

0.1

0.10.5em

0.1.1

0.1.10.5em

0.1.1.00.5em

0em



FIG. 2.
PONY "MAGIC"

Sistemas Web Desconectados

Release 1

van Haaster, Diego Marcos; Defossé, Nahuel

August 21, 2009

Índice general

0.1.	I
subsection0.11spanish I Tecnologías del servidor	3
1. CGI	7
2. WSGI	9
3. Lenguajes interpretados	11
4. Frameworks	13
4.1. Framework Web	14
5. Model View Controller	15
6. Mapeador Objeto-Relacional	17
7. Django	19
7.1. Estructuración de un proyecto en Django	19
7.2. Módulo settings	20
7.3. Módulo manage	20
7.4. Módulo urls	21
7.5. Módulo models	21
7.6. Módulo views	21
7.7. El ciclo de una petición	22
7.8. Interactuar con una base de datos	23
II Glosario	27
III Referencia sobre Django	31
8. Instalación de Django	33
9. Comandos del módulo manage	35
9.1. El comando syncdb	35

9.2.	El comando runserver	35
9.3.	El comando validate	35
10.	Comandos de usuario	37
IV	Indices, glosario y tablas	39
V	Referencia de la definición de modelos	43
A.	Campos	47
A.1.	AutoField	48
A.2.	BooleanField	48
A.3.	CharField	48
A.4.	DateField	48
A.5.	DateTimeField	48
A.6.	EmailField	48
A.7.	FileField	48
A.8.	FilePathField	49
A.9.	FloatField	50
A.10.	ImageField	50
A.11.	IntegerField	51
A.12.	IPAddressField	51
A.13.	NullBooleanField	51
A.14.	PositiveIntegerField	51
A.15.	PositiveSmallIntegerField	51
A.16.	SlugField	51
A.17.	SmallIntegerField	51
A.18.	TextField	52
A.19.	TimeField	52
A.20.	URLField	52
B.	Opciones para todos los campos	53
B.1.	null	53
B.2.	blank	53
B.3.	choices	53
B.4.	db_column	54
B.5.	db_index	54
B.6.	default	54
B.7.	editable	55
B.8.	help_text	55
B.9.	primary_key	55
B.10.	radio_admin	55
B.11.	unique	55
B.12.	unique_for_date	55
B.13.	unique_for_month	56
B.14.	unique_for_year	56
B.15.	verbose_name	56
C.	Relaciones	57
C.1.	Relaciones Muchos-a-Uno	57
C.2.	Relaciones Muchos-a-Muchos	60
D.	Opciones de los Metadatos del Modelo	63

D.1.	db_table	63
D.2.	get_latest_by	64
D.3.	order_with_respect_to	64
D.4.	ordering	64
D.5.	unique_together	65
D.6.	verbose_name	65
D.7.	verbose_name_plural	65
E.	Managers	67
E.1.	Nombres de Manager	67
E.2.	Managers Personalizados	67
F.	Métodos de Modelo	71
F.1.	__str__	72
F.2.	Ejecutando SQL personalizado	72
F.3.	Sobreescribiendo los Métodos por omisión del Modelo	72
	Bibliografía	75

Índice:

Parte I

Tecnologías del servidor

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 mediante 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 transferidos 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 estándar.

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

Dentro de las variables 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, siendo muy popular Perl y más recientemente el lenguaje PHP.

Lenguajes interpretados

2.1. PHP

2.2. Ruby

2.3. Python

Python es un lenguaje interpretado. .. Escribir sobre el concepto de módulos y algo de programación modular, ya que da soporte a prototipo http://es.wikipedia.org/wiki/Programaci%C3%B3n_modular

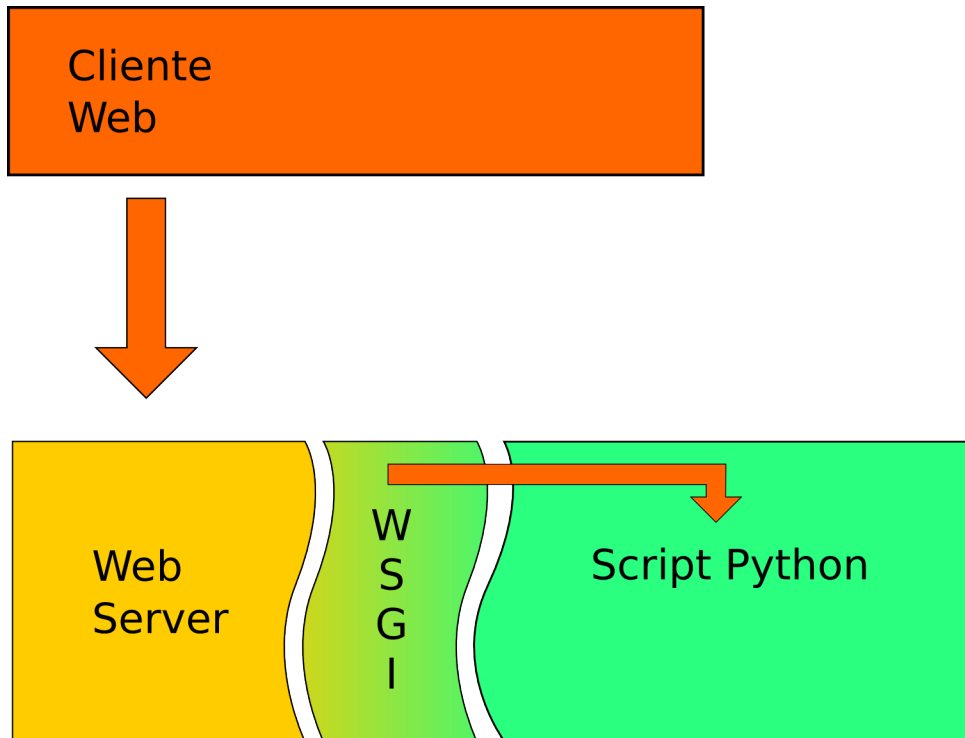
2.3.1. WSGI

WSGI o Web Server Gateway Interface es una especificación 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 estructuras 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 *objeto llamable*) al cual se invoca para iniciar la respuesta.

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



En el siguiente ejemplo, la función `app` devuelve *Hello World* informándole al navegador web, que el contenido se trata de texto plano.

```
def app(environ, start_response):  
    start_response('200 OK', [('Content-Type', 'text/plain')])  
    return ['Hello World\n']
```

Mapeador Objeto-Relacional

En las aplicaciones modernas, la lógica arbitraria a menudo implica interactuar con una base de datos. Detrás de escena, un *programa impulsado por una base de datos* se conecta a un servidor de base de datos, recupera algunos datos de esta, y los presenta al usuario con un formato agradable para su interpretación. Una aplicación web no escapa a esta aseveración, solo que presenta los datos representados en HTML, así mismo un sitio puede proporcionar funcionalidad que permita a los visitantes del sitio poblar la base de datos por su propia cuenta.

Amazon.com, por ejemplo, es un buen 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 se envían datos al servidor, como opiniones de cliente, estos son insertados en la base de datos de opiniones.

La forma simple de interactuar con una base de datos, es mediante el uso de bibliotecas provistas por los lenguajes para ejecutar consultas SQL y una vez obtenidos los datos, procesarlos.

En este ejemplo se usa la biblioteca MySQLdb para conectar con una base de datos MySQL, recuperar algunos registros:

```
import MySQLdb

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()
```

Este enfoque funciona, pero presenta algunos problemas:

- **Los parámetros de la conexión a la base de datos están codificandos** *en duro (hard-coding)*.
- **Se debe escribir una cantidad de código estereotípico: crear una** conexión, un cursor, ejecutar una sentencia, y cerrar la conexión.
- **Ata a las aplicaciones a MySQL. Si, en el camino, se quiere cambiar MySQL** por PostgreSQL por ejemplo, se deben alterar todas las líneas que hagan falta para la nueva biblioteca o conector, parámetros de conexión, posiblemente reescribir el SQL, etc.

Por otro lado y quizá más importante a la hora de desarrollar un programador que trabajó con programación orientada a objetos y bases de datos relacionales, debe realizar un cambio de contexto cada vez que requiera interactuar con la base de datos, escribiendo consultas en SQL y luego lidiar con los resultados obtenidos de las consultas entre los objetos. Este *cambio de contexto* es debido a una diferencia que existe entre los dos paradigmas involucrados. Mientras que

el modelo relacional trata con relaciones, conjuntos y la lógica matemática correspondiente, el paradigma orientado a objetos trata con objetos, atributos

y asociaciones de unos con otros. Tan pronto como se quieran persistir los objetos utilizando una base de datos relacional esta desaveniencia resulta evidente.

Las primeras aproximaciones al mapeo relacional de objetos, surgen de convertir los valores de los objetos en grupos de valores simples para almacenarlos en la base de datos (y volverlos a convertir luego de recuperarlos de la base de datos). Sin embargo, esta traducción simple dista mucho del concepto de *objetos persistentes*, la idea de estos es la traducción automática de objetos en formas almacenables en la base de datos y su posterior recuperación conservando las propiedades y las relaciones entre los mismos.

Con la finalidad de lograr *objetos persistentes* un buen número de sistemas de mapeo objeto-relacional se han desarrollado a lo largo de los años y aunque su efectividad es muy discutida la realidad es que estos permiten agilizar el proceso de desarrollo, paleando mucho de los problemas presentados con anterioridad.

Desde el punto de vista de un programador, un ORM debe lucir como un almacén de objetos persistentes. Uno puede crear objetos y trabajar normalmente con ellos, los cambios que sufran terminarán siendo reflejados en la base de datos.

Model View Controller

En aplicaciones complejas que impliquen sofisticadas interfaces, como las aplicaciones web, la lógica de la interfaz de usuario cambia con más frecuencia que los almacenes de datos y la lógica de negocio. Si se realiza un diseño mezclando los componentes de interfaz y de negocio, entonces las consecuencias serán que, cuando se necesite cambiar la interfaz, se tendrá que modificar trabajosamente los componentes de negocio, teniendo de esta forma mayor trabajo y mayor riesgo de error.

El patrón arquitectural MVC, *Modelo Vista Controlador* trata de realizar un diseño que desacople la interfaz o vista del modelo, con la finalidad de mejorar la reusabilidad. De esta forma las modificaciones en las vistas impactan en menor medida en la lógica de negocio o de datos.

Este patrón fue descrito por primera vez en 1979 por Trygve Reenskaug [[Tryg1979](#)], entonces trabajando en Smalltalk en laboratorios de investigación de Xerox. La implementación original está descrita a fondo en Programación de Aplicaciones en Smalltalk-80(TM): Como utilizar Modelo Vista Controlador [[SmallMVC](#)].

Descripción del patrón:

- **Modelo** Esta es la capa de datos, una representación de la información con la cual el sistema opera. La lógica de datos asegura la integridad y permite derivar nuevos datos.
- **Vista** Esta es la capa de presentación del modelo, seleccionando qué mostrar y cómo mostrarlo, usualmente la interfaz de usuario.
- **Controlador** Esta capa responde a eventos, usualmente acciones del usuario, e invoca cambios en el modelo y probablemente en la vista.

El patrón MVC se ve frecuentemente en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página. El modelo es el sistema de gestión de base de datos y la lógica de negocio, y el controlador es el responsable de recibir los eventos de entrada desde la vista.

Frameworks

Nota: Poner CLI

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*.

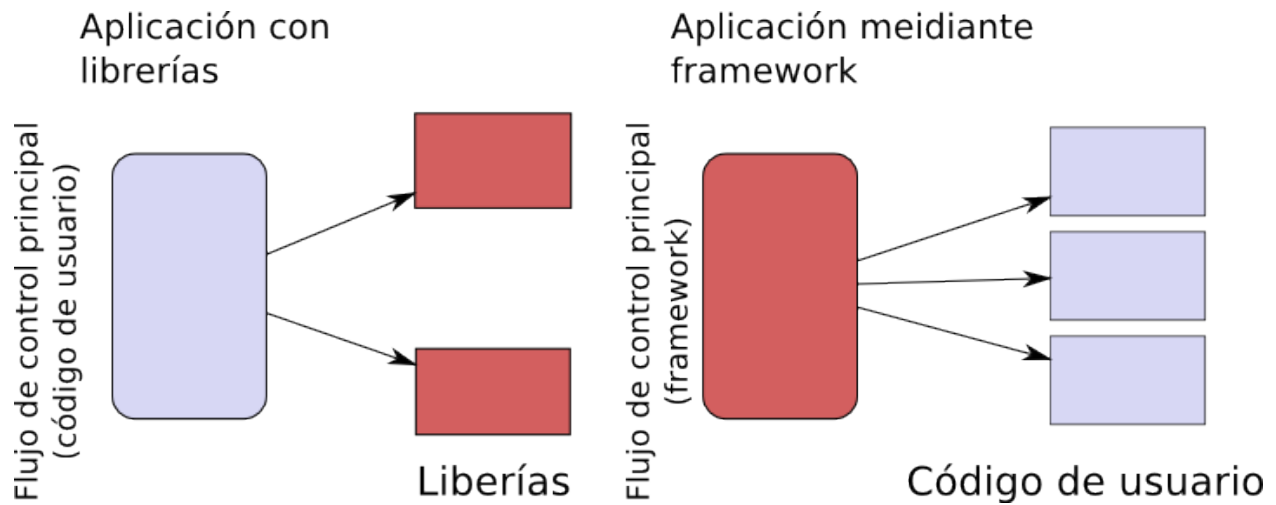
Además agrega que los frameworks son similares a las bibliotecas de software (a veces llamadas librerías) dado que proveen abstracciones reusables de código a las cuales se accede mediante una API bien definida. Sin embargo, existen ciertas características que diferencian al framework de una librería 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 componente del framework, existe un comportamiento genérico con alguna utilidad, que puede ser redefinido con funcionalidad del usuario.
- **Extensibilidad** 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 requerimientos 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 extracciones de dinero, en vez de la mecánica para preservar 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 genera 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 equipo de desarrollo. Tras completar el desarrollo en un framework, el equipo de desarrollo no debe volver a invertir tiempo en aprendizaje en sucesivos desarrollos.



5.1. 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 Wikipeda [\[WIKI002\]](#)

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

Rails

Symfony

Django

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

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

En junio del 2008 fue anunciada la creación de la Django Software Foundation, 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 syndicar 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.

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,

¹ Del ingles “The Web framework for perfectionists with deadlines”

- *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.

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:

- **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 medios 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 lenguaje python, la configuración 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 cliente. Las URLs se escriben mediante expresiones regulares del lenguaje Python. Este sistema de URLs aprovecha muy bien el modulo de expresiones regulares del lenguaje permitiendo por ejemplo recuperar grupos 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 regular 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 otorgen 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 siguiente
(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 ciclo de una petición

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

² Mediante el comando `syncdb` del módulo `manage` del proyecto

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

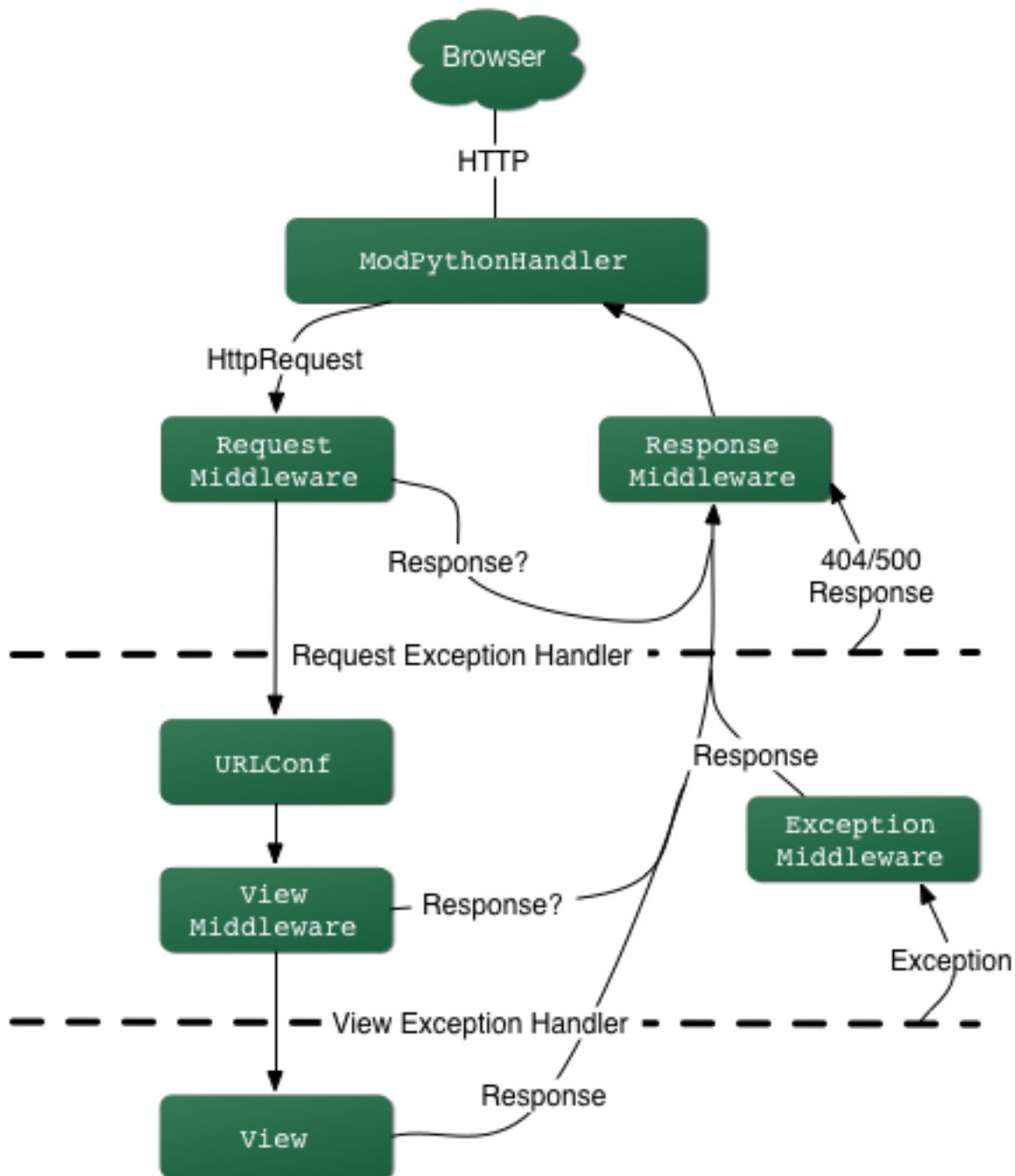
Luego de atravesar los middlewares de request, mediante las definiciones de URLs, se selecciona la vista a ser ejecutada.

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 ha 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.



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 vez 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 código 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 básicos,** 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 modelo de datos de la aplicación. Con esta información Django genera automáticamente 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 acceder los datos:

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

En estas líneas se ven algunos detalles de la interacción 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 comunicación 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 operaciones 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 clase *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 consulta, 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 mediante encadenamiento.

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

³ En el lenguaje SQL, las consultas se realizan mediante la sentencia `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 )
```

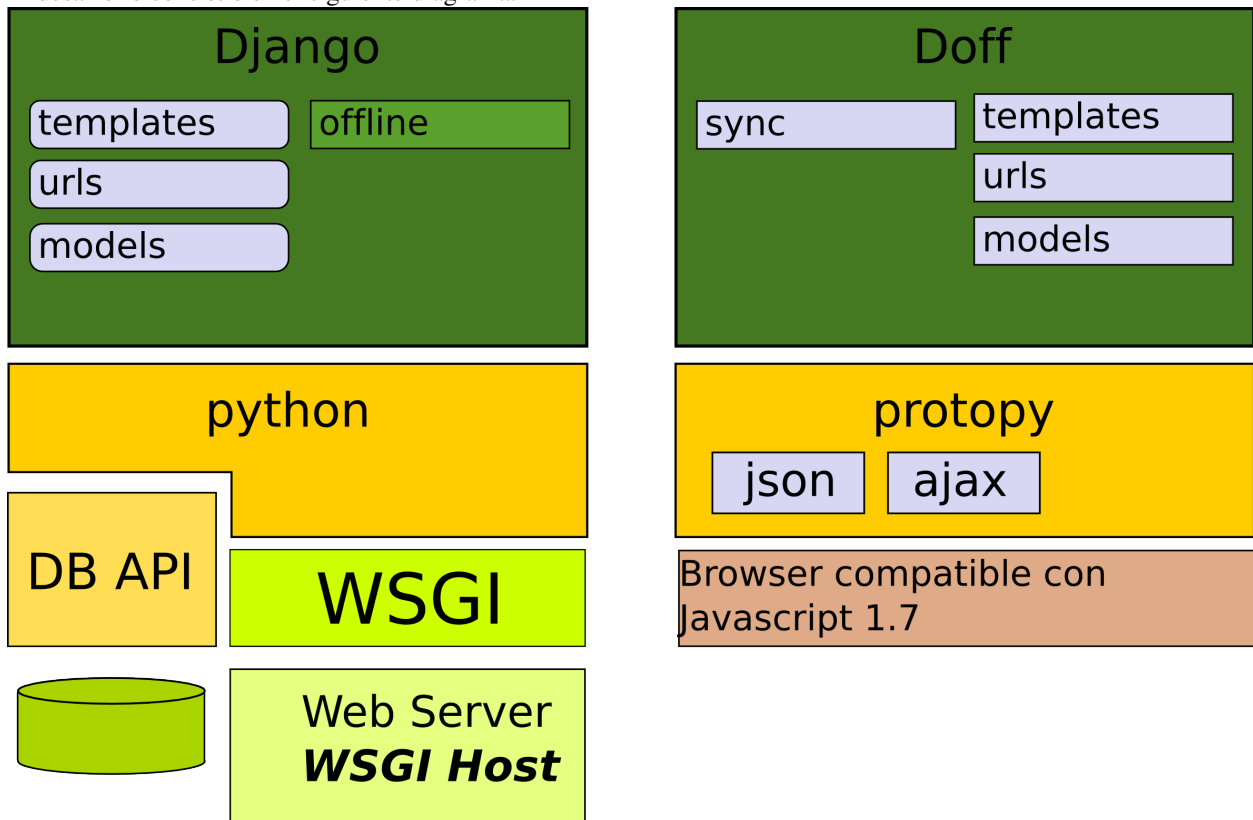
el ORM agrega a cada modelo la propiedad *objects*:

```
>>> 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 ambiente de un navegador web, como es Firefox, deja entrever muchos aspectos que no resultan para nada triviales al momento de codificar.

- Se requieren varias líneas de código para implementar un framework.
- Como llega el código al navegador y se inicia su ejecución.
- La cara visible o vista debe ser fácilmente manipulable por la aplicación de usuario.
- Como los datos generados en el cliente son informados al servidor.
- El framework debe brindar soporte a la aplicación de usuario de una forma natural y transparente.
- Se debe promover al reuso y la extensión de funcionalidad del framework.
- Como se ponen en marcha los mecanismos o acciones que la aplicación de usuario define.

En este capítulo se introducen las ideas principales que motivaron la creación de una librería en JavaScript, que brinde el soporte necesario al framework y a buena parte de los ítems expuestos.

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

proto‘type + ‘py‘thon = ‘protopy

‘‘La creación nace del caos’’, la librería ‘‘Protopy’’ no escapa a esta afirmación e inicialmente nace de la integración de Prototype con las primeras funciones para lograr la modularización; con el correr de las líneas de código footnote{Forma en que los informáticos miden el paso del tiempo} el desarrollo del framework torna el enfoque inicial poco sustentable, requiriendo este de funciones más Python-compatibles se desecha la librería base y se continúa con un enfoque más ‘‘pythonico’’, persiguiendo de esta forma acercar la semántica de JavaScript 1.7 a la del lenguaje de programación Python.

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

Protopy

Protopy es una librería JavaScript para el desarrollo de aplicaciones web dinámicas. 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 remítase al apéndice *Protopy*

Organizando el código

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 código que será 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 (validación, accesibilidad, etc) y los fragmentos de código que se pasan al cliente son bien conocidos por el desarrollador; en proyectos que implican mayor cantidad de funcionalidad JavaScript con grandes cantidades de código este enfoque resulta complejo de mantener y evolucionar en el tiempo.

Es por lo expuesto hasta aquí que se busca una forma de organizar y obtener el código que resulte sustentable y escalable; similar al concepto de “módulo” en Python *servidor-lenguajes-python*, se enfocó el desarrollo de Protopy en pequeñas unidades de código manejables, que puedan ser fácilmente obtenidas e interpretadas por el cliente y sumadas entre sí cumplan una determinada tarea. Esta técnica no es nueva en programación y básicamente implica llevar el concepto de “divide y vencerás” ó “análisis descendente (Top-Down)” al ámbito de JavaScript.

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

En su forma más 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 módulos es usando nombres de **paquetes** asociados a urls, de forma similar a la anterior 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 relación 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 módulos 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 estándar. Ver el apéndice *Módulos estándar* 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:

Esto importa todos los nombres excepto aquellos que empiecen con un underscore (`_`).

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

Uno de los principales inconvenientes a los que Protopy da solución es a la inclusión dinámica de funcionalidad bajo demanda, esto se logra con los “módulos”.

Las funciones principales para trabajar con los módulos son “require” para cargar un módulo en el ámbito de nombres local y “publish” para que los módulos 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 constructor 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 provee el metodo `verbl__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”, ambos 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.

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.

Manejando los eventos

`%event.connect %event.`

Envolviendo a gears

%Almacenamiento en la base de datos local %LocalServer para guardar código

Auditando el código

%Logger

Interactuando con el servidor

%HttpRequest

Soporte para json

%JSON REF <http://www.json.org/> %Esto es algo sobre json, quizá 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 programación, 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 JSON<http://www.json.org/>

%Ahora vemos que hace protopy, la necesidad Mientras que un cliente se encuentre sin conexión con el servidor web, es capaz de generar y almacenar datos usando su base de datos local. Al reestablecer la conexión con el servidor web, estos datos deben ser transmitedos a la base de datos central para su actualización y posterior sincronización del resto de los clientes. 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 conexión; este formato debe ser “comprendido” tanto por el cliente como por el servidor. Desde un primer momento se penso en JSON como 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 razón 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 implementación de un parser JSON contra la implementación de uno en XML. Para el lector interesado agregar el soporte para XML en Protopy consta de escribir un modulo que realice esa tarea y agregarlo al paquete base.

%El como El soporte para JSON se 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 lenguaje JavaScript estan soportados y tienen su representación correspondiente, object, array, number, string, etc. pero este modulo interpreta además de una forma particular a aquellos objetos que implementen el metodo `verbl__json__`, dejando de este modo en manos del desarrollador la representación en JSON de determinados objetos. La inclusión del metodo

verbl__json__l resulta de especial impotancia a la hora de pasar a JSON los objetos creados en base a tipos definidos por el desarrollador mediante el constructor “type”.

Con el soporte de datos ya establecidos en la libreria, el framework solo debe limitarse 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 código 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 rendering 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 sentencia **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 grupos 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 grupo 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
- [Tryg1979] Trygve Reenskaug, <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>
- [SmallMVC] Steve Burbeck, Ph.D. <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>
- [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)
```

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 conecta 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 verificación de sintaxis

Comandos de usuario

Django permite

Parte VIII

Etiquetas de plantilla y filtros predefinidos

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

Etiquetas predefinidas

E.1. block

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

E.2. comment

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

E.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.

E.4. debug

Advertencia: No implementado

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

E.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.

E.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%}
```

E.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%}

```

E.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 E.1: Variables accesibles dentro de bucles `{% for %}`

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

E.9. if

La etiqueta `{% if %}` evalúa una variable. Si dicha variable se evalúa como una expresión “verdadera” (Es decir, que el valor exista, no esté vacía 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%}
```

E.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.

E.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.

E.12. ifnotequal

Es igual que `ifequal`, excepto que comprueba que los dos parámetros suministrados *no* sean iguales.

E.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 %}
```

E.14. load

Carga una biblioteca de plantillas. En el ‘**Capítulo 10**’ puedes encontrar más información acerca de las bibliotecas de plantillas.

E.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 E.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 puntos)
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 Greenwich (<i>Greenwich Mean Time</i> - 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 E.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 meridiano

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”.

E.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 atributos, `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%}
```

E.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%}
```

E.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 E.3: Argumentos válidos de templatetag

Argumento	Salida
openblock	{ %
closeblock	% }
openvariable	{ {
closevariable	} }
openbrace	{
closebrace	}
opencomment	{ #
closecomment	# }

E.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:

```

```

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

F.1. add

Ejemplo:

```
{{ value|add:"5" }}
```

Suma el argumento indicado.

F.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:

F.3. capfirst

Ejemplo:

```
{{ string|capfirst }}
```

Pasa a mayúsculas la primera letra de la primera palabra.

F.4. center

Ejemplo:


```
{{ string|center:"50" }}
```

Centra el texto en un campo de la anchura indicada.

F.5. cut

Ejemplo:

```
{{ string|cut:"spam" }}
```

Elimina todas las apariciones del valor indicado.

F.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`).

F.7. default

Ejemplo:

```
{{ value|default:"(N/A) " }}
```

Si `value` no está definido, se usa el valor del argumento en su lugar.

F.8. default_if_none

Ejemplo:

```
{{ value|default_if_none:"(N/A) " }}
```

Si `value` es nulo, se usa el valor del argumento en su lugar.

F.9. dictsort

Ejemplo:

```
{{ list|dictsort:"foo" }}
```

Acepta una lista de diccionarios y devuelve una lista ordenada según la propiedad indicada en el argumento.

F.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.

F.11. divisibleby

Ejemplo:

```
{% if value|divisibleby:"2" %}  
    Even!  
{% else %}  
    Odd!  
{% else %}
```

Devuelve True si es valor pasado es divisible por el argumento.

F.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 '" '
- '"' (comillas simple) a '' '

F.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.

F.14. first

Ejemplo:

```
{{ list|first }}
```

Devuelve el primer elemento de una lista.

F.15. fix_ampersands

Ejemplo:

```
{{ string|fix_ampersands }}
```

Reemplaza los símbolos *ampersand* con la entidad `&`.

F.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 se 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.

F.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.

F.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.

F.19. length

Ejemplo:

```
{{ list|length }}
```

Devuelve la longitud del valor.

F.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.

F.21. linebreaks

Ejemplo:

```
{{ string|linebreaks }}
```

Convierte los saltos de línea en etiquetas `<p>` y `
`.

F.22. linebreaksbr

Ejemplo:

```
{{ string|linebreaksbr }}
```

Convierte los saltos de línea en etiquetas `
`.

F.23. `linenumbers`

Ejemplo:

```
{{ string|linenumbers }}
```

Muestra el texto de la entrada con números de línea.

F.24. `ljust`

Ejemplo:

```
{{ string|ljust:"50" }}
```

Justifica el texto de la entrada a la izquierda utilizando la anchura indicada.

F.25. `lower`

Ejemplo:

```
{{ string|lower }}
```

Convierte el texto de la entrada a letras minúsculas.

F.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.

F.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" }}.
```

F.28. random

Ejemplo:

```
{{ list|random }}
```

Devuelve un elemento elegido al azar de la lista.

F.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.

F.30. rjust

Ejemplo:

```
{{ string|rjust:"50" }}
```

Justifica el texto de la entrada a la derecha utilizando la anchura indicada..

F.31. slice

Ejemplo:

```
{{ some_list|slice:":2" }}
```

Devuelve una sección de la lista.

F.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.

F.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.

F.34. striptags

Ejemplo:

```
{{ string|striptags }}
```

Elimina todas las etiquetas [X]HTML.

F.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`).

F.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”.

F.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”.

F.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).

F.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.

F.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.

F.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 (y respectivamente).

Se asume que las listas están 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>
```

F.42. upper

Ejemplo:

```
{{ string|upper }}
```

Convierte una string a mayúsculas.

F.43. urlencode

Ejemplo:

```
<a href="{{ link|urlencode }}">linkage</a>
```

Escapa la entrada de forma que pueda ser utilizado dentro de una URL.

F.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.

F.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.

F.46. wordcount

Ejemplo:

```
{{ string|wordcount }}
```

Devuelve el número de palabras en la entrada.

F.47. wordwrap

Ejemplo:

```
{{ string|wordwrap:"75" }}
```

Ajusta la longitud del texto para las líneas se adecúen a la longitud especificada como argumento.

F.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 F.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.