` es `'/home/media'`, y `upload_to` es `'photos/%Y/%m/%d'`. La parte `'%Y/%m/%d'` de `upload_to` es formato strftime; `'%Y'` es el año en cuatro dígitos, `'%m'` es el mes en dos digitos, y `'%d'` es el día en dos dígitos. Si subes un archivo el 15 de enero de 2007, será guardado en `/home/media/photos/2007/01/15`.

Si quieres recuperar el nombre en disco del archivo subido, o una URL que se refiera a ese archivo, o el tamaño del archivo, puedes usar los métodos `get_FIELD_filename()`, `get_FIELD_url()`, y `get_FIELD_size()`. Ver el Apéndice C para una explicación completa de estos métodos.

Nota: Cualquiera sea la forma en que manejes tus archivos subidos, tienes que prestar mucha atención a donde los estás subiendo y que tipo de archivos son, para evitar huecos en la seguridad. *Valida todos los archivos subidos* para asegurarte que esos archivos son lo que piensas que son.

Por ejemplo, si dejas que cualquiera suba archivos ciegamente, sin validación, a un directorio que está dentro de la raíz de documentos (*document root*) de tu servidor web, alguien podría subir un script CGI o PHP y ejecutarlo visitando su URL en tu sitio. ¡No permitas que pase!

E.8 FilePathField

Un campo cuyas opciones están limitadas a los nombres de archivo en una cierta ruta en el sistema de archivos. Tiene tres argumentos especiales, que se muestran en la *Tabla*.

Cuadro E.3: Opciones extra de FilePathField

Argumento	Descripción
`path`	*Requerido*; la ruta absoluta en el sistema de archivos hacia el directorio del cual este `FilePathField` debe tomar sus opciones (ej.: `"/home/images"`).
`match`	Opcional; una expresión regular como string, que `FilePathField` usará para filtrar los nombres de archivo. Observar que la regex será aplicada al nombre de archivo base, no a la ruta completa (ej.: `"foo.*\.txt^"`, va a matchear con un archivo llamado `foo23.txt`, pero no con `bar.txt` o `foo23.gif`).
`recursive`	Opcional; `true` o `false`. El valor por omisión es `false`. Especifica si deben incluirse todos los subdirectorios de `path`.

Por supuesto, estos argumentos pueden usarse juntos.

El único 'gotcha' potencial es que `match` se aplica sobre el nombre de archivo base, no la ruta completa. De esta manera, este ejemplo:

```
FilePathField({path:"/home/images", match:"foo.*", recursive:true})
```

va a matchear con `/home/images/foo.gif` pero no con `/home/images/foo/bar.gif` porque el `match` se aplica al nombre de archivo base (`foo.gif` y `bar.gif`).

E.9 FloatField

Un numero de punto flotante, representado en JavaScript por una instancia de `Number`. Tiene dos argumentos requeridos, que se muestran en la *Tabla*.

Cuadro E.4: Opciones extra de FloatField

Argumento	Descripción
`max_digits`	La cantidad máximo de dígitos permitidos en el número.
`decimal_places`	La cantidad de posiciones decimales a almacenar con el número.

Por ejemplo, para almacenar números hasta 999 con una resolución de dos decimales, hay que usar:

```
models.FloatField(..., max_digits=5, decimal_places=2)
```

Y para almacenar números hasta aproximadamente mil millones con una resolución de diez dígitos decimales, hay que usar:

```
models.FloatField(..., max_digits=19, decimal_places=10)
```

E.10 ImageField

Advertencia: No este implementado actualmente ver bien la doc.

Similar a `FileField`, pero valida que el objeto capturado sea una imagen válida. Tiene dos argumentos opcionales extra, `height_field` y `width_field`, que si se utilizan, serán auto-rellenados con la altura y el ancho de la imagen cada vez que se guarde una instancia del modelo.

Además de los métodos especiales `get_FIELD_*` que están disponibles para `FileField`, un `ImageField` tiene también los métodos `get_FIELD_height()` y `get_FIELD_width()`. Éstos están documentados en el *Referencia de la API de base de datos*.

E.11 IntegerField

Un entero.

E.12 IPAddressField

Una dirección IP, en formato string (ej.: `"24.124.1.30"`).

E.13 NullBooleanField

Similar a `BooleanField`, pero permite `NULL` como opción. Usar éste en lugar de un `BooleanField` con `null = true`.

E.14 PositiveIntegerField

Similar a `IntegerField`, pero debe ser positivo.

E.15 PositiveSmallIntegerField

Similar a `PositiveIntegerField`, pero solo permite valores por debajo de un límite. El valor máximo permitido para estos campos depende de la base de datos, pero como las bases de datos tienen un tipo entero corto de 2 bytes, el valor máximo positivo usualmente es 65,535.

E.16 SlugField

"Slug" es un término de la prensa. Un *slug* es una etiqueta corta para algo, que contiene solo letras, números, guiones bajos o simples. Generalmente se usan en URLs.

De igual forma que en `CharField`, puedes especificar `max_length`. Si `max_length` no está especificado, el valor por omisión es de 50.

Un `SlugField` implica `db_index=true` debido a que son los se usan principalmente para búsquedas en la base de datos.

E.17 SmallIntegerField

Similar a `IntegerField`, pero solo permite valores en un cierto rango dependiente de la base de datos (usualmente -32,768 a +32,767).

E.18 TextField

Un campo de texto de longitud ilimitada.

E.19 TimeField

Un campo de hora. Acepta las mismas opciones de autocompletación de `DateField` y `DateTimeField`.

E.20 URLField

Un campo para una URL. Si la opción `verify_exists` es `true` (valor por omisión), se chequea la existencia de la URL dada (la URL carga y no da una respuesta 404).

Como los otros campos de caracteres, `URLField` toma el argumento `max_length`. Si no se especifica, el valor por omisión es 200.

Opciones para todos los campos

Los siguientes argumentos están disponibles para todos los tipos de campo. Todos son opcionales.

F.1 null

Si está en `true`, se almacenaran valores vacíos como `NULL` en la base de datos. El valor por omisión es `false`.

Los valores de string nulo siempre se almacenan como strings vacíos, no como `NULL`. `null=true` se debe utilizar solo para campos no-string, como enteros, booleanos y fechas. En los dos casos, también es necesario establecer `blank=true` si se desea permitir valores vacíos en los formularios, ya que el parámetro `null` solo afecta el almacenamiento en la base de datos (ver la siguiente sección, titulada *blank*).

Se debe evitar utilizar `null` en campos basados en string como `CharField` y `TextField` salvo que se tenga una excelente razón para hacerlo. Si un campo basado en string tiene `null=true`, eso significa que tiene dos valores posibles para "sin datos": `NULL` y el string vacío. En la mayoría de los casos, esto es redundante; la convención es usar el string vacío, no `NULL`.

F.2 blank

Si está en `true`, está permitido que el campo esté en blanco. El valor por omisión es `false`.

Este es diferente de `null`. `null` solo se relaciona con la base de datos, mientras que `blank` está relacionado con la validación. Si un campo tiene `blank=true`, la validación permitirá la entrada de un valor vacío. Si un campo tiene `blank=false`, es un campo requerido.

F.3 choices

Un arreglo conteniendo tuplas para usar como opciones para este campo.

Si esto está dado, el sistema de formualarios utilizará un cuadro de selección en lugar del campo de texto estándar, y limitará las opciones a las dadas.

Una lista de opciones se ve así:

```
var YEAR_IN_SCHOOL_CHOICES = [
    ['FR', 'Freshman'],
    ['SO', 'Sophomore'],
    ['JR', 'Junior'],
    ['SR', 'Senior'],
    ['GR', 'Graduate'],
]
```

El primer elemento de cada tupla es el valor actual a ser almacenado. El segundo elemento es el nombre legible por humanos para la opción.

La lista de opciones puede ser definida también como parte del modelo:

```
var Foo = type('Foo', [ models.Model ], {
    GENDER_CHOICES: [
        ['M', 'Male'],
        ['F', 'Female']
    ],
    gender: new models.CharField({ max_length:1, choices:GENDER_CHOICES}),
    });
```

o fuera del modelo:

```
var GENDER_CHOICES: [
        ['M', 'Male'],
        ['F', 'Female']
    ];
var Foo = type ('Foo', [models.Model], {
    gender: new models.CharField({ max_length:1, choices:GENDER_CHOICES}),
    });
```

Para cada campo del modelo que tenga establecidas `choices`, Se agregará un método para recuperar el nombre legible por humanos para el valor actual del campo. Ver *Referencia de la API de base de datos* para más detalles.

F.4 db_column

El nombre de la columna de la base de datos a usar para este campo. De no estar definido, se utilizará el nombre del campo. Esto es útil cuando se está definiendo un modelo sobre una base de datos existente.

Si el nombre de columna de la base de datos es una palabra reservada de SQL, o contiene caracteres que no están permitidos en un nombre de variable, no hay problema. Los nombres de columna y tabla son escapeados por comillas detrás de la escena.

F.5 db_index

Si está en `true`, Un índice es creado en la base de datos para esta columna cuando cree la tabla.

F.6 default

El valor por omisión del campo.

F.7 editable

Si es `false`, el campo no será editable en el procesamiento de formularios. El valor por omisión es `true`.

F.8 help_text

Texto de ayuda extra a ser mostrado bajo el campo en el formulario. Es útil como documentación aunque el objeto no termine siendo representado en un formulario.

F.9 primary_key

Si es `true`, este campo es la clave primaria del modelo.

Si no se especifica `primary_key=true` para ningún campo del modelo, se agregará automáticamente este campo:

```
id = new models.AutoField('ID', { primary_key: true });
```

Por lo tanto, no es necesario establecer `primary_key=true` en ningún campo, salvo que se quiera sobreescribir el comportamiento por omisión de la clave primaria.

`primary_key=true` implica `blank=false`, `null=false`, y `unique=true`. Solo se permite una clave primaria en un objeto.

F.10 radio_admin

Por omisión, la generacion de formularios usa una interfaz de cuadro de selección (<select>) para campos que son `ForeignKey` o tienen `choices`. Si `radio_admin` es `true`, un radio-button es utilizado en su lugar.

No utilice esto para un campo que no sea `ForeignKey` o no tenga `choices`.

F.11 unique

Si es `true`, el valor para este campo debe ser único en la tabla.

F.12 unique_for_date

Asignar como valor el nombre de un `DateField` o `DateTimeField` para requerir que este campo sea único para el valor del campo tipo fecha, por ejemplo:

```
var Story = type('Story', [ models.Model ] {
    pub_date: new models.DateTimeField(),
    slug: new models.SlugField({unique_for_date:"pub_date"}),
    ...
});
```

En este código, no se permitie la creación de dos historias con el mismo slug publicados en la misma fecha. Esto difiere de usar la restricción `unique_together` en que solo toma en cuenta la fecha del campo `pub_date`; la hora no importa.

F.13 unique_for_month

Similar a `unique_for_date`, pero requiere que el campo sea único con respecto al mes del campo dado.

F.14 unique_for_year

Similar a `unique_for_date` y `unique_for_month`, pero para el año.

F.15 verbose_name

Cada tipo de campo, excepto `ForeignKey`, `ManyToManyField`, y `OneToOneField`, toma un primer argumento posicional opcional – un nombre descriptivo –. Si el nombre descriptivo no está dado, se crea automáticamente usando el nombre de atributo del campo, convirtiendo guiones bajos en espacios.

En este ejemplo, el nombre descriptivo es `"Person's first name"`:

```
first_name = new models.CharField("Person's first name", { max_length: 30 })
```

En este ejemplo, el nombre descriptivo es `"first name"`:

```
first_name = new models.CharField({maxlength: 30})
```

`ForeignKey`, `ManyToManyField`, y `OneToOneField` requieren que el primer argumento sea una clase del modelo, en este caso hay que usar `verbose_name` como argumento con nombre:

```
poll = new models.ForeignKey(Poll, {verbose_name: "the related poll"})
sites = new models.ManyToManyField(Site, {verbose_name: "list of sites"})
place = new models.OneToOneField(Place, {verbose_name: "related place"})
```

La convención es no capitalizar la primera letra del `verbose_name` estas son pasadas a mayúscula automáticamente cuando sea necesario.

Relaciones

Es claro que el poder de las bases de datos se basa en relacionar tablas entre sí. Los tres tipos de relaciones más comunes en las bases de datos estan soportadas: muchos-a-uno, muchos-a-muchos, y uno-a-uno (utilizada indirectamente en la herencia).

G.1 Relaciones Muchos-a-Uno

El campo `ForeignKey` definie las relaciones muchos-a-uno. Se usa como cualquier otro tipo `Field`: incluyéndolo como un atributo en el modelo.

`ForeignKey` requiere un argumento posicional: el tipo al cual se relaciona el modelo.

Por ejemplo, si un modelo `Car` tiene un `Manufacturer` – es decir, un `Manufacturer` fabrica múltiples autos pero cada `Car` tiene solo un `Manufacturer` – la definición es:

```
var Manufacturer = type('Manufacturer', [ models.Model ], {
    ...
});

var Car = type('Car', [ models.Model ],
    manufacturer: new models.ForeignKey(Manufacturer),
    ...
});
```

Para crear una relación *recursiva* – un objeto que tiene una relación muchos-a-uno con él mismo – `models.ForeignKey('this')`:

```
var Employee = type('Employee', [ models.Model ], {
    manager: new models.ForeignKey('this'),
    ...
});
```

Si se necesita crear una relación con un modelo que aún no se ha definido, el nombre del modelo puede ser utilizado en lugar del objeto modelo:

```
var Car = type('Car', [ models.Model ], {
    manufacturer: new models.ForeignKey('Manufacturer'),
    ...
```

```
});

var Manufacturer = type('Manufacturer', [ models.Model ], {
    ...
});
```

Observar que de todas formas solo se pueden usar strings para hacer referencia a modelos dentro del mismo archivo `models.js` – no se pueden usar strings para hacer referencias a un modelo en una aplicación diferente, o hacer referencia a un modelo que ha sido requerido de cualquier otro lado.

Detrás de la escena, `"_id"` es agregado al nombre de campo para crear su nombre de columna en la base de datos. En el ejemplo anterior, la tabla de la base de datos correspondiente al modelo `Car`, tendrá una columna `manufacturer_id`. (Esto puede se puede cambiar explícitamente especificando `db_column`; ver más arriba en la sección *db_column*.) De todas formas, el código nunca debe utilizar el nombre de la columna de la base de datos, salvo que escribas SQL. Siempre se utilizaran los nombres de campo del modelo.

Se sugiere, pero no es requerido, que el nombre de un campo `ForeignKey` (`manufacturer` en el ejemplo) sea el nombre del modelo en minúsculas. Igualmente se puede poner cualquier nombre. Por ejemplo:

```
var Car = type('Car', [ models.Model ], {
    company_that_makes_it: new models.ForeignKey(Manufacturer),
    // ...
});
```

Los campos `ForeignKey` reciben algunos argumentos extra para definir como debe trabajar la relación (ver *Tabla*). Todos son opcionales.

Advertencia: ver esta tabla hay opciones que no estan.

Cuadro G.1: Opciones de ForeignKey

Argumento	Descripción
edit_inline	Si no es false, este objeto relacionado se edita "inline" en la página del objeto relacionado. Esto significa que el objeto no tendrá su propia interfaz de administración. Usa models.TABULAR o models.STACKED, que designan si los objetos editables inline se muestran como una tabla o como una pila de conjuntos de campos, respectivamente.
limit_choices_to	Un diccionario para buscar argumentos y valores (ver el Apéndice C) que limita las opciones de administración disponibles para este objeto. Usa esto con funciones del módulo datetime de Python para limitar las opciones de fecha de los objetos. Por ejemplo: limit_choices_to: {'pub_date__lte': datetime.now} sólo permite la elección de objetos relacionados con pub_date anterior a la fecha/hora actual. En lugar de un diccionario, esto puede ser un objeto Q (ver Apéndice C) para consultas más complejas. No es compatible con edit_inline.
max_num_in_admin	Para objetos editados inline, este es el número máximo de objetos relacionados a mostrar en la interfaz de administración. Por lo tanto, si una pizza puede tener como máximo diez ingredientes, max_num_in_admin=10 asegurará que un usuario nunca ingresará más de diez ingredientes. Observar que esto no asegura que no se puedan crear más de diez ingredientes relacionados. Simplemente controla la interfaz de administración; no fortalece cosas a nivel de Python API o base de datos.
min_num_in_admin	La cantidad mínima de objetos relacionados que se muestran en la interfaz de administración. Normalmente, en el momento de la creación se muestran num_in_admin objetos inline , y en el momento de edición se muestran num_extra_on_change objetos en blanco además de todos los objetos relacionados preexistentes. De todas formas, nunca se mostrarán menos de min_num_in_admin objetos relacionados.
num_extra_on_change	La cantidad de campos en blanco extra de objetos relacionados a mostrar en el momento de realizar cambios.
num_in_admin	El valor por omisión de la cantidad de objetos inline a mostrar en la página del objeto en el momento de agregar.
raw_id_admin	Solo muestra un campo para ingresar un entero en lugar de un menú desplegable. Esto es útil cuando se relaciona con un tipo de objeto que tiene demasiadas filas para que sea práctico utilizar una caja de selección. No es utilizado con edit_inline.
related_name	El nombre a utilizar para la relación desde el objeto relacionado de hacia éste objeto. Para más información, ver el Apéndice C.
to_field	El campo en el objeto relacionado con el cual se establece la relación. Por omisión, Django usa la clave primaria del objeto relacionado.

G.2 Relaciones Muchos-a-Muchos

Para definir una relación muchos-a-muchos, `ManyToManyField` es el campo. Al igual que `ForeignKey`, `ManyToManyField` requiere un argumento posicional: el tipo al cual se relaciona el modelo.

Por ejemplo, si una `Pizza` tiene múltiples objetos `Topping` – es decir, un `Topping` puede estar en múltiples pizzas y cada `Pizza` tiene múltiples ingredientes (toppings) – debe representarse así:

```
var Topping = type('Topping', [ models.Model ], {
    ...
});

var Pizza = type('Pizza', [ models.Model ], {
    toppings: new models.ManyToManyField(Topping),
    ...
});
```

Como sucede con `ForeignKey`, una relación de un objeto con sí mismo puede definirse usando el string `'this'` en lugar del nombre del modelo, y se pueden hacer referencias a modelos que todavía no se definieron usando un string que contenga el nombre del modelo. De todas formas solo se pueden usar strings para hacer referencia a modelos dentro del mismo archivo `models.js` – no se puede usar un string para hacer referencia a un modelo en una aplicación diferente, o hacer referencia a un modelo que ha sido importado de cualquier otro lado.

Se sugiere, pero no es requerido, que el nombre de un campo `ManyToManyField` (`toppings`, en el ejemplo) sea un término en plural que describa al conjunto de objetos relacionados con el modelo.

Detrás de la escena, se crea una tabla join intermedia para representar la relación muchos-a-muchos.

No importa cual de los modelos tiene el `ManyToManyField`, pero es necesario que esté en uno de los modelos – no en los dos.

Los objetos `ManyToManyField` toman algunos argumentos extra para definir como debe trabajar la relación (ver *Tabla*). Todos son opcionales.

Advertencia: ver esta tabla hay opciones que no estan.

Cuadro G.2: Opciones de ManyToManyField

Argumento	Descripción
`related_name`	El nombre a utilizar para la relación desde el objeto relacionado hacia este objeto. Ver Apéndice C para más información..
`filter_interface`	Usa una interfaz de "filtro" JavaScript agradable y discreta en lugar de la menos cómoda `<select multiple>` en el formulario administrativo de este objeto. El valos debe ser `models.HORIZONTAL` o `models.VERTICAL` (es decir, la interfaz debe apilarse horizontal o verticalmente).
`limit_choices_to`	Ver la descripción en `ForeignKey`.

`symmetrical`	Solo utilizado en la definición de `ManyToManyField` sobre sí mismo. Considera el siguiente modelo:
	`var Person = type('Person', [ models.Model ], {`
	`  friends: new models.ManyToManyField("this")`
	`});`
	Cuando Django procesa este modelo, identifica que tiene un `ManyToManyField` sobre sí mismo, y como resultado, no agrega un atributo `person_set` a la clase `Person`. En lugar de eso, se asumen que el `ManyToManyField` es simétrico – esto es, si yo soy tu amigo, entonces tu eres mi amigo.
	Si no deseas la simetría en las relaciones `ManyToMany` con `self`, establece `symmetrical` en `false`. Esto forzará a Django a agregar el descriptor para la relación inversa, permitiendo que las relaciones `ManyToMany` sean asimétricas.

`db_table`	El nombre de la tabla a crear para almacenar los datos de la relación muchos-a-muchos. Si no se provee, Django asumirá un nombre por omisión basado en los nombres de las dos tablas a ser vinculadas.

Opciones de los Metadatos del Modelo

Los metadatos específicos de un modelo viven en un `Object Meta` definido en el cuerpo del modelo:

```
var Book = type('Book', [ models.Model ], {
    title: new models.CharField({max_length:100}),

    Meta: {
        // model metadata options go here
        ...
    }
});
```

Los metadatos del modelo son "cualquier cosa que no sea un campo", como opciones de ordenamiento, etc.

Las secciones que siguen presentan una lista de todas las posibles `Meta` opciones. Ninguna de estas opciones es requerida. Agregar `Meta` a un modelo es completamente opcional.

H.1 db_table

El nombre de la tabla de la base de datos a usar para el modelo.

Si no se define el nombre de la tabla de la base de datos es derivado automáticamente a partir del nombre del modelo y la aplicación que lo contiene. Un nombre de tabla de base de datos de un modelo se construye uniendo la etiqueta de la aplicación del modelo – el nombre que tiene la aplicacion – con el nombre del modelo, con un guión bajo entre ellos.

Por ejemplo, para la aplicación `books`, un modelo definido como `Book` tendrá una tabla en la base de datos llamada `book_books`.

Para sobreescribir el nombre de la tabla de la base de datos, se debe usar el parámetro `db_table` dentro de `Meta`:

```
var Book = type('Book', [ models.Model ], {
    ...

    Meta: {
        db_table: 'things_to_read'
    }
});
```

Si el nombre de tabla de base de datos es una palabra reservada de SQL, o contiene caracteres que no están permitidos en los nombres de variable, no hay problema. Los nombres de tabla y de columna son escapeados con comillas al generar el SQL.

H.2 get_latest_by

El nombre de un `DateField` o `DateTimeField` del modelo. Esto especifica el campo a utilizar por omisión en el método `latest()` del `Manager` del modelo.

Aquí hay un ejemplo:

```
var CustomerOrder = type('CustomerOrder' ,[ models.Model ], {
    order_date: new models.DateTimeField(),
    ...

    Meta: {
        get_latest_by: "order_date"
    }
});
```

Ver *Referencia de la API de base de datos* para más información sobre el método `latest()`.

H.3 order_with_respect_to

Marca este objeto como "ordenable" con respecto al campo dado. Esto se utiliza casi siempre con objetos relacionados para permitir que puedan ser ordenados respecto a un objeto padre. Por ejemplo, si un `Answer` se relaciona a un objeto `Question`, y una pregunta tiene más de una respuesta, y el orden de las respuestas importa:

```
var Answer = type('Answer', [ models.Model ], {
    question: new models.ForeignKey(Question),
    ...

    Meta: {
        order_with_respect_to: 'question'
    }
});
```

H.4 ordering

El ordenamiento por omisión del objeto, utilizado cuando se obtienen listas de objetos:

```
var Book = type('Book', [ models.Model ], {
    title: new models.CharField({maxlength: 100}),

    Meta: {
        ordering: ['title']
    }
});
```

Esto es una arreglo de strings. Cada string es un nombre de campo con un prefijo opcional −, que indica orden descendente. Los campos sin un − precedente se ordenarán en forma ascendente. Use el string `"?"` para ordenar al azar.

Por ejemplo, para ordenar por un campo `title` en orden ascendente:

```
ordering: ['title']
```

Para ordenar por `title` en orden descendente:

```
ordering: ['-title']
```

Para ordenar por `title` en orden descendente, y luego por `author` en orden ascendente:

```
ordering: ['-title', 'author']
```

H.5 unique_together

Conjuntos de nombres de campo que tomados juntos deben ser únicos:

```
var Employee = type('Employee', [ models.Model ], {
    department: new models.ForeignKey(Department),
    extension: new models.CharField({max_length: 10}),
    ...

    Meta: {
        unique_together: [["department", "extension"]]
    }
});
```

Esto es un arreglo de arreglos de campos que deben ser únicos cuando se consideran juntos. Es usado en la validacion de formularios y se refuerza a nivel de base de datos (esto es, se incluyen las sentencias `UNIQUE` apropiadas en la sentencia `CREATE TABLE`).

H.6 verbose_name

Un nombre legible por humanos para el objeto, en singular:

```
var CustomerOrder = type('CustomerOrder', [ models.Model ], {
    order_date: new models.DateTimeField(),
    ...

    Meta: {
        verbose_name: "order"
    }
});
```

Si no se define, se utilizará una versión adaptada del nombre del modelo, en la cual `CamelCase` se convierte en `camel case`.

H.7 verbose_name_plural

El nombre del objeto en plural:

```
var Sphynx = type('Sphynx', [ models.Model ], {
    ...

    Meta: {
        verbose_name_plural: "sphynges"
    }
});
```

Si no se define, se agregará una "s" al final del `verbose_name`.

Managers

Un `Manager` es la interfaz a través de la cual se proveen las operaciones de consulta de la base de datos a los modelos. Existe al menos un `Manager` para cada modelo en una aplicación.

La forma en que trabajan los tipos `Manager` está documentada en el *Referencia de la API de base de datos*. Esta sección trata específicamente las opciones del modelo que personaliza el comportamiento del `Manager`.

I.1 Nombres de Manager

Por omisión, se agrega un `Manager` llamado `objects` a cada tipo de modelo. De todas formas, si se quiere usar `objects` como nombre de campo, o usar un nombre distinto de `objects` para el `Manager`, se puede renombrar en cada uno de los modelos. Para renombrar el `Manager` para un modelo dato, define un atributo de clase de tipo `models.Manager()` en ese modelo, por ejemplo:

```
var models = require(doff.db.models.base');

var Person = type('Person', [ models.Model ], {
    ...

    people: new models.Manager(),
});
```

Usando este modelo de ejemplo, `Person.objects` generará una excepción `AttributeError` (dado que `Person` no tiene un atributo `objects`), pero `Person.people.all()` devolverá una lista de todos los objetos `Person`.

I.2 Managers Personalizados

Se puede utilizar un `Manager` personalizado en un modelo en particular extendiendo el tipo base `Manager` e instanciando un `Manager` personalizado.

Hay dos razones por las que se puede querer personalizar un `Manager`: para agregar métodos extra al `Manager`, y/o para modificar el `QuerySet` inicial que devuelve el `Manager`.

I.2.1 Agregando Métodos Extra al Manager

Agregar métodos extra al `Manager` es la forma preferida de agregar funcionalidad a nivel de tabla a los modelos. (Para funcionalidad a nivel de registro – esto es, funciones que actúan sobre una instancia simple de un objeto modelo – se deben usar métodos del modelo (ver *metodos del modelo*), no métodos de `Manager` personalizados.)

Un método `Manager` personalizado puede retornar cualquier cosa que se necesite. No tiene que retornar un `QuerySet`.

Por ejemplo, este `Manager` personalizado ofrece un método `with_counts()`, que retorna una lista de todos los objetos `OpinionPoll`, cada uno con un atributo extra `num_responses` que es el resultado de una consulta agregada:

```
require('doff.db.base', 'connection');

var PollManager = type('PollManager', [ models.Manager ], {

    with_counts: function() {
        var cursor = connection.cursor();
        cursor.execute("
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY 1, 2, 3
            ORDER BY 3 DESC");
        var result_list = [];
        for each (var row in cursor.fetchall()) {
            var p = new this.model({ id: row[0], question: row[1], poll_date: row[2]});
            p.num_responses = row[3];
            result_list.append(p);
        }
        return result_list;
    }
});

var OpinionPoll = type(OpinionPoll, [ models.Model ], {
    question: new models.CharField({ max_length: 200 }),
    poll_date: new models.DateField(),
    objects: new PollManager()
});

var Response = type('Response', [ models.Model ], {
    poll: new models.ForeignKey(Poll),
    person_name: new models.CharField({ max_length: 50 }),
    response: new models.TextField()
});
```

En este ejemplo, se puede usar `OpinionPoll.objects.with_counts()` para retornar la lista de objetos `OpinionPoll` con el atributo `num_responses`.

Otra cosa a observar en este ejemplo es que los métodos de un `Manager` pueden acceder a `this.model` para obtener el tipo del modelo a la cual están anexados.

I.2.2 Modificando los QuerySets iniciales del Manager

Un `QuerySet` base de un `Manager` devuelve todos los objetos en el sistema. Por ejemplo, usando este modelo:

```
var Book = type('Book', [ models.Model ], {
    title: new models.CharField({ max_length: 100 }),
    author: new models.CharField({ max_length: 50 })
});
```

la sentencia `Book.objects.all()` retornará todos los libros de la base de datos.

Se puede sobreescribir el `QuerySet` base, sobreescribiendo el método `Manager.get_query_set()`. `get_query_set()` debe retornar un `QuerySet` con las propiedades requeridas.

Por ejemplo, el siguiente modelo tiene *dos* managers – uno que devuelve todos los objetos, y otro que retorna solo los libros de Roald Dahl:

```
// First, define the Manager subclass.
var DahlBookManager = type('DahlBookManager', [ models.Manager ], {
    get_query_set: function() {
        return super(Manager, this).get_query_set().filter({ author: 'Roald Dahl' });
});

// Then hook it into the Book model explicitly.
var Book = type('Book', [ models.Model ], {
    title: new models.CharField({ max_length: 100 }),
    author: new models.CharField({ max_length: 50 }),

    objects: new models.Manager(), // The default manager.
    dahl_objects: new DahlBookManager() // The Dahl-specific manager.
});
```

Con este modelo de ejemplo, `Book.objects.all()` retornará todos los libros de la base de datos, pero `Book.dahl_objects.all()` solo retornará aquellos escritos por Roald Dahl.

Por supuesto, como `get_query_set()` devuelve un objeto `QuerySet`, se puede usar `filter()`, `exclude()`, y todos los otro métodos de `QuerySet` sobre él. Por lo tanto, estas sentencias son todas legales:

```
Book.dahl_objects.all();
Book.dahl_objects.filter({ title: 'Matilda' });
Book.dahl_objects.count();
```

Este ejemplo también señala otra técnica interesante: usar varios managers en el mismo modelo. Se pueden agregar tantas instancias de `Manager()` como se requieran. Esta es una manera fácil de definir "filters" comunes para tus modelos. Aquí hay un ejemplo:

```
var MaleManager = type('MaleManager', [ models.Manager ], {
    get_query_set: function() {
        return super(Manager, this).get_query_set().filter({ sex: 'M' });
    }
});

var FemaleManager = type('FemaleManager', [ models.Manager ], {
    get_query_set: function() {
        return super(Manager, this).get_query_set().filter({ sex: 'F' });
    }
});

var Person = type('Person', [ models.Model ], {
    first_name: new models.CharField({ max_length: 50 }),
    last_name: new models.CharField({ max_length: 50 }),
```

```
        sex: new models.CharField({ max_length: 1, choices: [['M', 'Male'], ['F', 'Female']] }),
        people: new models.Manager(),
        men: new MaleManager(),
        women: new FemaleManager(),
    });
```

Este ejemplo permite consultar `Person.men.all()`, `Person.women.all()`, y `Person.people.all()`, con los resultados predecibles.

Si se usan objetos `Manager` personalizados, el primer `Manager` que se encuentre (en el orden en el que están definidos en el modelo) tiene un status especial. Se interpreta el primer `Manager` definido en una clase como el `Manager` por omisión, por lo que generalmente es una buena idea que el primer `Manager` esté relativamente sin filtrar. En el último ejemplo, el manager `people` está definido primero – por lo cual es el `Manager` por omisión.

Métodos de Modelo

La forma de agregar funcionalidad es definiendo métodos en un modelo, de este modo se personaliza a nivel de registro. Mientras que los métodos `Manager` están pensados para hacer cosas a nivel de tabla, los métodos de modelo deben actual en una instancia particular del modelo.

Esta es una técnica valiosa para mantener la lógica del negocio en un sólo lugar: el modelo. Por ejemplo, este modelo tiene algunos métodos personalizados:

```
.. code-block:: javascript

    var Person = type('Person', [ models.Model ], {
        first_name: new models.CharField({ max_length: 50 }),
        last_name: new models.CharField({ max_length: 50 }),
        birth_date: new models.DateField(),
        address: new models.CharField({ max_length: 100 }),
        city: new models.CharField({ max_length: 50 }),

        baby_boomer_status: function() {
            /*Returns the person's baby-boomer status.*/
            if (Date(1945, 8, 1) <= this.birth_date <= Date(1964, 12, 31))
                return "Baby boomer";
            if (this.birth_date < Date(1945, 8, 1))
                return "Pre-boomer";
            return "Post-boomer";
        },

        get full_name() {
            /*Returns the person's full name.*/
            return '%s %s'.subs(this.first_name, this.last_name);
        }
    });
```

El último método en este ejemplo es un *getter* – un atributo implementado por código personalizado. Los getter son un un truco ingenioso agregado en JavaScript 1.6; puedes leer más acerca de ellas en .. Diego cambiar este link http://www.python.org/download/releases/2.2/descrintro/#property.

Existen también un puñado de métodos de modelo que tienen un significado "especial" para JavaScript o Protopy. Estos métodos se describen en las secciones que siguen.

J.1 __str__

`__str__()` es un "método mágico" de Protopy que define lo que debe ser devuelto si llamas a `string()` sobre el objeto. Se usa `string(obj)` en varios lugares, particularmente como el valor mostrado para hacer el render de un objeto y como el valor insertado en un plantilla cuando muestra un objeto. Por eso, siempre se debe retornar un string agradable y legible por humanos en el `__str__` de un objeto. A pesar de que esto no es requerido, es altamente recomendado.

Aquí hay un ejemplo:

```
var Person = type('Person', [ models.Model ], {
    first_name: new models.CharField({ max_length: 50 }),
    last_name: new models.CharField({ max_length: 50 }),

    __str__: function() {
        return '%s %s'.subs(this.first_name, this.last_name);
    }
});
```

J.2 Ejecutando SQL personalizado

Se pueden escribir escribir sentencias SQL personalizadas en métodos personalizados de modelo y métodos a nivel de módulo. El objeto `doff.db.base.connection` representa la conexión actual a la base de datos. Para usarla, se invoca a `connection.cursor()` para obtener un objeto cursor. Después, se llama a `cursor.execute(sql, [params])` para ejecutar la SQL, y `cursor.fetchone()` o `cursor.fetchall()` para devolver las filas resultantes:

```
my_custom_sql: function() {
    require('doff.db.base', 'connection');
    var cursor = connection.cursor()
    cursor.execute("SELECT foo FROM bar WHERE baz =%s", [this.baz]);
    row = cursor.fetchone();
    return row;
}
```

`connection` y `cursor` implementan en su mayor parte la DB-API estándar. Si no estás familiarizado con la DB-API, observa que la sentencia SQL en `cursor.execute()` usa placeholders, `"%s"`, en lugar de agregar los parámetros directamente dentro de la SQL. Si usas esta técnica, la biblioteca subyacente de base de datos automáticamente agregará comillas y secuencias de escape a tus parámetros según sea necesario.

Una nota final: Si todo lo que quieres hacer es usar una cláusula `WHERE` personalizada, puedes usar los argumentos `where`, `tables`, y `params` de la API estándar de búsqueda. Ver Apéndice C.

J.3 Sobreescribiendo los Métodos por omisión del Modelo

Como se explica en el *Referencia de la API de base de datos*, cada modelo obtiene algunos métodos automáticamente – los más notables son `save()` y `delete()`. Estos se pueden sobreescribir para alterar el comportamiento.

Un caso de uso clásico de sobreescritura de los métodos incorporados es cuando se necesita que suceda algo cuando guardas un objeto, por ejemplo:

```
var Blog = type('Blog', [ models.Model ], {
    name: new models.CharField({ maxlength: 100 }),
    tagline: new models.TextField(),

    save: function() {
        do_something();
        super(models.Model, this).save() // Call the "real" save() method.
        do_something_else();
    }
});
```

También se puede evitar el guardado:

```
var Blog = type('Blog', [ models.Model ], {
    name: new models.CharField({ maxlength: 100 }),
    tagline: new models.TextField(),

    save: function() {
        if (this.name == "Yoko Ono's blog")
            return; // Yoko shall never have her own blog!
        else
            super(models.Model, this).save() // Call the "real" save() method
    }
});
```

Parte IX

Referencia de la API de base de datos

La API de base de datos es la otra mitad de la API de modelos discutido en *apendices-doff-modelos*. Una vez definido un modelo, esta *API* es la que se utiliza en todo momento que se necesite acceder a la base de datos. Este apéndice explica todas las varias opciones de acceso a los datos detalladamente.

A lo largo de este apéndice, vamos a hacer referencia a los siguientes modelos, los cuales pueden formar una simple aplicación de blog: .. code-block:: javascript

```
var models = require('doff.db.models.base');

var Blog = type('Blog', [ models.Model ], {  name: new models.CharField({ max_length: 100 }), tagline: new models.TextField(),

    __str__: function() {  return this.name;

    }

});

var Author = type('Author', [ models.Model ], {  name: new models.CharField({ max_length: 50 }), email: new models.EmailField(),

    __str__: function() {  return this.name;

    }

});

var Entry = type('Entry', [ models.Model ], {  blog: new models.ForeignKey(Blog), headline: new models.CharField({ max_length: 255 }), body_text: new models.TextField(), pub_date: new models.DateTimeField(), authors: new models.ManyToManyField(Author),

    __str__: function() {  return this.headline;

    }

});
```

Creando Objetos

Para crear un objeto, se debe crea una instancia de la clase modelo usando argumentos de palabra clave y luego llama a `save()` para grabarlo en la base de datos:

```
>>> require('mysite.blog.models', 'Blog');
>>> var b = new Blog({ name: 'Beatles Blog', tagline: 'All the latest Beatles news.' });
>>> b.save();
```

Esto, ejecuta una sentencia SQL `INSERT`. No se produce el acceso a la base de datos hasta que explícitamente se invoque a `save()`.

El método `save()` no retorna nada.

Para crear un objeto y grabarlo todo en un paso se usa método `create` de el tipo `Manager` que describiremos en breve.

K.1 Qué pasa cuando grabas?

Cuando grabas un objeto, los siguientes pasos son realizados:

1. **Emitir un evento pre_save.** Esto provee una notificación de que un objeto está a punto de ser grabado. Puedes registrar un ***listener*** que será invocado en cuanto esta señal sea emitida.
2. **Pre-procesar los datos.** Se le solicita a cada campo del objeto implementar cualquier modificación automatizada de datos que pudiera necesitar realizar.

   La mayoría de los campos *no* realizan pre-procesamiento – los datos del campo se guardan tal como están. Sólo se usa pre-procesamiento en campos que tienen comportamiento especial, como campos de archivo.
3. **Preparar los datos para la base de datos.** Se le solicita a cada campo que provea su valor actual en un tipo de dato que puede ser grabado en la base de datos.

   La mayoría de los campos no requieren preparación de los datos. Los tipos de datos simples, como enteros y cadenas, están "listos para escribir". Sin embargo, tipo de datos más complejos requieren a menudo alguna modificación. Por ejemplo, `DateFields` usa un objeto `datetime` para almacenar datos. Las bases de datos no almacenan objetos `datetime`, de manera que el valor del campo debe ser convertido en una cadena de fecha que cumpla con la norma ISO correspondiente para la inserción en la base de datos.
4. **Insertar los datos en la base.** Los datos pre-procesados y preparados son entonces incorporados en una sentencia SQL para su inserción en la base de datos.

5. **Emitir un evento post_save.** Como con el evento `pre_save`, este es utilizado para proporcionar notificación de que un objeto ha sido grabado satisfactoriamente.

K.2 Claves primarias autoincrementales

Por conveniencia, a cada modelo se le da una clave primaria autoincremental llamada `id` a menos que explícitamente se especifique `primary_key=True` en el campo (ver la sección titulada *AutoField*).

Si el modelo tiene un `AutoField`, ese valor incrementado automáticamente será calculado y grabado como un atributo del objeto la primera vez que `save()` se llame:

```
>>> var b2 = new Blog({ name: 'Cheddar Talk', tagline: 'Thoughts on cheese.' });
>>> var b2.id;     // Retorna indefinido, porque no tiene ID.
null

>>> b2.save();
>>> b2.id;     // Retorna el ID del objeto.
14
```

No hay forma de saber cual será el valor de un identificador antes que se llame a `save()` esto se debe a que ese valor es calculado por la base de datos.

Si un modelo tiene un `AutoField` pero se quiere definir el identificador de un nuevo objeto explícitamente cuando grabas, solo hay que definirlo explícitamente antes de grabarlo en vez de confiar en la asignación automática de valor del identificador:

```
>>> var b3 = new Blog({ id: 3, name: 'Cheddar Talk', tagline: 'Thoughts on cheese.'} );
>>> b3.id;
3
>>> b3.save();
>>> b3.id;
3
```

Si se asigna manualmente valores de claves primarias autoincrementales ¡No usar un valor de clave primaria que ya existe!. Si se crea un objeto con un valor explícito de clave primaria que ya existe en la base de datos, Se asumirá que se esta cambiando el registro existente en vez de crear uno nuevo.

Dado el ejemplo precedente de blog `'Cheddar Talk'`, este ejemplo sobrescribiría el registro previo en la base de datos:

```
>>> var b4 = new Blog({ id: 3, name: 'Not Cheddar', tagline: 'Anything but cheese.' });
>>> b4.save();  // Sobrescribe el objeto previo con ID=3!
```

El especificar explícitamente valores de claves primarias autoincrementales es más útil cuando se están grabando objetos en lotes, cuando se está seguro de que no se tendrán colisiones de claves primarias.

Grabando cambios de objetos

Para grabar los cambios hechos a un objeto que existe en la base de datos, utilizar `save()`.

Dada la instancia de `Blog` `b5` que ya ha sido grabada en la base de datos, este ejemplo cambia su nombre y actualiza su registro en la base:

```
>>> b5.name = 'New name';
>>> b5.save();
```

Esto ejecuta una sentencia SQL `UPDATE`. De nuevo: No se accede a la base de datos hasta que `save()` es llamado explícitamente.

Como se sabe cuando usar `UPDATE` y cuando usar `INSERT`

Los objetos de base de datos usan el mismo método `save()` para crear y cambiar objetos. La necesidad de usar sentencias SQL `INSERT` o `UPDATE` es abstraida. Específicamente, cuando se llama a `save()`, este es el algoritmo a seguir:

- Si el atributo clave primaria del objeto tiene asignado un valor que evalúa `true` (esto es, un valor distinto a `null` o a la cadena vacía) se ejecuta una consulta `SELECT` para determinar si existe un registro con la clave primaria especificada.
- Si el registro con la clave primaria especificada ya existe, Se ejecuta una consulta `UPDATE`.
- Si el atributo clave primaria del objeto *no* tiene valor o si lo tiene pero no existe un registro, Se ejecuta un `INSERT`.

Debido a esto, se debe tener cuidado de no especificar un valor explícito para una clave primaria cuando se graba nuevos objetos si es que no se puede garantizar que el valor de clave primaria está disponible para ser usado.

La actualización de campos `ForeignKey` funciona exactamente de la misma forma; simplemente se asign un objeto del tipo correcto al campo en cuestión:

```
>>> var joe = Author.objects.create({ name: "Joe" });
>>> entry.author = joe;
>>> entry.save();
```

Recuperando objetos

Se recuperan objetos usando código como el siguiente:

```
>>> blogs = Blog.objects.filter({ author__name__contains: "Joe" });
```

Hay bastantes partes móviles detrás de escena aquí: cuando se recuperan objetos de la base de datos, lo que realmente ocurre es la creación de un `QuerySet` usando el `Manager` del modelo. Este `QuerySet` sabe como ejecutar SQL y retornar los objetos solicitados.

El *apendices-doff-modelos* trató ambos objetos desde el punto de vista de la definición del modelo; ahora vamos a ver cómo funcionan.

Un `QuerySet` representa una colección de objetos de la base de datos. Puede tener cero, uno, o muchos filtros – criterios que limitan la colección basados en parámetros provistos. En términos de SQL un `QuerySet` se compara a una declaración `SELECT` y un filtro es una cláusula de limitación como por ejemplo `WHERE` o `LIMIT`.

Se consigue un `QuerySet` usando el `Manager` del modelo. Cada modelo tiene por lo menos un `Manager` y tiene, por omisión, el nombre `objects`. Se accede al mismo directamente a través de la clase del modelo, así:

```
>>> Blog.objects
<doff.db.models.manager.Manager object>
```

Los `Managers` solo son accesibles a través de las clases de los modelos, en vez desde una instancia de un modelo, para así hacer cumplir con la separación entre las operaciones a "nivel de tabla" y las operaciones a "nivel de registro":

```
>>> var b = new Blog({ name: 'Foo', tagline: 'Bar' });
>>> b.objects;
AttributeError: Manager isn't accessible via Blog instances.
```

El `Manager` es la principal fuente de `QuerySets` para un modelo. Actúa como un `QuerySet` "raíz" que describe todos los objetos de la tabla de base de datos del modelo. Por ejemplo, `Blog.objects` es el `QuerySets` inicial que contiene todos los objetos `Blog` en la base de datos.

Caching y QuerySets

Cada `QuerySet` contiene un cache, para minimizar el acceso a la base de datos. Es importante entender como funciona, para escribir código mas eficiente.

En un `QuerySet` recién creado, el cache esta vacío. La primera vez que un `QuerySet` es evaluado – y, por lo tanto, ocurre un acceso a la base de datos – Se graba el resultado de la consulta en el cache del `QuerySet` y retorna los resultados que han sido solicitados explícitamente (por ejemplo, el siguiente elemento, si se está iterando sobre el `QuerySet`). Evaluaciones subsecuentes del `QuerySet` re-usan los resultados alojados en el cache.

Hay que tener presente este comportamiento de caching, para usar los `QuerySet`s correctamente. Por ejemplo, lo siguiente creará dos `QuerySet`s, los evaluará, y los descartará:

```
print([e.headline for (e in Entry.objects.all())]);
print([e.pub_date for (e in Entry.objects.all())]);
```

Eso significa que la consulta sera ejecutada dos veces en la base de datos, duplicando la carga sobre la misma. También existe una posibilidad de que las dos listas pudieran no incluir los mismos registros de la base de datos, porque se podría haber agregado o borrado un `Entry` durante el pequeñísimo período de tiempo entre ambas peticiones.

Para evitar este problema, simplemente se debe grabar el `QuerySet` y re-usalo:

```
var queryset = Poll.objects.all();
print([p.headline for (p in queryset)]); // Evaluate the query set.
print([p.pub_date for (p in queryset)]); // Reuse the cache from the evaluation.
```

Filtrando objetos

La manera mas simple de recuperar objetos de una tabla es conseguirlos todos. Para hacer esto, usa el método `all()` en un `Manager`:

```
>>> Entry.objects.all();
```

El método `all()` retorna un `QuerySet` de todos los objetos de la base de datos.

Sin embargo, usualmente solo se necesita seleccionar un subconjunto del conjunto completo de objetos. Para crear tal subconjunto, se refina el `QuerySet` inicial, añadiendo condiciones con filtros. Usualmente estos son lo métodos `filter()` o `exclude()`:

```
>>> y2006 = Entry.objects.filter({ pub_date__year: 2006 });
>>> not2006 = Entry.objects.exclude({ pub_date__year: 2006 })
```

Tanto `filter()` como `exclude()` toman argumentos de *patrones de búsqueda*, los cuales se discutirán detalladamente en breve.

Ñ.1 Encadenando filtros

El resultado de refinar un `QuerySet` es otro `QuerySet` así que es posible enlazar refinamientos, por ejemplo:

```
>>> qs = Entry.objects.filter({ headline__startswith: 'What' });
>>> qs = qs.exclude({ pub_date__gte: datetime.datetime.now() });
>>> qs = qs.filter({ pub_date__gte: datetime.datetime(2005, 1, 1) });
```

Esto toma el `QuerySet` inicial de todas las entradas en la base de datos, agrega un filtro, luego una exclusión, y luego otro filtro. El resultado final es un `QuerySet` conteniendo todas las entradas con un título que empieza con "What" que fueron publicadas entre Enero 1, 2005, y el día actual.

Es importante precisar aquí que los `QuerySet` son perezosos – el acto de crear un `QuerySet` no implica ninguna actividad en la base de datos. De hecho, las tres líneas precedentes no hacen *ninguna* llamada a la base de datos; se pueden enlazar/encadenar filtros todo el día y no se ejecutará realmente la consulta hasta que el `QuerySet` sea *evaluado*.

La evaluacion de un `QuerySet` ocurre en cualquiera de las siguientes formas:

- *Iterando*: Un `QuerySet` es iterable, y ejecuta su consulta en la base de datos la primera vez que se itera sobre el. Por ejemplo, el siguiente `QuerySet` no es evaluado hasta que sea iterado sobre él en el bucle `for`:

```
qs = Entry.objects.filter({ pub_date__year: 2006 });
qs = qs.filter({ headline__icontains: "bill" });
for (var e in qs)
    print(e.headline);
```

Esto imprime todos los títulos desde el 2006 que contienen "bill" pero genera solo un acceso a la base de datos.

- *Rebanado*: Según lo explicado en la próxima sección *Limitando QuerySets*, un `QuerySet` puede ser rebanado usando la sintaxis de rebanado de arreglos de Javascript. Usualmente el rebanar un `QuerySet` retorna otro `QuerySet` (no evaluado), pero se ejecutará la consulta a la base de datos si se usa el parámetro "step" de la sintaxis de rebanado.
- *Convirtiendo a un arreglo*: Puedes forzar la evaluación de un `QuerySet` ejecutando `array()` sobre el mismo, por ejemplo:

```
>>> entry_list = array(Entry.objects.all());
```

Sin embargo, se adveierte de que esto podría significar un gran impacto en la memoria ya que se cargará cada elemento de la lista en memoria. En cambio, el iterar sobre un `QuerySet` se saca ventaja de la base de datos para cargar datos e inicializar objetos solo a medida que se van necesitando los mismos.

Los QuerySets filtrados son únicos

Cada vez que se refina un `QuerySet` se obtiene un nuevo `QuerySet` que no está de ninguna manera atado al *QuerySet'* anterior. Cada refinamiento crea un `QuerySet` separado y distinto que puede ser almacenado, usado y re-usado:

```
q1 = Entry.objects.filter({ headline__startswith: "What" });
q2 = q1.exclude({ pub_date__gte: datetime.now() });
q3 = q1.filter({ pub_date__gte: datetime.now() });
```

Estos tres `QuerySets` son separados. El primero es un `QuerySet` base que contiene todas las entradas que contienen un título que empieza con "What". El segundo es un sub-conjunto del primero, con un criterio adicional que excluye los registros cuyo `pub_date` es mayor que el día de hoy. El tercero es un sub-conjunto del primero, con un criterio adicional que selecciona solo los registros cuyo `pub_date` es mayor que el día de hoy. El `QuerySet` inicial (`q1`) no es afectado por el proceso de refinamiento.

Ñ.2 Limitando QuerySets

Se usa la sintaxis de rebanado de arreglos de JavaScript para limitar los `QuerySet` a un cierto número de resultados. Esto es equivalente a las clausulas de SQL de `LIMIT` y `OFFSET`.

Por ejemplo, esto retorna las primeras cinco entradas (`LIMIT 5`):

```
>>> Entry.objects.all().slice(0, 5);
```

Esto retorna las entradas desde la sexta hasta la décima (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all().slice(5, 10);
```

Generalmente, el rebanar un `QuerySet` retorna un nuevo `QuerySet` – no evalúa la consulta. Una excepción es si se usa el parámetro "step" de la sintaxis de rebanado. Por ejemplo, esto realmente ejecutaría la consulta con el objetivo de retornar una lista, objeto de por medio de los primeros diez:

```
>>> Entry.objects.all().slice(0, 10, 2);
```

Para recuperar *un* solo objeto en vez de una lista (por ej. `SELECT foo FROM bar LIMIT 1`) se usa un simple índice en vez de un rebanado. Por ejemplo, esto retorna el primer `Entry` en la base de datos, después de ordenar las entradas alfabéticamente por título:

```
>>> Entry.objects.order_by('headline').get(0);
```

Ñ.3 Métodos de consulta que retornan nuevos QuerySets

Se provee una variedad de métodos de refinamiento de `QuerySet` que modifican ya sea los tipos de resultados retornados por el `QuerySet` o la forma como se ejecuta la consulta SQL. Estos métodos se describen en las secciones que siguen. Algunos de estos métodos reciben argumentos de patrones de búsqueda, los cuales se discuten en detalle mas adelante.

Ñ.3.1 filter(**lookup)

Retorna un nuevo `QuerySet` conteniendo objetos que son iguales a los parámetros de búsqueda provistos.

Ñ.3.2 exclude(**kwargs)

Retorna un nuevo `QuerySet` conteniendo objetos que *no* son iguales a los parámetros de búsqueda provistos.

Ñ.3.3 order_by(*campos)

Por omisión, los resultados retornados por un `QuerySet` están ordenados por la tupla de ordenamiento indicada por la opción `ordering` en los metadatos del modelo (ver *apendices-doff-modelos*). Se puede sobrescribir esto para una consulta particular usando el método `order_by()`:

```
>>> Entry.objects.filter({ pub_date__year: 2005 }).order_by('-pub_date', 'headline');
```

Este resultado será ordenado por `pub_date` de forma descendente, luego por `headline` de forma ascendente. El signo negativo en frente de `"-pub_date"` indica orden *descendente*. Si el – esta ausente se asume un orden ascendente. Para ordenar aleatoriamente, `"?"`, así:

```
>>> Entry.objects.order_by('?');
```

Ñ.3.4 distinct()

Retorna un nuevo `QuerySet` que usa `SELECT DISTINCT` en su consulta SQL. Esto elimina filas duplicadas en el resultado de la misma.

Por omisión, un `QuerySet` no eliminará filas duplicadas. En la práctica esto raramente es un problema porque consultas simples como `Blog.objects.all()` no introducen la posibilidad de registros duplicados.

Sin embargo, si tu consulta abarca múltiples tablas, es posible obtener resultados duplicados cuando un `QuerySet` es evaluado. Esos son los casos en los que se usaría `distinct()`.

Ñ.3.5 values(*campos)

Retorna un `QuerySet` especial que evalúa a una lista de diccionarios en lugar de objetos instancia de modelo. Cada uno de esos diccionarios representa un objeto, con las las claves en correspondencia con los nombre de los atributos de los objetos modelo:

```
// This list contains a Blog object.
>>> Blog.objects.filter({ name__startswith: 'Beatles' });
[Beatles Blog]

// This list contains a dictionary.
>>> Blog.objects.filter({ name__startswith: 'Beatles' }).values();
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]
```

`values()` puede recibir argumentos posicionales opcionales, `*campos`, los cuales especifican los nombres de campos a los cuales debe limitarse el `SELECT`. Si se especifican los campos, cada diccionario contendrá solamente las claves/valores de campos para los campos que especifiques. Si no se especifican los campos, cada diccionario contendrá una clave y un valor para todos los campos en la table de base de datos:

```
>>> Blog.objects.values();
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}],
>>> Blog.objects.values('id', 'name');
[{'id': 1, 'name': 'Beatles Blog'}]
```

Este método es útil cuando se sabe de antemano que solo se va a necesitar valores de un pequeño número de los campos disponibles y no necesitarás la funcionalidad de un objeto instancia de modelo. Es más eficiente el seleccionar solamente los campos que se necesitan usar. .. Diego hasta aca dates(campo, tipo, orden) ~~~~~~~~~~~~~~~~~~~~~~~~~~

Retorna un `QuerySet` especial que evalúa a una lista de objetos `datetime.datetime` que representan todas las fechas disponibles de un cierto tipo en el contenido de la `QuerySet`.

El argumento `campo` debe ser el nombre de un `DateField` o de un `DateTimeField` de tu modelo. El argumento `tipo` debe ser ya sea `year`, `month` o `day`. Cada objeto `datetime.datetime` en la lista de resultados es truncado de acuerdo al `tipo` provisto:

- `"year"` retorna una lista de todos los valores de años distintos entre sí para el campo.
- `"month"` retorna una lista de todos los valores de años/mes distintos entre sí para el campo.
- `"day"` retorna una lista de todos los valores de años/mes/día distintos entre sí para el campo.

`orden`, cuyo valor por omisión es `'ASC'`, debe ser `'ASC'` o `'DESC'`. El mismo especifica cómo ordenar los resultados.

Aquí tenemos algunos ejemplos:

```
>>> Entry.objects.dates('pub_date', 'year')
[datetime.datetime(2005, 1, 1)]

>>> Entry.objects.dates('pub_date', 'month')
[datetime.datetime(2005, 2, 1), datetime.datetime(2005, 3, 1)]

>>> Entry.objects.dates('pub_date', 'day')
[datetime.datetime(2005, 2, 20), datetime.datetime(2005, 3, 20)]

>>> Entry.objects.dates('pub_date', 'day', order='DESC')
[datetime.datetime(2005, 3, 20), datetime.datetime(2005, 2, 20)]

>>> Entry.objects.filter(headline__contains='Lennon').dates('pub_date', 'day')
[datetime.datetime(2005, 3, 20)]
```

Ñ.3.6 select_related()

Retorna un `QuerySet` que seguirá automáticamente relaciones de clave foránea, seleccionando esos datos adicionales de objetos relacionados cuando ejecuta su consulta. Esto contribuye a la mejora de rendimiento que resulta en consultas (aveces mucho) más grandes pero significan que el uso posterior de relaciones de clave foránea no requerirán consultas a la base de datos.

Los siguientes ejemplos ilustran la diferencia entre búsquedas normales y búsquedas `select_related()`. Esta es una búsqueda normal:

```
# Hits the database.
>>> e = Entry.objects.get(id=5)

# Hits the database again to get the related Blog object.
>>> b = e.blog
```

Esta es una búsqueda `select_related`:

```
# Hits the database.
>>> e = Entry.objects.select_related().get(id=5)

# Doesn't hit the database, because e.blog has been prepopulated
# in the previous query.
>>> b = e.blog
```

`select_related()` sigue claves foráneas tan lejos como le sea posible. Si tienes los siguientes modelos:

```
class City(models.Model):
    # ...

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(City)

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person)
```

entonces una llamada a `Book.objects.select_related().get(id=4)` colocará en el cache la `Person` relacionada *y* la `City` relacionada:

```
>>> b = Book.objects.select_related().get(id=4)
>>> p = b.author         # Doesn't hit the database.
>>> c = p.hometown       # Doesn't hit the database.

>>> b = Book.objects.get(id=4) # No select_related() in this example.
>>> p = b.author         # Hits the database.
>>> c = p.hometown       # Hits the database.
```

Notar que `select_related` no sigue claves foráneas que tienen `null=True`.

Usualmente, el usar `select_related()` puede mejorar muchísimo el desempeño porque tu aplicación puede puede entonces evitar muchas llamadas a la base de datos. Sin embargo, en siuaciones con conjuntos de relaciones profundamente anidadas, `select_related()` puede en algunos casos terminar siguiendo "demasiadas" relaciones y puede generar consultas tan grandes que terminan siendo lentas.

Ñ.3.7 extra()

A veces, el lenguaje de consulta de Django no puede expresar facilmente cláusulas `WHERE` complejas. Para estos casos extremos, Django provee un modificador de `QuerySet` llamado `extra()` – una forma de inyectar cláusulas especificas dentro del SQL generado por un `QuerySet`.

Por definición, estas consultas especiales pueden no ser portables entre los distintos motores de bases de datos (debido a que estás escribiendo código SQL explícito) y violan el principio DRY, así que deberías evitarlas de ser posible.

Se puede especificar uno o más de `params`, `select`, `where`, o `tables`. Ninguno de los argumentos es obligatorio, pero deberías indicar al menos uno.

El argumento `select` permite indicar campos adicionales en una cláusula de `SELECT`. Debe contener un diccionario que mapee nombres de atributo a cláusulas SQL que se utilizarán para calcular el atributo en cuestión:

```
>>> Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
```

Como resultado, cada objeto `Entry` tendrá en este caso un atributo adicional, `is_recent`, un booleano que representará si el atributo `pub_date` del entry es mayor que el 1 de Enero de 2006.

El siguiente ejemplo es más avanzado; realiza una subconsulta para darle a cada objeto `Blog` resultante un atributo `entry_count`, un entero que indica la cantidad de objetos `Entry` asociados al blog:

```
>>> subq = 'SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id'
>>> Blog.objects.extra(select={'entry_count': subq})
```

(En este caso en particular, estamos aprovechando el hecho de que la consulta ya contiene la tabla `blog_blog` en su cláusula `FROM`.)

También es posible definir cláusulas `WHERE` explícitas – quizás para realizar joins implícitos – usando el argumento `where`. Se puede agregar tablas manualmente a la cláusula `FROM` del SQL usando el argumento `tables`.

Tanto `where` como `tables` reciben una lista de cadenas. Todos los argumentos de `where` son unidos con AND a cualquier otro criterio de búsqueda:

```
>>> Entry.objects.extra(where=['id IN (3, 4, 5, 20)'])
```

Los parámetros `select` y `where` antes descriptos pueden utilizar los comodines normales para bases de datos en Python: `'%s'` para indicar parámetros que deberían ser escapados automáticamente por el motor de la base de datos. El argumento `params` es una lista de los parámetros que serán utilizados para realizar la sustitución:

```
>>> Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
```

Siempre se debe utilizar `params` en vez de utilizar valores directamente en `select` o `where` ya que `params` asegura que los valores serán escapados correctamente de acuerdo con tu motor de base de datos particular.

Este es un ejemplo de lo que está incorrecto:

```
Entry.objects.extra(where=["headline='%s'" % name])
```

Este es un ejemplo de lo que es correcto:

```
Entry.objects.extra(where=['headline=%s'], params=[name])
```

Ñ.4 Metodos de `QuerySet` que no devuelven un `QuerySet`

Los métodos de `QuerySet` que se describen a continuación evaluan el `QuerySet` y devuelven algo *que no es* un `QuerySet` – un objeto, un valor, o algo así.

Ñ.4.1 get(**lookup)

Devuelve el objeto que matchee el parámetro de búsqueda provisto. El parámetro debe proveerse de la manera descripta en la sección "Patrones de búsqueda". Este método levanta `AssertionError` si más de un objecto concuerda con el patrón provisto.

Si no se encuentra ningún objeto que coincida con el patrón de búsqueda provisto `get()` levanta una excepción de `DoesNotExist`. Esta excepción es un atributo de la clase del modelo, por ejemplo:

```
>>> Entry.objects.get(id='foo') # levanta Entry.DoesNotExist
```

La excepción `DoesNotExist` hereda de `django.core.exceptions.ObjectDoesNotExist`, así que puedes protegerte de múltiples excepciones `DoesNotExist`:

```
>>> from django.core.exceptions import ObjectDoesNotExist
>>> try:
...     e = Entry.objects.get(id=3)
...     b = Blog.objects.get(id=1)
... except ObjectDoesNotExist:
...     print "Either the entry or blog doesn't exist."
```

Ñ.4.2 create(**kwargs)

Este método sirve para crear un objeto y guardarlo en un mismo paso. Te permite abreviar dos pasos comunes:

```
>>> p = Person(first_name="Bruce", last_name="Springsteen")
>>> p.save()
```

en una sola línea:

```
>>> p = Person.objects.create(first_name="Bruce", last_name="Springsteen")
```

Ñ.4.3 get_or_create(**kwargs)

Este método sirve para buscar un objeto y crearlo si no existe. Devuelve una tupla `(object, created)`, donde `object` es el objecto encontrado o creado, y `created` es un booleano que indica si el objeto fue creado.

Está pensado como un atajo para el caso de uso típico y es más que nada útil para scripts de importación de datos, por ejemplo:

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lennon')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday=date(1940, 10, 9))
    obj.save()
```

Este patrón se vuelve inmanejable a medida que aumenta el número de campos en el modelo. El ejemplo anterior puede ser escrito usando `get_or_create` así:

```
obj, created = Person.objects.get_or_create(
    first_name = 'John',
    last_name  = 'Lennon',
    defaults   = {'birthday': date(1940, 10, 9)}
)
```

Cualquier argumento que se le pase a `get_or_create()` – *excepto* el argumento opcional `defaults` – será utilizado en una llamada a `get()`. Si se encuentra un objecto, `get_or_create` devolverá una tupla con ese objeto y `False`. Si *no* se encuentra un objeto, `get_or_create()` instanciará y guardará un objeto nuevo, devolviendo una tupla con el nuevo objeto y `True`. El nuevo objeto será creado de acuerdo con el siguiente algoritmo:

```
defaults = kwargs.pop('defaults', {})
params = dict([(k, v) for k, v in kwargs.items() if '__' not in k])
params.update(defaults)
obj = self.model(**params)
obj.save()
```

Esto es, se comienza con los argumentos que no sean `'defaults'` y que no contengan doble guión bajo (lo cual indicaría una búsqueda no exacta). Luego se le agrega el contenido de `defaults`, sobreescribiendo cualquier valor que ya estuviera asignado, y se usa el resultado como claves para el constructor del modelo.

Si el modelo tiene un campo llamado `defaults` y es necesario usarlo para una búsqueda exacta en `get_or_create()`, simplemente hay que utilizar `'defaults__exact'` así:

```
Foo.objects.get_or_create(
    defaults__exact = 'bar',
    defaults={'defaults': 'baz'}
)
```

Ñ.4.4 count()

Devuelve un entero representando el número de objetos en la base de datos que coincidan con el `QuerySet`. `count()` nunca levanta excepciones. He aquí un ejemplo:

```
# Returns the total number of entries in the database.
>>> Entry.objects.count()
4
```

```
# Returns the number of entries whose headline contains 'Lennon'
>>> Entry.objects.filter(headline__contains='Lennon').count()
1
```

`count()` en el fondo realiza un `SELECT COUNT(*)`, así que deberías siempre utilizar `count()` en vez de cargar todos los registros en objetos Python y luego invocar `len()` sobre el resultado.

Dependiendo de la base de datos que estés utilizando (e.g., PostgreSQL o MySQL), `count()` podría devolver un entero largo en vez de un entero normal de Python. Esto es una característica particular de la implementación subyacente que no debería ser ningún problema en la vida real.

Ñ.4.5 in_bulk(id_list)

Este método toma una lista de claves primarias y devuelve un diccionario que mapea cada clave primaria en una instancia con el ID dado, por ejemplo:

```
>>> Blog.objects.in_bulk([1])
{1: Beatles Blog}
>>> Blog.objects.in_bulk([1, 2])
{1: Beatles Blog, 2: Cheddar Talk}
>>> Blog.objects.in_bulk([])
{}
```

Si no se encuentra un objeto en la base para un ID en particular, este id no aparecerá en el diccionario resultante. Si le pasas una lista vacía a `in_bulk()`, obtendrás un diccionario vacío.

Ñ.4.6 latest(field_name=None)

Devuelve el último objeto de la tabla, ordenados por fecha, utilizando el campo que se provea en el argumento `field_name` como fecha. Este ejemplo devuelve el `Entry` más reciente en la tabla, de acuerdo con el campo `pub_date`:

```
>>> Entry.objects.latest('pub_date')
```

Si el `Meta` de tu modelo especifica `get_latest_by`, se puede omitir el argumento `field_name`. Django utilizará el campo indicado en `get_latest_by` por defecto.

Al igual que `get()`, `latest()` levanta `DoesNotExist` si no existe un objeto con los parámetros provistos.

Patrones de búsqueda

Los patrones de búsqueda son la manera en que se especifica la carne de una cláusula `WHERE` de SQL. Consisten de argumentos de palabra clave para los métodos `filter()`, `exclude()` y `get()` de `QuerySet`.

Los parámetros básicos de búsqueda toman la forma de `campo__tipodebusqueda=valor` (notar el doble guión bajo). Por ejemplo:

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

se traduce (aproximadamente) al siguiente comando SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

Si se suministra un argumento de palabra clave inválido, la función levantará una excepción de `TypeError`.

A continuación se listan los tipos de búsqueda que existen.

O.1 exact

Realiza una búsqueda por coincidencias exactas:

```
>>> Entry.objects.get(headline__exact="Man bites dog")
```

Esto busca objetos que tengan en el campo headline la frase exacta "Man bites dog".

Si no se suministra un tipo de búsqueda – O sea, si tu argumento de palabra clave no contiene un doble guión bajo – el tipo de búsqueda se asume como `exact`.

Por ejemplo, las siguientes dos sentencias son equivalentes:

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
```

Esto es por conveniencia, dado que las búsquedas con tipo de búsqueda `exact` son las más frecuentes.

O.2 iexact

Realiza una búsqueda por coincidencias exactas sin distinguir mayúsculas de minúsculas:

```
>>> Blog.objects.get(name__iexact='beatles blog')
```

Traerá objetos con nombre `'Beatles Blog'`, `'beatles blog'`, `'BeAtLes BLoG'`, etcétera.

O.3 contains

Realiza una búsqueda de subcadenas, distinguiendo mayúsculas y minúsculas:

```
Entry.objects.get(headline__contains='Lennon')
```

Esto coincidirá con el titular `'Today Lennon honored'` pero no con `'today lennon honored'`.

SQLite no admite sentencias `LIKE` distinguiendo mayúsculas y minúsculas; cuando se utiliza SQLite, `contains` se comporta como `icontains`.

O.4 icontains

Realiza una búsqueda de subcadenas, sin distinguir mayúsculas y minúsculas:

```
>>> Entry.objects.get(headline__icontains='Lennon')
```

A diferencia de `contains`, `icontains` *sí* trerá `today lennon honored`.

O.5 gt, gte, lt, and lte

Estos representan los operadores de mayor a, mayor o igual a, menor a, y menor o igual a, respectivamente:

```
>>> Entry.objects.filter(id__gt=4)
>>> Entry.objects.filter(id__lt=15)
>>> Entry.objects.filter(id__gte=0)
```

Estas consultas devuelven cualquier objeto con un ID mayor a 4, un ID menor a 15, y un ID mayor o igual a 1, respectivamente.

Por lo general estos operadores se utilizarán con campos numéricos. Se debe tener cuidado con los campos de caracteres, ya que el orden no siempre es el que uno se esperaría (i.e., la cadena "4" resulta ser *mayor* que la cadena "10").

O.6 in

Aplica un filtro para encontrar valores en una lista dada:

```
Entry.objects.filter(id__in=[1, 3, 4])
```

Esto devolverá todos los objetos que tengan un ID de 1, 3 o 4.

O.7 startswith

Busca coincidencias de prefijos distinguiendo mayúsculas y minúsculas:

```
>>> Entry.objects.filter(headline__startswith='Will')
```

Esto encontrará los titulares "Will he run?" y "Willbur named judge", pero no "Who is Will?" o "will found in crypt".

O.8 istartswith

Realiza una búsqueda por prefijos, sin distinguir mayúsculas y minúsculas:

```
>>> Entry.objects.filter(headline__istartswith='will')
```

Esto devolverá los titulares "Will he run?", "Willbur named judge", y "will found in crypt", pero no "Who is Will?"

O.9 endswith and iendswith

Realiza búsqueda de sufijos, distinguiendo y sin distinguir mayúsculas de minúsculas, respectivamente:

```
>>> Entry.objects.filter(headline__endswith='cats')
>>> Entry.objects.filter(headline__iendswith='cats')
```

O.10 range

Realiza una búsqueda por rango:

```
>>> start_date = datetime.date(2005, 1, 1)
>>> end_date = datetime.date(2005, 3, 31)
>>> Entry.objects.filter(pub_date__range=(start_date, end_date))
```

Se puede utilizar `range` en cualquier lugar donde podrías utilizar `BETWEEN` en SQL – para fechas, números, e incluso cadenas de caracteres.

O.11 year, month, and day

Para campos `date` y `datetime`, realiza búsqueda exacta por año, mes o día:

```
# Búsqueda por año
>>>Entry.objects.filter(pub_date__year=2005)

# Búsqueda por mes -- toma enteros
>>> Entry.objects.filter(pub_date__month=12)
```

```
# Búsqueda por día
>>> Entry.objects.filter(pub_date__day=3)

# Combinación: devuelve todas las entradas de Navidad de cualquier año
>>> Entry.objects.filter(pub_date__month=12, pub_date_day=25)
```

O.12 isnull

Toma valores `True` o `False`, que corresponderán a consultas SQL de `IS NULL``y ``IS NOT NULL`, respectivamente:

```
>>> Entry.objects.filter(pub_date__isnull=True)
```

O.13 search

Un booleano que realiza búsquedas `full-text`, que aprovecha el indexado `full-text`. Esto es como `contains` pero significativamente más rápido debido al indexado `full-text`.

Nótese que este tipo de búsqueda sólo está disponible en MySQL y requiere de manipulación directa de la base de datos para agregar el índice `full-text`.

O.14 El patrón de búsqueda pk

Por conveniencia, Django provee un patrón de búsqueda `pk`, que realiza una búsqueda sobre la clave primaria del modelo (`pk` por `primary key`, del inglés).

En el modelo de ejemplo `Blog`, la clave primaria es el campo `id`, así que estas sentencias serían equivalentes:

```
>>> Blog.objects.get(id__exact=14) # Forma explícita
>>> Blog.objects.get(id=14) # __exact implícito
>>> Blog.objects.get(pk=14) # pk implica id__exact
```

El uso de `pk` no se limita a búsquedas `__exact` – cualquier patrón de búsqueda puede ser combinado con `pk` para realizar una búsqueda sobre la clave primaria de un modelo:

```
# Buscar entradas en blogs con id 1, 4, o 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Buscar entradas en blogs con id > 14
>>> Blog.objects.filter(pk__gt=14)
```

Las búsquedas `pk` también funcionan con joins. Por ejemplo, estas tres sentencias son equivalentes:

```
>>> Entry.objects.filter(blog__id__exact=3) # Forma explícita
>>> Entry.objects.filter(blog__id=3) # __exact implícito
>>> Entry.objects.filter(blog__pk=3) # __pk implica __id__exact
```

Búsquedas complejas con Objetos Q

Los argumentos de palabras clave en las búsquedas – en `filter()` por ejemplo – son unidos con AND. Si necesitas realizar búsquedas más complejas (e.g., búsquedas con sentencias `OR`), puedes utilizar objetos `Q`.

Un objeto `Q` (`django.db.models.Q`) es un objeto que se utiliza para encapsular una colección de argumentos de palabra clave. Estos argumentos de palabra clave son especificados como se indica en la sección "Patrones de búsqueda".

Por ejemplo, este objeto `Q` encapsula una consulta con un único `LIKE`:

    Q(question__startswith='What')

Los objetos `Q` pueden ser combinados utilizando los operadores `&` y `|`. Cuando se utiliza un operador sobre dos objetos, se obtiene un nuevo objeto `Q`. Por ejemplo, un `OR` de dos consultas `question__startswith` sería:

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

Esto será equivalente a la siguiente cláusula `WHERE` en SQL:

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

Puede componer sentencias de complejidad arbitraria combinando objetos `Q` con los operadores `&` y `|`. También se pueden utilizar paréntesis para agrupar.

Cualquier función de búsqueda que tome argumentos de palabra clave (e.g., `filter()`, `exclude()`, `get()`) puede recibir también uno o más objetos `Q` como argumento posicional (no nombrado). Si se proveen multiples objetos `Q` como argumentos a una función de búsqueda, los argumentos serán unidos con AND, por ejemplo:

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

se traduce aproximadamente al siguiente SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Las funciones de búsqueda pueden además mezclar el uso de objetos `Q` y de argumentos de palabra clave. Todos los argumentos provistos a una función de búsqueda (sean argumentos de palabra clave u objetos `Q`) son unidos con AND. Sin embargo, si se provee un objeto `Q` debe preceder la definición de todos los argumentos de palabra clave. Por ejemplo, lo siguiente:

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who')
```

es una consulta válida, equivalente al ejemplo anterior, pero esto:

```
# CONSULTA INVALIDA
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))
```

no es válido.

Hay algunos ejemplos disponibles online en http://www.djangoproject.com/documentation/0.96/models/or_lookups/ .

Objetos Relacionados

Cuando defines una relación en un modelo (i.e. un `ForeignKey`, `OneToOneField`, or `ManyToManyField`), las instancias de ese modelo tendrán una API conveniente para acceder a estos objetos relacionados.

Por ejemplo, si `e` es un objeto `Entry`, puede acceder a su `Blog` asociado accediendo al atributo `blog`, esto es `e.blog`.

Django también crea una API para acceder al "otro" lado de la relación – el vínculo del modelo relacionado al modelo que define la relación. Por ejemplo, si `b` es un objeto `Blog`, tiene acceso a la lista de todos los objetos `Entry` a través del atributo `entry_set`: `b.entry_set.all()`.

Todos los ejemplos en esta sección utilizan los modelos de ejemplo `Blog`, `Author` y `Entry` que se definen al principio de esta sección.

Q.1 Consultas Que Cruzan Relaciones

Django ofrece un mecanismo poderoso e intuitivo para "seguir" relaciones cuando se realizan búsquedas, haciéndose cargo de los `JOIN`s de SQL de manera automática. Para cruzar una relación simplemente hace falta utilizar el nombre de campo de los campos relacionados entre modelos, separados por dos guiones bajos, hasta que llegues al campo que necesitabas.

Este ejemplo busca todos los objetos `Entry` que tengan un `Blog` cuyo `nombre` sea `'Beatles Blog'`:

```
>>> Entry.objects.filter(blog__name__exact='Beatles Blog')
```

Este camino puede ser tan largo como quieras.

También Funciona en la otra dirección. Para referirse a una relación "inversa", simplemente hay que utilizar el nombre en minúsculas del modelo.

Este ejemplo busca todos los objetos `Blog` que tengan al menos un `Entry` cuyo `headline` contenga `'Lennon'`:

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

Q.2 Relaciones de Clave Foránea

Si un modelo contiene un `ForeignKey`, las instancias de ese modelo tendrán acceso al objeto relacionado (foráneo) vía un simple atributo del modelo, por ejemplo:

```
e = Entry.objects.get(id=2)
e.blog # Devuelve el objeto Blog relacionado
```

Se puede acceder y asignar el valor de la clave foránea vía el atributo. Como sería de esperar, los cambios a la clave foránea no se guardan en el modelo hasta que invoques el método `save()`, por ejemplo:

```
e = Entry.objects.get(id=2)
e.blog = some_blog
e.save()
```

Si un campo `ForeignKey` tiene la opción `null=True` seteada (i.e. permite valores `NULL`), se le puede asignar `None`:

```
e = Entry.objects.get(id=2)
e.blog = None
e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

El acceso a relaciones uno-a-muchos se almacena la primera vez que se accede al objeto relacionado. Cualquier acceso subsiguiente a la clave foránea del mismo objeto son cacheadas, por ejemplo:

```
e = Entry.objects.get(id=2)
print e.blog  # Busca el Blog asociado en la base de datos.
print e.blog  # No va a la base de datos; usa la versión cacheada.
```

Notar que el método de `QuerySet` `select_related()` busca inmediatamente todos los objetos de relaciones uno-a-muchos de la instancia:

```
e = Entry.objects.select_related().get(id=2)
print e.blog  # No va a la base de datos; usa la versión cacheada.
print e.blog  # No va a la base de datos; usa la versión cacheada.
```

`select_related()` está documentada en la sección "Métodos de consulta que retornan nuevos QuerySets".

Q.3 Relaciones de Clave Foreánea "Inversas"

Las relaciones de clave foránea son automáticamente simétricas – se infiere una relación inversa de la presencia de un campo `ForeignKey` que apunte a otro modelo.

Si un modelo tiene una `ForeignKey`, las instancias del modelo de la clave foránea tendrán acceso a un `Manager` que devuelve todas las instancias del primer modelo. Por defecto, este `Manager` se llama `FOO_set`, donde `FOO` es el nombre modelo que contiene la clave foránea, todo en minúsculas. Este `Manager` devuelve `QuerySets`, que pueden ser filtradas y manipuladas como se describe en la sección "Recuperando objetos".

Aquí se muestra un ejemplo:

```
b = Blog.objects.get(id=1)
b.entry_set.all() # Encontrar todos los objetos Entry relacionados a b.

# b.entry_set es un Manager que devuelve QuerySets.
```

```
b.entry_set.filter(headline__contains='Lennon')
b.entry_set.count()
```

Se puede cambiar el nombre del atributo `FOO_set` indicando el parámetro `related_name` en la definición del `ForeignKey()`. Por ejemplo, si el modelo `Entry` fuera cambiado por `blog = ForeignKey(Blog, related_name='entries')`, el ejemplo anterior pasaría a ser así:

```
b = Blog.objects.get(id=1)
b.entries.all() # Encontrar todos los objetos Entry relacionados a b.

# b.entries es un Manager que devuelve QuerySets.
b.entries.filter(headline__contains='Lennon')
b.entries.count()
```

No se puede acceder al `Manager` de `ForeignKey` inverso desde la clase misma; debe ser accedido desde una instancia:

```
Blog.entry_set # Raises AttributeError: "Manager must be accessed via instance".
```

Además de los metodos de `QuerySet` definidos en la sección "Recuperando Objetos", el `Manager` de `ForeignKey` tiene los siguientes métodos adicionales:

- `add(obj1, obj2, ...)`: Agrega los objetos del modelo indicado al conjunto de objetos relacionados, por ejemplo:

  ```
  b = Blog.objects.get(id=1)
  e = Entry.objects.get(id=234)
  b.entry_set.add(e) # Associates Entry e with Blog b.
  ```

- `create(**kwargs)`: Crea un nuevo objeto, lo guarda, y lo deja en el conjunto de objetos relacionados. Devuelve el objeto recién creado:

  ```
  b = Blog.objects.get(id=1)
  e = b.entry_set.create(headline='Hello', body_text='Hi', pub_date=datetime.date(2005, 1, 1))
  # No hace falta llamar a e.save() acá -- ya ha sido guardado
  ```

  Esto es equivalente a (pero más simple que) lo siguiente:

  ```
  b = Blog.objects.get(id=1)
  e = Entry(blog=b, headline='Hello', body_text='Hi', pub_date=datetime.date(2005, 1, 1))
  e.save()
  ```

  Notar que no es necesario especificar el argumento de palabra clave correspondiente al modelo que define la relación. En el ejemplo anterior, no le pasamos el parámetro `blog` a `create()`. Django deduce que el campo `blog` del nuevo `Entry` debería ser `b`.

- `remove(obj1, obj2, ...)`: Quita los objetos indicados del conjunto de objetos relacionados:

  ```
  b = Blog.objects.get(id=1)
  e = Entry.objects.get(id=234)
  b.entry_set.remove(e) # Desasociar al Entry e del Blog b.
  ```

  Para evitar inconsistencias en la base de datos, este método sólo existe para objetos `ForeignKey` donde `null=True`. Si el campo relacionado no puede pasar ser `None` (`NULL`), entonces un objeto no puede ser quitado de una relación sin ser agregado a otra. En el ejemplo anterior, el quitar a `e` de `b.entry_set()` es

equivalente a hacer `e.blog = None`, y dado que la definición del campo `ForeignKey` `blog` (en el modelo `Entry`) no indica `null=True`, esto es una acción inválida.

- `clear()`: Quita todos los objetos del conjunto de objetos relacionados:

  ```
  b = Blog.objects.get(id=1)
  b.entry_set.clear()
  ```

  Notar que esto no borra los objetos relacionados – simplemente los desasocia.

  Al igual que `remove()`, `clear` solo está disponible para campos `ForeignKey` donde `null=True`.

Para asignar todos los miembros de un conjunto relacionado en un solo paso, simplemente se le asigna al conjunto un objeto iterable, por ejemplo:

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

Si el método `clear()` está definido, todos los objetos pre-existentes serán quitados del `entry_set` antes de que todos los objetos en el iterable (en este caso, la lista) sean agregados al conjunto. Si el método `clear()` *no* está disponible, todos los objetos del iterable son agregados al conjunto sin quitar antes los objetos pre-existentes.

Todas las operaciones "inversas" definidas en esta sección tienen efectos inmediatos en la base de datos. Toda creación, borradura y agregado son inmediata y automáticamente grabados en la base de datos.

Q.4 Relaciones muchos-a-muchos

Ambos extremos de las relaciones muchos-a-muchos obtienen una API de acceso automáticamente. La API funciona igual que las funciones "inversas" de las relaciones uno-a-muchos (descriptas en la sección anterior).

La única diferencia es el nombrado de los atributos: el modelo que define el campo `ManyToManyField` usa el nombre del atributo del campo mismo, mientras que el modelo "inverso" utiliza el nombre del modelo original, en minúsculas, con el sufijo `'_set'` (tal como lo hacen las relaciones uno-a-muchos).

Un ejemplo de esto lo hará más fácil de entender:

```
e = Entry.objects.get(id=3)
e.authors.all() # Devuelve todos los objetos Author para este Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Devuelve todos los obejtos Entry para este Author.
```

Al igual que los campos `ForeignKey`, los `ManyToManyField` pueden indicar un `related_name`. En el ejemplo anterior, si el campo `ManyToManyField` en el modelo `Entry` indicara `related_name='entries'`, cualquier instancia de `Author` tendría un atributo `entries` en vez de `entry_set`.

Q.5 Consultas que Abarcan Objetos Relacionados

Las consultas que involucran objetos relacionados siguen las mismas reglas que las consultas que involucran campos normales. Cuando se indica el valor que se requiere en una búsqueda, se puede utilizar tanto una instancia del modelo o bien el valor de la clave primaria del objeto.

Por ejemplo, si `b` es un objeto `Blog` con `id=5`, las tres siguientes consultas son idénticas:

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

Borrando Objectos

El métodos para borrar se llama `delete()`. Este método inmediatamente borra el objeto y no tiene ningún valor de retorno:

```
e.delete()
```

También se puede borrar objetos en grupo. Todo objeto `QuerySet` tiene un método `delete()` que borra todos los miembros de ese `QuerySet`. Por ejemplo, esto borra todos los objetos `Entry` que tengan un año de `pub_date` igual a 2005:

```
Entry.objects.filter(pub_date__year=2005).delete()
```

Cuando Django borra un objeto, emula el comportamiento de la restricción de SQL `ON DELETE CASCADE` – en otras palabras, todos los objetos que tengan una clave foránea que apunte al objeto que está siendo borrado serán borrados también, por ejemplo:

```
b = Blog.objects.get(pk=1)
# Esto borra el Blog y todos sus objetos Entry.
b.delete()
```

Notar que `delete()` es el único método de `QuerySet` que no está expuesto en el `Manager` mismo. Esto es un mecanismo de seguridad para evitar que accidentalmente solicites `Entry.objects.delete()` y borres *todos* los Entry. Si *realmente* quieres borrar todos los objetos, hay que pedirlo explícitamente al conjunto completo de objetos:

```
Entry.objects.all().delete()
```

Métodos de Instancia Adicionales

Además de `save()` y `delete()`, un objeto modelo puede tener cualquiera o todos de los siguientes métodos.

S.1 get_FOO_display()

Por cada campo que indica la opción `choices`, el objeto tendrá un método `get_FOO_display()`, donde `FOO` es el nombre del campo. Este método devuelve el valor "humanamente legible" del campo. Por ejemplo, en el siguiente modelo:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)
class Person(models.Model):
    name = models.CharField(max_length=20)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

cada instancia de `Person` tendrá un método `get_gender_display`:

```
>>> p = Person(name='John', gender='M')
>>> p.save()
>>> p.gender
'M'
>>> p.get_gender_display()
'Male'
```

S.2 get_next_by_FOO(**kwargs) y get_previous_by_FOO(**kwargs)

Por cada campo `DateField` y `DateTimeField` que no tenga `null=True`, el objeto tendrá dos métodos `get_next_by_FOO()` y `get_previous_by_FOO()`, donde `FOO` es el nombre del campo. Estos métodos devuelven el objeto siguiente y anterior en orden cronológico respecto del campo en cuestión, respectivamente, levantando la excepción `DoesNotExist` cuando no exista tal objeto.

Ambos métodos aceptan argumentos de palabra clave opcionales, que deberían ser de la forma descripta en la sección "Patrones de búsqueda".

Notar que en el caso de valores de fecha idénticos, estos métodos utilizarán el ID como un chequeo secundario. Esto garantiza que no se saltearán registros ni aparecerán duplicados. Hay un ejemplo completo en los ejemplos de la API de búsqueda, en http://www.djangoproject.com/documentation/0.96/models/lookup/.

S.3 get_FOO_filename()

Todo campo `FileField` le dará al objeto un método `get_FOO_filename()`, donde `FOO` es el nombre del campo. Esto devuelve el nombre de archivo completo en el sistema de archivos, de acuerdo con la variable `MEDIA_ROOT`.

Notar que el campo `ImageField` es técnicamente una subclase de `FileField`, así que todo modelo que tenga un campo `ImageField` obtendrá también este método.

S.4 get_FOO_url()

Por todo campo `FileField` el objeto tendrá un método `get_FOO_url()`, donde `FOO` es el nombre del campo. Este método devuelve la URL al archivo, de acuerdo con tu variable `MEDIA_URL`. Si esta variable está vacía, el método devolverá una cadena vacía.

S.5 get_FOO_size()

Por cada campo `FileField` el objeto tendrá un método `get_FOO_size()`, donde `FOO` es el nombre del campo. Este método devuelve el tamaño del archivo, en bytes. (La implementación de este método utiliza `os.path.getsize`.)

S.6 save_FOO_file(filename, raw_contents)

Por cada campo `FileField`, el objeto tendrá un método `save_FOO_file()`, donde `FOO` es el nombre del campo. Este método guarda el archivo en el sistema de archivos, utilizando el nombre dado. Si un archivo con el nombre dado ya existe, Django le agrega guiones bajos al final del nombre de archivo (pero antes de la extensión) hasta que el nombre de archivos esté disponible.

S.7 get_FOO_height() and get_FOO_width()

Por cada campo `ImageField`, el objeto obtendrá dos métodos, `get_FOO_height()` y `get_FOO_width()`, donde `FOO` es el nombre del campo. Estos métodos devuelven el alto y el ancho (respectivamente) de la imagen, en pixeles, como un entero.

Atajos (Shortcuts)

A medida que desarrolles tus vistas, descubrirás una serie de modismos en la manera de utilizar la API de la base de datos. Django codifica algunos de estos modismos como atajos que pueden ser utilizados par simplificar el proceso de escribir vistas. Estas funciones se pueden hallar en el módulo `django.shortcuts`.

T.1 get_object_or_404()

Un modismo frecuente es llamar a `get()` y levantar un `Http404` si el objeto no existe. Este modismo es capturado en la función `get_object_or_404()`. Esta funcion toma un modelo Django como su primer argumento, y una cantidad arbitraria de argumentos de palabra clave, que le pasa al método `get()` del `Manager` por defecto del modelo. Luego levanta un `Http404` si el objeto no existe, por ejemplo:

```
# Get the Entry with a primary key of 3
e = get_object_or_404(Entry, pk=3)
```

Cuando se le pasa un modelo a esta función, se utiliza el `Manager` por defecto para ejecutar la consulta `get()` subyacente. Si no quieres que se utilice el manager por defecto, o si quiere buscar en una lista de objetos relacionados, se le puede pasar a `get_object_or_404()` un objeto `Manager` en vez:

```
# Get the author of blog instance e with a name of 'Fred'
a = get_object_or_404(e.authors, name='Fred')

# Use a custom manager 'recent_entries' in the search for an
# entry with a primary key of 3
e = get_object_or_404(Entry.recent_entries, pk=3)
```

T.2 get_list_or_404()

`get_list_or_404()` se comporta igual que `get_object_or_404()`, salvo porque llama a `filter()` en vez de a `get()`. Levanta un `Http404` si la lista resulta vacía.

Utilizando SQL Crudo

Si te encuentras necesitando escribir una consulta SQL que es demasiado compleja para manejarlo con el mapeador de base de datos de Django, todavía puede optar por escribir la sentencia directamente en SQL crudo.

La forma preferida para hacer esto es dándole a tu modelo métodos personalizados o métodos de `Manager` personalizados que realicen las consultas. Aunque no exista ningún requisito en Django que *exija* que las consultas a la base de datos vivan en la capa del modelo, esta implementación pone a toda tu lógica de acceso a los datos en un mismo lugar, lo cual es una idea astuta desde el punto de vista de organización del código. Por más instrucciones, véase el Apéndice B..

Finalmente, es importante notar que la capa de base de datos de Django es meramente una interfaz a tu base de datos. Puedes acceder a la base de datos utilizando otras herramientas, lenguajes de programación o frameworks de bases de datos – No hay nada específicamente de Django acerca de tu base de datos.

Parte X

Etiquetas de plantilla y filtros predefinidos

En este apéndice se listan la mayoria de las etiquetas y filtros utilizados en las plantillas o *templates*.

Etiquetas predefinidas

V.1 block

Define un bloque que puede ser sobreescrito por las plantillas derivadas. Véase la sección acerca de herencia de plantillas en el *doff-plantillas-herencia* para más información.

V.2 comment

Ignora todo lo que aparezca entre `{ % comment %}` y `{ % endcomment %}`.

V.3 cycle

Rota una cadena de texto entre diferentes valores, cada vez que aparece la etiqueta.

Dentro de un bucle, el valor rotan entre los distintos valores disponibles en cada iteración del bucle:

```
{ % for o in some_list %}
    <tr class="{ % cycle row1,row2 %}">
        ...
    </tr>
{ % endfor %}
```

Fuera de un bucle, hay que asignar un nombre único la primera vez que se usa la etiqueta, y luego hay que incluirlo ese nombre en las sucesivas llamadas:

```
<tr class="{ % cycle row1,row2,row3 as rowcolors %}">...</tr>
<tr class="{ % cycle rowcolors %}">...</tr>
<tr class="{ % cycle rowcolors %}">...</tr>
```

Se pueden usar cualquier número de valores, separándolos por comas. Asegúrese de no poner espacios entre los valores, sólo comas.

V.4 debug

Advertencia: No implementado

Muestra un montón de información para depuración de errores, incluyendo el contexto actual y los módulos importados.

V.5 extends

Sirve para indicar que esta plantilla extiende una plantilla padre.

Esta etiqueta se puede usar de dos maneras:

- `{% extends "base.html" %}` (Con las comillas) interpreta literalmente `"base.html"` como el nombre de la plantilla a extender.
- `{% extends variable %}` usa el valor de `variable`. Si la variable apunta a una cadena de texto, se usará dicha cadena como el nombre de la plantilla padre. Si la variable es un objeto de tipo `Template`, se usará ese mismo objeto como plantilla base.

En el *doff-plantillas* se pueden encontrar muchos ejemplo de uso de esta etiqueta.

V.6 filter

Filtra el contenido de una variable.

Los filtros pueden ser encadenados sucesivamente (La salida de uno es la entrada del siguiente), y pueden tener argumentos, como en la sintaxis para variables

He aquí un ejemplo:

```
{% filter escape|lower %}
    This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

V.7 firstof

Presenta como salida la primera de las variables que se le pasen que evalúe como no falsa. La salida será nula si todas las variables pasadas valen `False`.

He aquí un ejemplo:

```
{% firstof var1 var2 var3 %}
```

Equivale a:

```
{% if var1 %}
    {{ var1 }}
{% else %}{% if var2 %}
    {{ var2 }}
```

```
{% else %}{% if var3 %}
    {{ var3 }}
{% endif %}{% endif %}{% endif %}
```

V.8 for

Itera sobre cada uno de los elementos de un array o *lista*. Por ejemplo, para mostrar una lista de atletas, cuyos nombres estén en la lista `athlete_list`, podríamos hacer:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

También se puede iterar la lista en orden inverso usando `{% for obj in list reversed %}`.

Dentro de un bucle, la propia sentencia `for` crea una serie de variables. A estas variables se puede acceder únicamente dentro del bucle. Las distintas variables se explican en la *Tabla*.

Cuadro V.1: Variables accesibles dentro de bucles { % for %}

Variable	Descripción
`forloop.counter`	El número de vuelta o iteración actual (usando un índice basado en 1).
`forloop.counter0`	El número de vuelta o iteración actual (usando un índice basado en 0).
`forloop.revcounter`	El número de vuelta o iteración contando desde el fin del bucle (usando un índice basado en 1).
`forloop.revcounter0`	El número de vuelta o iteración contando desde el fin del bucle (usando un índice basado en 0).
`forloop.first`	`true` si es la primera iteración.
`forloop.last`	`true` si es la última iteración.
`forloop.parentloop`	Para bucles anidados, es una referencia al bucle externo.

V.9 if

La etiqueta `{% if %}` evalua una variable. Si dicha variable se evalua como una expresión "verdadera" (Es decir, que el valor exista, no esté vacia y no es el valor booleano `false`), se muestra el contenido del bloque:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}
```

Si la lista `athlete_list` no está vacía, podemos mostrar el número de atletas con la expresión `{{ athlete_list|length }}`

Además, como se puede ver en el ejemplo, la etiqueta `if` puede tener un bloque opcional `{% else %}` que se mostrará en el caso de que la evaluación de falso.

Las etiquetas `if` pueden usar operadores lógicos como `and`, `or` y `not` para evaluar expresiones más complejas:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches (OK, so
    writing English translations of Boolean logic sounds
    stupid; it's not our fault).
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

La etiqueta `if` no admite, sin embargo, mezclar los operadores `and` y `or` dentro de la misma comprobación, porque la orden de aplicación de los operadores lógicos sería ambigua. Por ejemplo, el siguiente código es inválido:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

Para combinar operadores `and` y `or`, puedes usar sentencias `if` anidadas, como en el siguiente ejemplo:

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

Es perfectamente posible usar varias veces un operador lógico, siempre que sea el mismo siempre. Por ejemplo, el siguiente código es válido:

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

V.10 ifchanged

Comprueba si un valor ha sido cambiado desde la última iteración de un bucle.

La etiqueta `ifchanged` solo tiene sentido dentro de un bucle. Tiene dos usos posibles:

1. Comprueba su propio contenido mostrado contra su estado anterior, y solo lo muestra si el contenido ha cambiado. El siguiente ejemplo muestra una lista de días, y solo aparecerá el nombre del mes si este cambia:

```
<h1>Archive for {{ year }}</h1>

{% for date in days %}
    {% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}
```

```
    <a href="{{ date|date:"M/d"|lower }}/">{{ date|date:"j" }}</a>
{% endfor %}
```

1. Se le pasa una o más variables, y se comprueba si esas variables han sido cambiadas:

   ```
   {% for date in days %}
       {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
       {% ifchanged date.hour date.date %}
           {{ date.hour }}
       {% endifchanged %}
   {% endfor %}
   ```

   El ejemplo anterior muestra la fecha cada vez que cambia, pero sólo muestra la hora si tanto la hora como el día han cambiado.

V.11 ifequal

Muestra el contenido del bloque si los dos argumentos suministrados son iguales.

He aquí un ejemplo:

```
{% ifequal user.id comment.user_id %}
    ...
{% endifequal %}
```

Al igual que con la etiqueta `{% if %}`, existe una cláusula `{% else %}` opcional.

Los argumentos pueden ser cadenas de texto, así que el siguiente código es válido:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

Sólo se puede comprobar la igualdad de variables o cadenas de texto. No se puede comparar con objetos `true` o `false`. Para ello, se debe utilizar la etiqueta `if` directamente.

V.12 ifnotequal

Es igual que `ifequal`, excepto que comprueba que los dos parámetros suministrados *no* sean iguales.

V.13 include

Carga una plantilla y la representa usando el contexto actual. Es una forma de "incluir" una plantilla dentro de otra.

El nombre de la plantilla puede o bien ser el valor de una variable o estar escrita en forma de cadena de texto, rodeada ya sea con comillas simples o comillas dobles, a gusto del lector.

El siguiente ejemplo incluye el contenido de la plantilla `"foo/bar.html"`:

```
{% include "foo/bar.html" %}
```

Este otro ejemplo incluye el contenido de la plantilla cuyo nombre sea el valor de la variable `template_name`:

```
{% include template_name %}
```

V.14 load

Carga una biblioteca de plantillas. En el **'Capítulo 10'_** puedes encontrar más información acerca de las bibliotecas de plantillas.

V.15 now

Muestra la fecha, escrita de acuerdo a un formato indicado.

Esta etiqueta fue inspirada por la función `date()` de PHP(), y utiliza el mismo formato que esta (http://php.net/date). Esta versión tiene, sin embargo, algunos extras.

La *Tabla* muestra las cadenas de formato que se pueden utilizar.

Cuadro V.2: Cadenas de formato para fechas y horas

Carác. formato	Descripción
a	`'a.m.'` o `'p.m.'`. (Obsérvese que la salida es ligeramente distinta de la de PHP, ya que aquí se incluyen punto
A	`'AM'` o `'PM'`.
b	El nombre del mes, en forma de abreviatura de tres letras minúsculas.
d	Día del mes, dos dígitos que incluyen rellenando con cero por la izquierda si fuera necesario.
D	Día de la semana, en forma de abreviatura de tres letras.
f	La hora, en formato de 12 horas y minutos, omitiendo los minutos si estos son cero.
F	El mes, en forma de texto
g	La hora, en formato de 12 horas, sin rellenar por la izquierda con ceros.
G	La hora, en formato de 24 horas, sin rellenar por la izquierda con ceros.
h	La hora, en formato de 12 horas.
H	La hora, en formato de 24 horas.
i	Minutos.
j	El día del mes, sin rellenar por la izquierda con ceros.
l	El nombre del día de la semana.
L	Booleano que indica si el año es bisiesto.
m	El día del mes, rellenando por la izquierda con ceros si fuera necesario.
M	Nombre del mes, abreviado en forma de abreviatura de tres letras.
n	El mes, sin rellenar con ceros
N	La abreviatura del mes siguiendo el estilo de la Associated Press.
O	Diferencia con respecto al tiempo medio de Grennwich (*Greenwich Mean Time* - GMT)
P	La hora, en formato de 12 horas, más los minutos, recto si estos son cero y con la indicación a.m./p.m. Además, s
r	La fecha en formato RFC 822.
s	Los segundos, rellenos con ceros por la izquierda de ser necesario.
S	El sufijo inglés para el día del mes (dos caracteres).
t	Número de días del mes.
T	Zona horaria
w	Día de la semana, en forma de dígito.
W	Semana del año, siguiente la norma ISO-8601, con la semana empezando el lunes.
y	Año, con dos dígitos.
Y	Año, con cuatro dígitos.

Continued on next page

Cuadro V.2 – continued from previous page

z	Día del año
Z	Desfase de la zona horaria, en segundos. El desplazamiento siempre es negativo para las zonas al oeste del meridi

He aquí un ejemplo:

```
It is {% now "jS F Y H:i" %}
```

Se pueden escapar los caracteres de formato con una barra invertida, si se quieren incluir de forma literal. En el siguiente ejemplo, se escapa el significado de la letra "f" con la barra invertida, ya que de otra manera se interpretaría como una indicación de incluir la hora. La "o", por otro lado, no necesita ser escapada, ya que no es un carácter de formato:

```
It is the {% now "jS o\f F" %}
```

El ejemplo mostraría: "It is the 4th of September".

V.16 regroup

Reagrupa una lista de objetos similares usando un atributo común.

Para comprender esta etiqueta, es mejor recurrir a un ejemplo. Digamos que `people` es una lista de objetos de tipo `Person`, y que dichos objetos tienen los atributos `first_name`, `last_name` y `gender`. Queremos mostrar un listado como el siguiente:

```
* Male:
    * George Bush
    * Bill Clinton
* Female:
    * Margaret Thatcher
    * Condoleezza Rice
* Unknown:
    * Pat Smith
```

El siguiente fragmento de plantilla mostraría como realizar esta tarea:

```
{% regroup people by gender as grouped %}
<ul>
{% for group in grouped %}
    <li>{{ group.grouper }}
    <ul>
        {% for item in group.list %}
        <li>{{ item }}</li>
        {% endfor %}
    </ul>
    </li>
{% endfor %}
</ul>
```

Como se puede ver, `{% regroup %}` crea una nueva variable, que es una lista de objetos que tienen dos tributos, `grouper` y `list`. En `grouper` se almacena el valor de agrupación, `list` contiene una lista de los objetos que tenían en común al valor de agrupación. En este caso, `grouper` podría valer `Male`, `Female` y `Unknown`, y `list` sería una lista con las personas correspondientes a cada uno de estos sexos.

Hay que destacar que `{% regroup %}` **no** funciona correctamente cuando la lista no está ordenada por el mismo atributo que se quiere agrupar. Esto significa que si la lista del ejemplo no está ordenada por el sexo, se debe ordenar antes correctamente, por ejemplo con el siguiente código:

```
{% regroup people|dictsort:"gender" by gender as grouped %}
```

V.17 spaceless

Elimina los espacios en blanco entre etiquetas Html. Esto incluye tabuladores y saltos de línea.

El siguiente ejemplo:

```
{% spaceless %}
    <p>
        <a href="foo/">Foo</a>
    </p>
{% endspaceless %}
```

Retornaría el siguiente código HTML:

```
<p><a href="foo/">Foo</a></p>
```

Sólo se eliminan los espacios *entre* las etiquetas, no los espacios entre la etiqueta y el texto. En el siguiente ejemplo, no se quitan los espacios que rodean la palabra `Hello`:

```
{% spaceless %}
    <strong>
        Hello
    </strong>
{% endspaceless %}
```

V.18 templatetag

Permite representar los caracteres que están definidos como parte del sistema de plantillas.

Como el sistema de plantillas no tiene el concepto de "escapar" el significado de las combinaciones de símbolos que usa internamente, tenemos que recurrir a la etiqueta `{% templatetag %}` si nos vemos obligados a representarlos.

Se le pasa un argumento que indica que combinación de símbolos debe producir. Los valores posibles del argumento se muestran en la *Tabla*.

Cuadro V.3: Argumentos válidos de templatetag

Argumento	Salida
openblock	{%
closeblock	%}
openvariable	{{
closevariable	}}
openbrace	{
closebrace	}
opencomment	{#
closecomment	#}

V.19 widthratio

Esta etiqueta es útil para presentar gráficos de barras y similares. Calcula la proporción entre un valor dado y un máximo predefinido, y luego multiplica ese cociente por una constante.

Veamos un ejemplo:

```
<img src="bar.gif" height="10" width="{% widthratio this_value max_value 100 %}" />
```

Si `this_value` vale 175 y `max_value` es 200, la imagen resultante tendrá un ancho de 88 pixels (porque 175/200 = 0.875 y 0.875 * 100 = 87.5, que se redondea a 88).

Filtros predefinidos

W.1 add

Ejemplo:

```
{{ value|add:"5" }}
```

Suma el argumento indicado.

W.2 addslashes

Ejemplo:

```
{{ string|addslashes }}
```

Añade barras invertidas antes de las comillas, ya sean simples o dobles. Es útil para pasar cadenas de texto como javascript, por ejemplo:

W.3 capfirst

Ejemplo:

```
{{ string|capfirst }}
```

Pasa a mayúsculas la primera letra de la primera palabra.

W.4 center

Ejemplo:

```
{{ string|center:"50" }}
```

Centra el texto en un campo de la anchura indicada.

W.5 cut

Ejemplo:

```
{{ string|cut:"spam" }}
```

Elimina todas las apariciones del valor indicado.

W.6 date

Ejemplo:

```
{{ value|date:"F j, Y" }}
```

Formatea una fecha de acuerdo al formato indicado en la cadena de texto (Se usa el mismo formato que con la etiqueta `now`).

W.7 default

Ejemplo:

```
{{ value|default:"(N/A)" }}
```

Si `value` no está definido, se usa el valor del argumento en su lugar.

W.8 default_if_none

Ejemplo:

```
{{ value|default_if_none:"(N/A)" }}
```

Si `value` es nulo, se usa el valor del argumento en su lugar.

W.9 dictsort

Ejemplo:

```
{{ list|dictsort:"foo" }}
```

Acepta una lista de diccionarios y devuelve una lista ordenada según la propiedad indicada en el argumento.

W.10 dictsortreversed

Ejemplo:

```
{{ list|dictsortreversed:"foo" }}
```

Acepta una lista de diccionarios y devuelve una lista ordenada de forma descendente según la propiedad indicada en el argumento.

W.11 divisibleby

Ejemplo:

```
{% if value|divisibleby:"2" %}
    Even!
{% else %}
    Odd!
{% else %}
```

Devuelve `True` si es valor pasado es divisible por el argumento.

W.12 escape

Ejemplo:

```
{{ string|escape }}
```

Transforma un texto que esté en HTML de forma que se pueda representar en una página web. Concretamente, realiza los siguientes cambios:

- `"&"` a `"&"`
- < a `"<"`
- > a `">"`
- `'"'` (comilla doble) a `'&quot;'`
- `"'"` (comillas simple) a `'&#39;'`

W.13 filesizeformat

Ejemplo:

```
{{ value|filesizeformat }}
```

Representa un valor, interpretándolo como si fuera el tamaño de un fichero y "humanizando" el resultado, de forma que sea fácil de leer. Por ejemplo, las salidas podrían ser `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, etc.

W.14 first

Ejemplo:

```
{{ list|first }}
```

Devuelve el primer elemento de una lista.

W.15 fix_ampersands

Ejemplo:

```
{{ string|fix_ampersands }}
```

Reemplaza los símbolos *ampersand* con la entidad `&`.

W.16 floatformat

Ejemplos:

```
{{ value|floatformat }}
{{ value|floatformat:"2" }}
```

Si se usa sin argumento, redondea un número en coma flotante a un único dígito decimal (pero sólo si hay una parte decimal que mostrar), por ejemplo:

- `36.123` se representaría como `36.1`.
- `36.15` se representaría como `36.2`.
- `36` se representaría como `36`.

Si te utiliza un argumento numérico, `floatformat` redondea a ese número de lugares decimales:

- `36.1234` con floatformat:3 se representaría como `36.123`.
- `36` con floatformat:4 se representaría como `36.0000`.

Si el argumento pasado a `floatformat` es negativo, redondeará a ese número de decimales, pero sólo si el número tiene parte decimal.

- `36.1234` con floatformat:-3 gets converted to `36.123`.
- `36` con floatformat:-4 gets converted to `36`.

Usar `floatformat` sin argumentos es equivalente a usarlo con un argumento de -1.

W.17 get_digit

Ejemplo:

```
{{ value|get_digit:"1" }}
```

Dado un número, devuelve el dígito que esté en la posición indicada, siendo 1 el dígito más a la derecha. En caso de que la entrada sea inválida, devolverá el valor original (Si la entrada o el argumento no fueran enteros, o si el argumento fuera inferior a 1). Si la entrada es correcta, la salida siempre será un entero.

W.18 join

Ejemplo:

```
{{ list|join:", " }}
```

Concatena todos los elementos de una lista para formar una cadena de texto, usando como separador el texto que se le pasa como argumento.

W.19 length

Ejemplo:

```
{{ list|length }}
```

Devuelve la longitud del valor.

W.20 length_is

Ejemplo:

```
{% if list|length_is:"3" %}
    ...
{% endif %}
```

Devuelve un valor booleano que será verdadero si la longitud de la entrada coincide con el argumento suministrado.

W.21 linebreaks

Ejemplo:

```
{{ string|linebreaks }}
```

Convierte los saltos de línea en etiquetas `<p>` y `<br />`.

W.22 linebreaksbr

Ejemplo:

```
{{ string|linebreaksbr }}
```

Convierte los saltos de línea en etiquetas `<br />`.

W.23 linenumbers

Ejemplo:

```
{{ string|linenumbers }}
```

Muestra el texto de la entrada con números de línea.

W.24 ljust

Ejemplo:

```
{{ string|ljust:"50" }}
```

Justifica el texto de la entrada a la izquierda utilizando la anchura indicada.

W.25 lower

Ejemplo:

```
{{ string|lower }}
```

Convierte el texto de la entrada a letras minúsculas.

W.26 make_list

Ejemplo:

```
{% for i in number|make_list %}
    ...
{% endfor %}
```

Devuelve la entrada en forma de lista. Si la entrada es un número entero, se devuelve una lista de dígitos. Si es una cadena de texto, se devuelve una lista de caracteres.

W.27 pluralize

Ejemplo:

```
The list has {{ list|length }} item{{ list|pluralize }}.
```

Retorno el sufijo para formar el plural si el valor es mayor que uno. Por defecto el sufijo es `'s'`.

Ejemplo:

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

Para aquellas palabras que requieran otro sufijo para formar el plural, podemos usar una sintaxis alternativa en la que indicamos el sufijo que queramos con un argumento.

Ejemplo:

```
You have {{ num_walruses }} walrus{{ num_walrus|pluralize:"es" }}.
```

Para aquellas palabras que forman el plural de forma más compleja que con un simple sufijo, hay otra tercera sintaxis que permite indicar las formas en singular y en plural a partir de una raíz común.

Ejemplo:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

W.28 random

Ejemplo:

```
{{ list|random }}
```

Devuelve un elemento elegido al azar de la lista.

W.29 removetags

Ejemplo:

```
{{ string|removetags:"br p div" }}
```

Elimina de la entrada una o varias clases de etiquetas [X]HTML. Las etiquetas se indican en forma de texto, separando cada etiqueta a eliminar por un espacio.

W.30 rjust

Ejemplo:

```
{{ string|rjust:"50" }}
```

Justifica el texto de la entrada a la derecha utilizando la anchura indicada..

W.31 slice

Ejemplo:

```
{{ some_list|slice:":2" }}
```

Devuelve una sección de la lista.

W.32 slugify

Ejemplo:

```
{{ string|slugify }}
```

Convierte el texto a minúsculas, elimina los caracteres que no formen palabras (caracteres alfanuméricos y carácter subrayado), y convierte los espacios en guiones. También elimina los espacios que hubiera al principio y al final del texto.

W.33 stringformat

Ejemplo:

```
{{ number|stringformat:"02i" }}
```

Formatea el valor de entrada de acuerdo a lo especificado en el formato que se le pasa como parámetro.

W.34 striptags

Ejemplo:

```
{{ string|striptags }}
```

Elimina todas las etiquetas [X]HTML.

W.35 time

Ejemplo:

```
{{ value|time:"P" }}
```

Formatea la salida asumiendo que es una fecha/hora, con el formato indicado como argumento (Lo mismo que la etiqueta `now`).

W.36 timesince

Ejemplos:

```
{{ datetime|timesince }}
{{ datetime|timesince:"other_datetime" }}
```

Representa una fecha como un intervalo de tiempo (por ejemplo, “4 days, 6 hours”).

Acepta un argumento opcional, que es una variable con la fecha a usar como punto de referencia para calcular el intervalo (Si no se especifica, la referencia es el momento actual). Por ejemplo, si `blog_date` es una fecha con valor igual a la medianoche del 1 de junio de 2006, y `comment_date` es una fecha con valor las 08:00 horas del día 1 de junio de 2006, entonces `{{ comment_date|timesince:blog_date }}` devolvería “8 hours”.

W.37 timeuntil

Ejemplos:

```
{{ datetime|timeuntil }}
{{ datetime|timeuntil:"other_datetime" }}
```

Es similar a `timesince`, excepto en que mide el tiempo desde la fecha de referencia hasta la fecha dada. Por ejemplo, si hoy es 1 de junio de 2006 y `conference_date` es una fecha cuyo valor es igual al 29 de junio de 2006, entonces `{{ conference_date|timeuntil }}` devolvería “28 days”.

Acepta un argumento opcional, que es una variable con la fecha a usar como punto de referencia para calcular el intervalo, si se quiere usar otra distinta del momento actual. Si `from_date` apunta al 22 de junio de 2006, entonces `{{ conference_date|timeuntil:from_date }}` devolvería “7 days”.

W.38 title

Ejemplo:

```
{{ string|titlecase }}
```

Representa una cadena de texto en forma de título, siguiendo las convenciones del idioma inglés (todas las palabras con la inicial en mayúscula).

W.39 truncatewords

Ejemplo:

```
{{ string|truncatewords:"15" }}
```

Recorta la salida de forma que tenga como máximo el número de palabras que se indican en el argumento.

W.40 truncatewords_html

Ejemplo:

```
{{ string|truncatewords_html:"15" }}
```

Es similar a `truncatewords`, excepto que es capaz de reconocer las etiquetas HTML y, por tanto, no deja etiquetas "huérfanas". Cualquier etiqueta que se hubiera abierto antes del punto de recorte es cerrada por el propio filtro.

Es menos eficiente que `truncatewords`, así que debe ser usada solamente si sabemos que en la entrada va texto HTML.

W.41 unordered_list

Ejemplo:

```
<ul>
    {{ list|unordered_list }}
</ul>
```

Acepta una lista, e incluso varias listas anidadas, y recorre recursivamente las mismas representándolas en forma de listas HTML no ordenadas, *sin incluir* las etiquetas de inicio y fin de lista (`<ul>` y `</ul>` respectivamente).

Se asume que las listas está en el formato correcto. Por ejemplo, si `var` contiene `['States', [['Kansas', [['Lawrence', []], ['Topeka', []]]], ['Illinois', []]]]`, entonces `{{ var|unordered_list }}` retornaría lo siguiente:

```
<li>States
<ul>
        <li>Kansas
        <ul>
                <li>Lawrence</li>
                <li>Topeka</li>
        </ul>
        </li>
        <li>Illinois</li>
</ul>
</li>
```

W.42 upper

Ejemplo:

```
{{ string|upper }}
```

Convierte una string a mayúsculas.

W.43 urlencode

Ejemplo:

```
<a href="{{ link|urlencode }}">linkage</a>
```

Escapa la entrada de forma que pueda ser utilizado dentro de una URL.

W.44 urlize

Ejemplo:

```
{{ string|urlize }}
```

Transforma un texto de entrada, de forma que si contiene direcciones URL en texto plano, las convierte en enlaces HTML.

W.45 urlizetrunc

Ejemplo:

```
{{ string|urlizetrunc:"30" }}
```

Convierte las direcciones URL de un texto en enlaces, recortando la representación de la URL para que el número de caracteres sea como máximo el del argumento suministrado.

W.46 wordcount

Ejemplo:

```
{{ string|wordcount }}
```

Devuelve el número de palabras en la entrada.

W.47 wordwrap

Ejemplo:

```
{{ string|wordwrap:"75" }}
```

Ajusta la longitud del texto para las líneas se adecúen a la longitud especificada como argumento.

W.48 yesno

Ejemplo:

```
{{ boolean|yesno:"Yes,No,Perhaps" }}
```

Dada una serie de textos que se asocian a los valores de `True`, `False` y (opcionalmente) `None`, devuelve uno de esos textos según el valor de la entrada. Véase la *Tabla*.

Cuadro W.1: Ejemplos del filtro yesno

Valor	Argumento	Salida
`True`	`"yeah,no,maybe"`	`yeah`
`False`	`"yeah,no,maybe"`	`no`
`None`	`"yeah,no,maybe"`	`maybe`
`None`	`"yeah,no"`	`"no"` (considera `None` como `False` si no se asigna ningún texto a `None`.

Índice