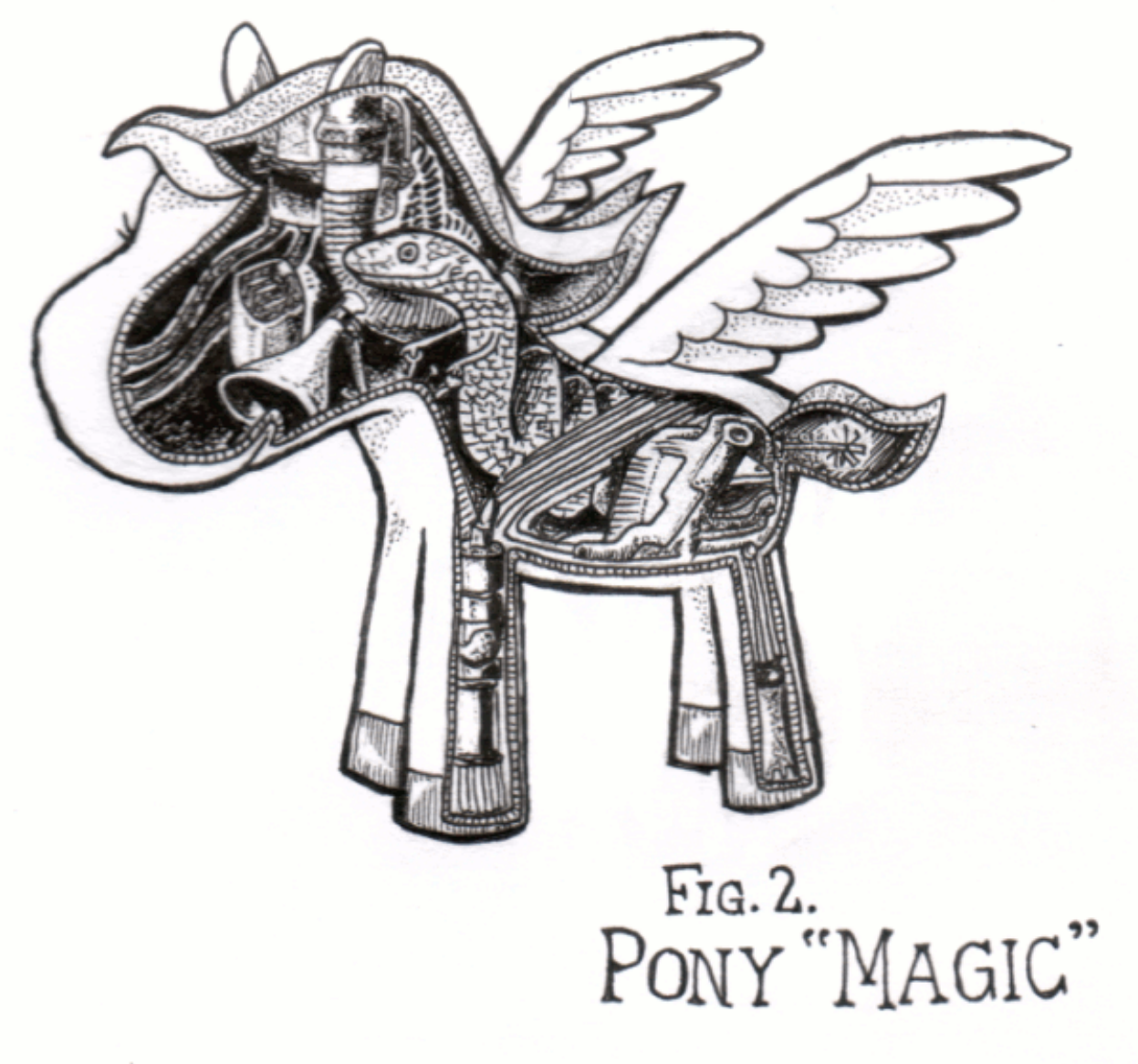


Sistemas Web Desconectados

Release 1

van Haaster, Diego Marcos; Defossé, Nahuel

October 03, 2009

Índice general

CAPÍTULO 1

Introducción

"Yo sólo puedo mostrarte la puerta. Tú eres quien debe atravesarla."

—Morfeo

1.1 Motivación

Hoy en día Internet supone más que un medio de intercambio de información, su constante expansión la ha convertido en un terreno muy atractivo para la implementación de sistemas de información y ha posibilitado las tareas de mantenimiento y actualización de aplicaciones sin necesidad de distribuir software adicional a miles de usuarios potenciales.

En vez de crear versiones específicas para los múltiples sistemas operativos existentes en el mercado, la aplicación web se escribe una vez y se ejecuta de la misma manera en todas las plataformas [SOverAdvWebFwk2009].

A partir de la necesidad de acelerar y estandarizar la forma en la que se desarrollan las aplicaciones web han aparecido plataformas de desarrollo, o frameworks, para la mayoría de los lenguajes de programación [WikiListFramework2009]. A lo largo de los últimos 10 años se ha producido una importante evolución de los frameworks [ApacheSlingEv2009], permitiendo que muchas tareas comunes se resuelven de una manera predefinida y modificable, ayudando al programador a focalizarse en la solución del problema particular de su desarrollo.

Las aplicaciones web tienen ciertas limitaciones en cuanto a la funcionalidad que ofrecen al usuario. Acciones comunes en las aplicaciones de escritorio, como dibujar en la pantalla o arrastrar y soltar, aún no están soportadas de manera directa. Sin embargo, existe una corriente encargada de mejorar la experiencia de usuario, que ha acuñando el término RIA [1] o Aplicación Rica Basada en Internet [CanalARRIA2005]. Un factor clave en la evolución hacia las RIAs es la incorporación

[1] Rich Internet Application

de la tecnología denominada AJAX [CanalARRIA2005], que permite, por ejemplo actualizaciones parciales del contenido de una página [PerezAJAX2009].

La web, en el ámbito del software, constituye un medio singular por su ubicuidad y sus estándares abiertos. El conjunto de normas que rigen la forma en que se generan y transmiten los documentos a través de la web son regulados por la W3C (Consorcio World Wide Web).

La mayor parte de la web está soportada sobre sistemas operativos y software de servidor [NetCraft2009] que se rigen bajo licencias OpenSource [OSI2009] (Apache, BIND, Linux, OpenBSD, FreeBSD). Los lenguajes con los que se desarrollan las aplicaciones web son generalmente OpenSource, como PHP, Python, Ruby, Perl y Java. Los frameworks web escritos sobre estos lenguajes utilizan alguna licencia OpenSource para su distribución; incluso frameworks basados en lenguajes propietarios son liberados bajo licencias OpenSource.

1.2 Objetivos

La principal limitación de las aplicaciones web, en comparación con las aplicaciones tradicionales, es la necesidad de contar con conexión constante para funcionar. Dotarlas de la capacidad de trabajo sin conexión ataca su principal limitación.

Si bien los elementos necesarios para llevar a cabo esta tarea están disponibles actualmente, aún no están contemplados en los diseños de los frameworks web.

Hoy en día, dotar a una determinada aplicación web de la capacidad de funcionar sin conexión requiere de un proceso de desarrollo que difiere del empleado en la aplicación en linea.

El objetivo principal del presente trabajo de tesis es aportar una extensión a un framework web que permita migrar las aplicaciones de manera simple, convergiendo en un único proceso de desarrollo.

Nota: Recuperar objetivos secundarios de la nota

1.3 Alcance

Nota: Debe estar en pasado. Hay que de todos modos re-readactar según marta.

Tras el estudio de las características se determinará el framework a utilizar. Se tendrán en cuenta características tales como como la calidad del mapeador de objetos, la simplicidad del controlador, extensibilidad del sistema de escritura de plantillas y flexibilidad general.

Se realizará un estudio sobre los componentes que intervienen en el desarrollo de aplicaciones web: el navegador y el servidor web. Se abordará un lenguaje de programación y un framework web del lenguaje seleccionado, se desarrollará para este un framework básico. Se formulará un esquema de sincronización para pequeñas aplicaciones.

CAPÍTULO 2

Tecnologías del servidor

Este capítulo tiene como finalidad introducir los conceptos básicos concernientes a la *generación dinámica de contenido* en el servidor web (2.1).

A continuación se realiza una revisión general de los componentes de un servidor web y, luego, se analizan los lenguajes dinámicos y sus frameworks.

A continuación se analizan las características que hacen relevante el estudio de los lenguajes dinámicos como plataforma de desarrollo de aplicaciones web dinámicas.

2.1 Generación dinámica de páginas Web

En el enfoque dinámico, cuando un usuario realiza una solicitud, el mensaje enviado tiene como objetivo la ejecución de un programa o secuencia de comandos en el servidor. Por lo general, el procesamiento involucra el uso de la información proporcionada por el usuario para buscar registros en una base de datos y generar una página HTML personalizada para el cliente.

Tradicionalmente, la tecnología utilizada se conoce como CGI, estándar que consiste en delegar la generación de contenido a un programa. CGI se limita a definir la entrada y salida de éste.

Un enfoque más moderno para la generación de contenido dinámico es la incrustación de secuencias de comandos dentro de las páginas HTML. Estos comandos son leídos y ejecutados en el servidor al momento de responder a la solicitud del cliente, como es el caso de los lenguajes PHP o ASP.

Los servidores web primigenios y monolíticos evolucionaron a una arquitectura modular [ApacheMod2009] [MicrosoftIIS2009], en la cual un módulo brinda soporte para una tarea específica. Los módulos más comunes son los de autenticación, bitácora, balance de carga, así como también como los de generación de contenido.

Las nuevas formas de interacción entre un servidor web y un programa se desarrolladas con el objeto de satisfacer necesidades más complejas, que no fueron tenidas en cuenta en la genericidad

de CGI.

En la plataforma Java, se especificaron los Servlets [SunServlet2009], con implementaciones como Tomcat [ApacheTomcat2009], y en 1996 se publicó la especificación J2EE, que formula una arquitectura de aplicación web dividida en capas que ejecuta un servidor de aplicaciones.

Basados en la especificación J2EE, se crearon numerosos frameworks cuyos lineamientos determinaron la manera de concebir las aplicaciones web durante casi una década.

En junio de 2004 se publica el proyecto Ruby On Rails, un framework de aplicaciones web, desarrollado en el lenguaje de programación Ruby, que revolucionó la forma de concebir los frameworks y la web.

James Duncan Davidson, el creador de Tomcat y Ant, llegó a decir que Ruby On Rails es el framework web mejor pensado que él ha usado. Davison pasó 10 años desarrollando aplicaciones web, frameworks y la especificación de los Servlets para el lenguaje Java [RailsQuotes2009] [1].

Nota: Esto se copió en la intro

A continuación se realiza una revisión general de los componentes de un servidor web y, luego, se analizan los lenguajes dinámicos y sus frameworks.

2.1.1 Servidor Web

Un servidor web, o *web server* es un software encargado de recibir solicitudes de recursos de un cliente, típicamente un *navegador web*, a través del protocolo HTTP y de generar una respuesta. Mediante la especificación MIME, que se incluye en el encabezado de la respuesta, se puede identificar qué tipo de archivo es devuelto, siendo el tipo más común el HTML.

El contenido que se envía al cliente puede ser de origen *estático* o *dinámico*. El contenido estático es aquel que proviene desde un archivo en el sistema de archivos sin ninguna modificación. El contenido dinámico, en contraposición al contenido estático, proviene de la salida de algún programa, un script o algún tipo de API invocada por el servidor web (como SSI, *CGI*, SCGI, FastCGI, JSP, ColdFusion, NSAPI o ISAPI).

La única forma de acceder a los recursos del servidor web es a través de una URL, independientemente de su naturaleza.

2.1.2 CGI

Common Gateway Interface (CGI) [2] surge alrededor del año 1998, como el primer estándar de comunicación o *API* entre un servidor web y un programa de generación de contenidos.

[1] *"Rails is the most well thought-out web development framework I've ever used. And that's in a decade of doing web applications for a living. I've built my own frameworks, helped develop the Servlet API, and have created more than a few web servers from scratch. Nobody has done it like this before."*

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

La salida del programa invocado por el servidor web puede ser un documento HTML entendible para el navegador, o cualquier otro tipo de archivo, como imágenes, sonidos, contenido interactivo, etc.

Las variables de entorno y la entrada estándar constituyen los mecanismos de entrada del programa, mientras que el contenido devuelto al servidor web proviene de la salida estándar.

Dentro de la información provista por el servidor web se tienen los parámetros HTTP (como la URL, el método, el nombre del host, el puerto, etc.) y la información sobre el servidor. Estos datos se transfieren mediante variables de entorno.

Si existiese un cuerpo en la petición HTTP, como por ejemplo el contenido de un formulario, la aplicación CGI accede a esta información como entrada estándar.

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

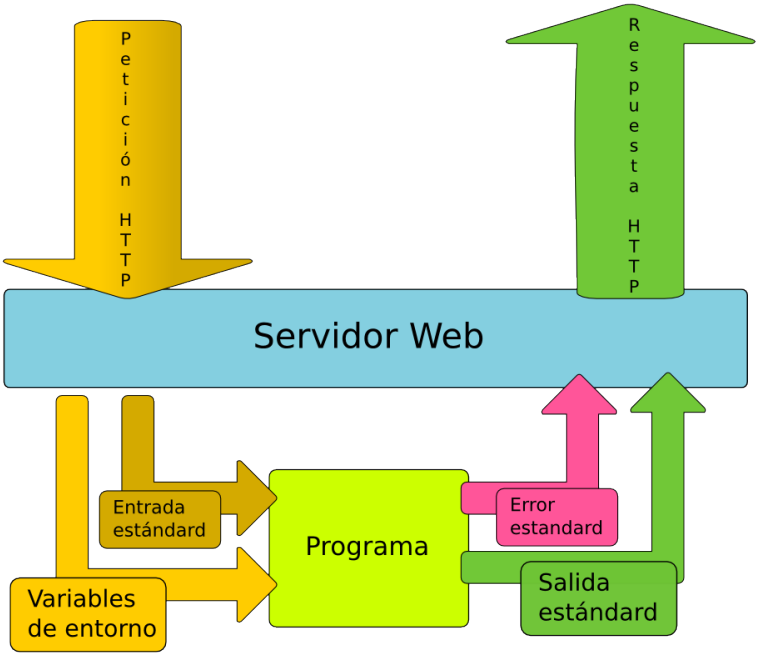


Figura 2.1: Proceso de una solicitud con CGI.

Dentro de las variables de entorno, la Wikipedia [WikiCGI2009] menciona:

- QUERY_STRING

  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 la 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).

Las variables de entorno que se intercambian del servidor al CGI son:

- 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.

CGI posee dos limitaciones importantes. Una es la sobrecarga producido por la ejecución de un programa y la segunda es que no fue diseñado para mantener información sobre la sesión. Cada petición se trata de manera independiente [DwightErwin1996].

2.1.3 Lenguajes Dinámicos

Nota: Se copio a la intro

A continuación se analizan las características que hacen relevante el estudio de los lenguajes dinámicos como plataforma de desarrollo de aplicaciones web dinámicas.

Una de las clasificaciones más generales que se realizan de los lenguajes de programación es según la identificación de su objetivo. Los lenguajes de programación de *propósito general* están orientados a resolver cualquier tipo de problema, mientras que los lenguajes de *propósito específico*, o *DSL*[]_*, están enfocados en resolver un tipo de problema de manera más eficaz. Un ejemplo muy popular de DSL es el lenguaje de macros embebido en la planilla de cálculo Microsoft Excel [DavidPollak2006].

Otra clasificación es la de lenguajes interpretados y lenguajes compilados. Un lenguaje de programación interpretado es aquel en el cual los programas son ejecutados por un intérprete, en lugar de realizarse una traducción a lenguaje máquina (proceso de compilación). En teoría cualquier lenguaje de programación podría ser compilado o interpretado.

En un lenguaje interpretado el código fuente es el ejecutable. En cambio, para llegar a ejecutar un programa escrito en un lenguaje compilado se deben atravesar dos etapas. La primera consiste en la traducción de las sentencias a código máquina y la segunda, en el enlace en la cual se ensambla el código objeto resultado de la compilación. En esta última etapa también se resuelven los enlaces entre los diferentes módulos compilados.

Existe un mecanismo intermedio de ejecución, conocido como máquina virtual. En éste existe un proceso de compilación del código fuente a un lenguaje intermedio, comúnmente denominado *bytecode*. Este bytecode es luego ejecutado sobre un intérprete, al cual se denomina máquina virtual. La traducción del código fuente a *bytecode* puede ser explícita, como en el lenguaje Java, o implícita como en Python, donde se mezcla en el intérprete la funcionalidad de compilación a bytecode e interpretación.

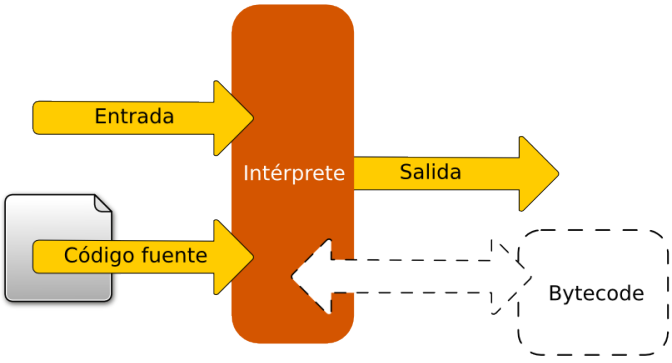


Figura 2.2: Lenguaje interpretado con máquina virtual.

Los lenguajes de programación interpretados suelen ser de alto nivel y de *tipado dinámico*, es decir que la mayoría de las comprobaciones realizadas en tiempo de compilación son evaluadas durante la ejecución [3] .

[3] Domain Specific Language

Nota: Hasta acá llegó la revisión del jueves 24 de Septiembre

La mayoría de los lenguajes interpretados permiten ser ejecutado en varias plataformas [4] (multiplataforma), ya que solo es necesario disponer de un intérprete compilado, a diferencia de los lenguajes compilados en los cuales se requiere disponer de un compilador para la plataforma y compilar el código.

La mayoría de los lenguajes interpretados son lenguajes dinámicos, esto permite el agregado de código, extensión o redefinición de objetos y hasta inclusive modificar tipos de datos, en tiempo de ejecución. Cabe destacar que si bien estas características son factibles de implementar sobre lenguajes estáticos, no resulta sencillo.

En base a la clasificación antes mencionada, se analizan los lenguajes de programación más populares para el desarrollo de aplicaciones web.

Perl es un lenguaje de programación de propósito general diseñado por Larry Wall en 1987. Perl toma características del lenguaje C, del lenguaje interpretado shell (sh), AWK, sed, Lisp y, en menor medida, de muchos otros lenguajes de programación. Su uso principal es el procesamiento de texto, siendo muy popular en programación de sistemas. Muchos sistemas basados en CGI están escritos en Perl (sistemas de administración de servidores, correo, etc.). Perl esta disponible para muchas plataformas, incluyendo todas las variantes de UNIX.

Estructuralmente, Perl está basado en un estilo de bloques como los del C o AWK, y fue ampliamente adoptado por su destreza en el procesamiento de texto.

La principal crítica al lenguaje es la ambigüedad y complejidad de su sintaxis, una tarea puede ser realizada de varias maneras diferentes, dado lugar a confusión en grupos de trabajo.

Java es un lenguaje de programación orientado a objetos, multiplataforma y propósito general, basado en máquina virtual, de compilación explícita. Java ha definido algunos elementos importantes en cuanto a la web dinámica, como los applets [5] y la especificación J2EE. Dos desventajas que presenta este lenguaje, al igual que su predecesor C++, del cual heredó una sintaxis verborragica [SeanKellyRecFromAdict2009] además del tipado estático. Como contrapartida resulta una alternativa veloz para ciertas tareas _poner_citas_, lo que motivó el desarrollo de lenguajes dinámicos [TolksdorfJVM2009] que hacen uso de su máquina virtual [6]. Se destaca, Scala, que hoy cuenta con frameworks con una concepción post RubyOnRails.

http://www.archive.org/details/SeanKellyRecoveryfromAddiction .. [SeanKellyRecFromAdict2009] *Recuperándose de la adicción*, Sean Kelly, Vídeo hospedado en *The Internet Archive*, último acceso Septiembre de 2009, http://www.archive.org/details/SeanKellyRecoveryfromAddiction

PHP es un lenguaje interpretado, originalmente diseñado para ser embebido dentro del código HTML y procesado en el servidor, lo cual lo convierte en un DSL. Con los años evolucionó hacia

[4] Estas comprobaciones comprenden el chequeo de tipos de datos, la resolución de métodos, etc.

[5] *Una plataforma es una combinación de hardware y software usada para ejecutar aplicaciones; en su forma más simple consiste únicamente de un sistema operativo, una arquitectura, o una combinación de ambos. La plataforma más conocida es probablemente Microsoft Windows en una arquitectura x86* [WikiPlataforma2009].

[6] Pequeños programas que se ejecutan en el navegador web

un lenguaje de propósito general. Toma elementos de Perl y shellscript, C, y recientemente Java.

Tradicionalmente el ciclo de ejecución consiste en:

- A partir de la URL de la solicitud del cliente se determina el archivo PHP que se encargará de generar la respuesta
- El servidor, activa el módulo encargado de la interpretación de PHP con el archivo y la solicitud como entrada
- La salida es devuelta al cliente.

Presenta importantes ventajas sobre CGI, ya que no es necesario confeccionar un programa de usuario y la resolución de URLs está dada por la estructura del sistema de archivos. Si bien es muy popular [PHPNetPopularity2009] y está disponible en la gran mayoría de los servidores UNIX, simplificando el *deployment* PHP es criticado por no poseer ámbito de nombres para los módulos, promover el código desordenado y tener serios problemas a la hora de la optimización [BlogHardz2008] . Los autores de Django y Ruby On Rails provenir del lenguaje PHP en [SnakesAndRubies2005] .

Ruby es un lenguaje orientado fuertemente objetos, multiplataforma, creado en 1995 por Yukihiro "Matz" Matsumoto, en Japón. A menudo comparado con *Smalltak*, se suele decir que Ruby es un lenguaje de objetos puro, ya que *todo* es un objeto. Posee muchas características avanzadas como metaclases, clausuras, iteradores, integración natural de expresiones regulares en la sintaxis, etc.

Su sintaxis es compacta, gracias a la utilización de simbología, parte de la cual fue tomada de Perl. Existen varios intérpretes de Ruby, siendo la oficial escrita en C, se conocen YARV [YARV2009] , JRuby [JRuby2009] , Rubinius [Rubinius2009] , IronRuby [IronRuby2009] , y MacRuby [MacRuby2009] .

Una de las *Killer App* según el autor es el framework para contenido web denominado "Ruby on Rails" . Su versión estable oficial fue liberada en el año 2005 y proponía un cambio radical al enfoque complejo de *J2EE* [RailsQuotes2009] .

Ruby aún posee baja aceptación debido, quizás, a que la documentación oficial solía estar en idioma Japonés (aunque la situación se ha venido revirtiendo últimamente). Otra desventaja importante es que la velocidad del intérprete oficial es bastante baja (cuando se la compara con otros lenguajes dinámicos similares como Python) y variables entre plataformas. **Python** es un lenguaje de programación interpretado multiparadigma, de propósito general. Fue creado por por Guido van Rossum en el año 1991. Se trata de un lenguaje dinámico que toma elementos de varios lenguajes, como C, Java, Scheme, entre otros.

Python puede ser extendido mediante módulos escritos en C o C++, y también se puede embeber el intérprete en otros lenguajes. Permite cargar bibliotecas de enlace dinámico.

Python es considerado por parte de la comunidad UNIX, como una evolución de Perl, de sintaxis limpia y potente. Eric Raymond, en el artículo "Why Python?", explica su conversión de Perl a Python [EricRaymon2000].

Muchos de las sistemas webs basados en CGI, están escritos en Perl, por lo cual no es sorpresa

encontrar una buena cantidad de proyectos del lenguaje Python orientados a la Web [PythonPyPi2009].

Los motivos por los cuales se seleccionó [7] el lenguaje Python para el desarrollo de la aplicación, son los que a continuación se enumeran:

- Popularidad [PythonPyPi2009]
- **Performance adecuada en función de las lineas de código escritas [DhananjayNene2009]**
    - Si bien Perl es más veloz para tareas de tratamiento de texto, su sintaxis es compleja
- **Filosofía de simplicidad**
    - Sintaxis clara
    - Zen de Python [PythonOrgZen2009]

En el *apéndice* se encuentra una referecia detallada del lenguaje.

Módulo de interconexión de servidor web con Python

Existen varios módulos para interconexión de intérprete de Python con un servidor web. Phillip J. Eby formuló un método que sigue la filosofía del lenguaje, en la [PEP333], que denominó WSGI (Web Server Gateway Interface). Informalmente se puede decir que WSGI es una traducción de CGI al lenguaje Python. Su objetivo principal fue estandarizar sobre el lenguaje el mecanismo de comunicación entre el servidor y una aplicación.

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

Para satisfacer una solicitud bajo WSGI, se invoca la función de entrada. Esta recibe 2 argumentos:

1. Un diccionario con las variables de entorno, al igual que en CGI
2. Una función (u *objeto llamable*) al cual se invoca para iniciar la respuesta.

En el siguiente ejemplo constituye una aplicación mínima en WSGI. La función app es el punto de entrada y devuelve *Hello World*. Utiliza la función que recibe como segundo argumento, 'start_response', para que el cliente determine como tratar la respuesta. En este caso, texto plano.

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

[7] Estos lenguajes interpretados utilizan compilación JIT a bytecode de la JVM.

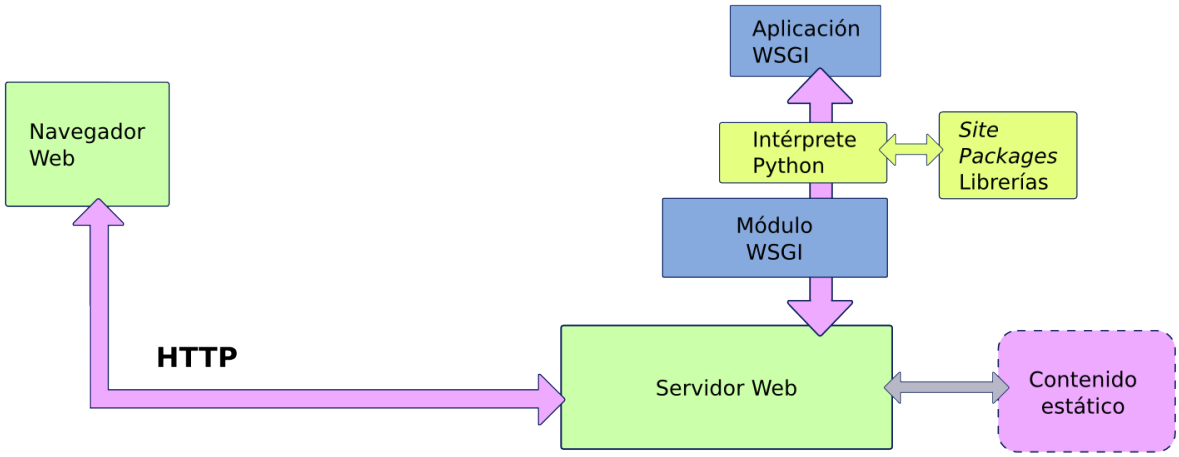


Figura 2.3: Esquema WSGI.

2.2 Frameworks Web

2.2.1 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:

- Inversión de control

  Tradicionalmente se escriben las aplicaciones haciendo llamadas a las bibliotecas de manera explicita. El flujo de control es definido por el programador. En el caso de una aplicación escrita sobre un framework, el flujo es definido por éste.

- Comportamiento por defecto definido

  En cada elemento del framework, existe un comportamiento genérico con alguna utilidad.

- Extensibilidad

  El comportamiento predefinido de cada componente es generalmente modificado por el programador con algún fin específico. Los métodos utilizados en los frameworks realizados sobre lenguajes orientados a objetos suelen ser redefinición o

especialización.

- No modificabilidad del código propio del framework

  Las extensiones y definiciones propias de una aplicación se realizan sobre el código proyecto y no sobre el código del framework.

Se utiliza un framework con el objeto de simplificar el desarrollo de un proyecto, permitiendo a los programadores desligarse de resolver detalles comunes de bajo nivel.

A modo de ejemplo, 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 una serie de argumentos comunes existen contra la utilización de frameworks:

- La complejidad de sus APIs
- Incertidumbre ante la existencia de múltiples alternativas para un mismo tipo de aplicación.
- El aprendizaje de un framework suele requerir tiempo extra, que debe ser tenido en cuenta.

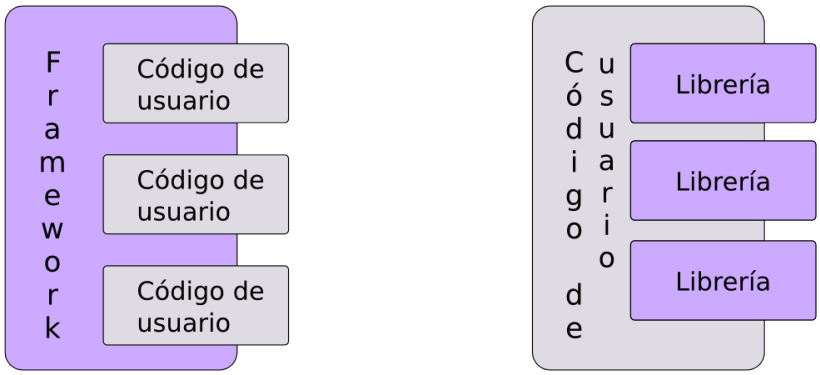


Figura 2.4: Librería vs Framework.

2.2.2 El patrón Model View Controler en frameworks web

Normalmente en el desarrollo de las aplicaciones web, mediante lenguajes de etiquetas como PHP o ASP, el diseño de la interfase, la lógica de la aplicación y el acceso a datos suelen estar agrupados

en un solo módulo. Esto conlleva un mantenimiento dificultoso y una baja interacción entre los diseñadores (artistas) y los programadores.

El patrón arquitectural MVC, *Modelo Vista Controlador* propone discriminar la aplicación en 3 capas. La interfase con el usuario (vista) se desacopla de la lógica (controlador) y ésta a su vez, del acceso a datos (modelo). Esta división favorece la reutilización de componentes.

Este patrón fue descrito por primera vez en 1979 por Trygve Reenskaug [Tryg1979], quien se encontraba entonces trabajando en Smalltalk en laboratorios de investigación de Xerox. La implementación original se describe por completo en "Programación de Aplicaciones en Smalltalk-80(TM): Como utilizar Modelo Vista Controlador" [SmallMVC].

Descripción del patrón:

- Modelo

  Esta capa define los datos, sus relaciones y la manera de acceder a estos. Se asegura de la integridad de los datos entre otras cosas. Suele permitir definir nuevas abstracciones de 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.

J2EE fue el primer framework que aplicó el concepto de MVC al desarrollo de aplicaciones web. Gran cantidad frameworks que surgieron desde entonces han aplicado en mayor o menos grado este concepto.

Componentes de un framework web MVC

Un framework web MVC suele contar con los siguientes componentes [WIKI002]

- Acceso a datos

  Los tipos de datos de los lenguajes orientados a objetos difieren de el modelo ER utilizado para la definición de bases de datos. Un ORM tiene como objetivo acortar esta brecha, permitiendo definir entidades en objetos que luego se transportan a la base de datos, además de la capacidad de CRUD y consultas sobre estos.

- Seguridad

  Existen diversos aspectos de seguridad que deben ser tenidos en cuenta cuando se implementa una aplicación web, como la validación de entrada, protección de Cross Site Scripting, etc.

- Mapeo de URLs

  Cada URL activa algún componente del controlador. La definición de esta asociación (o mapeo) generalmente está centralizada y bien especificada en un framework.

- Sistema de plantillas

  Las plantillas corresponden a la capa vista del patrón MVC. Su objetivo principal es definir la forma en la cual se muestran los datos al usuario, provenientes del controlador. Para definir la plantilla se utiliza un lenguaje básico que se integra con HTML o XML. La utilización de plantillas facilita la modificación de la fachada sin acceder al modelo o controlador.

- Caché

  La generación de ciertas páginas dentro de la aplicación puede requerir un tiempo considerable por lo que es normalmente utilizada alguna técnica de caché, que consiste en almacenar la salida durante un tiempo determinado, de manera de obtener una respuesta *cacheada*, lo que repercute en mayor velocidad. Algunos frameworks hacen uso de algún mecanismo unificado para realizar caché.

- AJAX

  Hoy en día es común la técnica de incorporar elementos dinámicos que modifican el estado de una página de manera asincrónica. Es decir partes de la página pueden realizar nuevas requerimientos al servidor cuya respuesta no fuerza una recarga de la página. Esta técnica es conocida como AJAX. Algunos frameworks como Ruby On Rails soportan AJAX directamente en el sistema de plantillas.

- Configuración mínima y simplificada

  Existen ciertos parámetros de las diversas capas que en algunos frameworks se encuentran especificados en un único archivo XML. Hoy en día se prefieren los lenguajes más "human readable" como YAML, archivos INI o código.

2.2.3 Mapeador Objeto-Relacional

Los lenguajes orientados a objetos utilizan clases, atributos y referencias para modelar el dominio de la aplicación, mientras que las bases de datos relacionales lo hacen a través de tablas y relaciones entre ellas. En el paradigma orientado a objetos una de las características que se enuncia es la persistencia, sin embargo, las implementaciones actuales de persistencias de objetos no brindan el performance ni la facilidad de consultas que ofrecen las bases de datos relacionales (RDBMS). El

lenguaje utilizado para definir y modificar los datos en los RDBMS, se conoce como SQL, el cual difiere radicalmente de la concepción de los lenguajes OO.

Es deseable utilizar una base de datos relacional para el almacenamiento. Un ORM libera la carga de la conversión explícita del lenguaje OO a SQL y logra independizar la estructura de almacenamiento del modelo de dominio.

Un ORM realiza en principio tres tipos de conversión del lenguaje OO a SQL:

- Conversión de definición de entidades.
- Creaci n, modificación y eliminación de entidades a partir de instancias de clases de modelo (persistencia).
- Definición de consultas en lenguaje OO y recuperación de datos en instancias de clases de modelo.

A través de una API bien definida un ORM se presenta como un mecanismo de presitencia de objetos.

2.3 Django

2.3.1 Introducción

Django es un framework web escrito en Python el cual se apoya en el patrón de Modelo Vista Controlador.

En principio surge como un administrador de contenido para varios sitios de noticias, luego, los desarrolladores encontraron que su CMS era lo suficientemente genérico como para cubrir un ámbito más amplio de aplicaciones.

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" [8].

En junio del 2008 fue anunciada la creación de la Django Software Fundation, la cual se encarga del desarrollo y mantenimiento.

Los orígenes de Django en la administración de páginas de noticias se ponen de manifiesto en su diseño, ya que proporciona una serie de características que facilitan el desarrollo rápido de páginas orientadas a contenidos.

Django como framework de desarrollo consiste en un conjunto de utilidades de consola (CLI) que permiten crear y manipular proyectos. Un proyecto está constituido por una o más aplicaciones. Se clasifican las aplicaciones de la siguiente manera:

- **Aplicaciones de usuario** Son aquellas que resuelven algún problema específico (ej: ventas, alquileres, blog, noticias, etc.)

[8] Por sin no les quedaba duda del fanatismo del defo, by **gisE**!!!

- **Aplicaciones genéricas** Son aquellas que resuelven problemas comunes, como la autenticación, bitácora, sindicación, etc. Algunas aplicaciones genéricas se distribuyen con Django, de las cuales resulta interesante destacar:
  - ***django.contrib.admin*** Provee de manera automática un sitio de administración que permite realizar operaciones CRUD sobre el modelo (definido a través del ORM), administrar usuarios y permisos, llevando un registro de todas las acciones realizadas sobre cada entidad (sistema de logging o bitácora).
  - ***django.contirb.comments*** Sistema de comentarios
  - ***django.contrib.syndication*** Herramientas para sindicar contenido vía RSS y/o Atom
  - ***django.contrib.gis*** Sistema de información geográfico
  - ***django.contrib.localflavour*** Localización (l10n) e internacionalización (i18n)

La concepción de MVC de Django difiere ligeramente del enfoque tradicional, se denomina al controlador vista y a la vista, template; resultando el acrónimo MTV. El controlador de MVC, se presenta en Django como conjunto de funciones y asociaciones de URLs a las mismas. Cada función de la vista delega la presentación de los datos al sistema de plantillas.

Django simplifica el patrón MVC, sin comprometer la separación de las capas. Debido a la creciente popularidad y a la sencillez del framework descripto, se seleccionó el mismo como plataforma para la implementación del desarrollo, puesto que se buscan tales características en el trabajo a realizar.

A continuación se describe de manera detallada la estructura de un proyecto realizado en Django.

2.3.2 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.

El siguiente ejemplo demuestra la creación de un proyecto:

```
$ 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 jerárquico del sistema de archivos mostraría la siguiente estructura:

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

El proyecto funciona como un contenedor de aplicaciones regidas bajo una misma configuración que comprende:

- Base de datos
- Templates
- Clases de Middleware

Se analizan a continuación la función de cada uno de estos 3 módulos.

Módulo settings

Este módulo configura globalmente el proyecto. Define las aplicaciones instaladas (*INSTALLED_APPS*), la base de datos que utilizarán (*DATABASE_**), el módulo de *urls inicial* (*ROOT_URLCONF*), la ruta en la cual se encuentran los medios estáticos y la URL donde serán publicados [9] (*MEDIA_URL*, *MEDIA_ROOT*). Otras configuraciones son idioma, zona horaria, clases de middleware [10] , ruta a los templates, etc.

El modulo settings es código fuente y es posible realizar configuraciones en tiempo de ejecución, tarea que no es factible utilizando lenguajes de marcado como XML, YAML, archivos INI, etc.

Módulo manage

Este módulo a diferencia de settings y urls, es ejecutable. Sirve como interfase con el framework. Su invocación recibe como primer argumento un nombre de comando como *startapp*, que crea una aplicación en el proyecto, *runserver* que lanza el servidor de desarrollo o *syncdb* que mediante el ORM crea el esquema en la base de datos a partir de el módulo **models** de cada aplicación.

Existen otros comandos que permiten realizar testing, iniciar la consola interactiva, administrar usuarios, etc.

Módulo urls

Este módulo define las asociaciones entre las urls y las vistas a nivel proyecto. Puede lograrse modularidad derivando el tratado de ciertas urls en módulos similares de aplicaciones instaladas en el proyecto.

[9] Del ingles "The Web framework for perfectionists with deadlines"
[10] Imágenes, sonido, flash, etc.

Cuando el usuario realiza una solicitud, Django busca una asociación que concuerde con la url dada. De encontrarla ejecuta la vista correspondiente a ésta.

La url en las asociaciones se definen mediante expresiones regulares que soportan recuperación de grupos nombrados (una subcadena identificada con un nombre). Los grupos nombrados definidos en cada asociación son argumentos de la función vista.

La asociación url-vistas se define bajo el nombre *urlpatterns*.

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

```
from mi_proyecto.personas.views import ver_personas
urlpatterns = patterns('',
                # Derivación en módulo de aplicación personas
    (r'^personas', include('mi_proyecto.personas.urls')),

    # Recuperación de datos de la url
    (r'^personas/(?P<persona_id>\d{1,4})', ver_persona),
)
```

2.3.3 El sistema de plantillas

El sistema de plantillas aparece en la última fase de la generación de la respuesta al cliente. Tiene como objetivo la separación del procesamiento de datos la presentación. Una vez hallada la vista asociada a la url de la solicitud, django ejecuta la vista. Ésta tras completar su procesamiento deriva la presentación de los datos al template o plantilla.

Django incluye por defecto funciones cargadoras de temaplates [11] que realizan la búsqueda del template que requiere la vista.

Con los datos provistos por la vista, la plantilla escogida es renderizada. Este proceso consiste en reemplazar etiquetas y ejecutar alguna lógica básica de iteración y bifurcaciones condicionales.

A continuación se presenta una simple plantilla de ejemplo:

```
<html>
        <head><title>{{title}}</title></head>

<body>

    <p>Hola {{ nombre }},</p>
```

[11] En django una clase middleware conecta los componentes de MTV, el usuario puede definir un comportamiento personalizado.

```
</body>
</html>
```

Si este template fuera “base.html”, una vista para el mismo podría ser: .. code-block:: python

def mi_vista(request): return render_to_response(‘base.html’, {‘nombre’: ‘pepe})

Esta plantilla es un HTML básico con algunas variables y etiquetas agregadas.

- **Cualquier texto encerrado por un par de llaves es una etiqueta de *variable*.** Esto le indica al sistema de templates que debe reemplazarla por el contenido de la variable entregada por la vista.
  - La representación de una variable de etiqueta puede ser alterada mediante un filtro. El cual se define mediante el caracter |. Ej:

    ```
    {{ fecha|date:"D d M Y" }}
    ```

    Si la variable fecha es un datetime, la salida es ‘Wed 09 Jan 2008’.
- Cualquier texto que esté rodeado por llaves y signos de porcentaje es una etiqueta de lógica o define algún bloque. Algunas etiquetas son if, for, extends, load entre otros.
- Es posible definir template tags (o etiquetas de template) de usuario, que actúan como una función en el template.
- Django posee un sistema jerárquico de templates que propone una estructura de herencia. Un template base puede definir una serie de bloques y entre otras cosas el diseño general de una página. Un template particular puede utilizar el tag extends y redefinir uno o más bloques, conservando intacto todo lo que se encuentre fuera de los bloques redefinidos.

Si bien los templates suelen estar orientados a la generación de HTML, es posible obtener otros formatos.

Normalmente los archivos de template se ubican en el directorio “templates” en la raíz del proyecto. Puede incluirse también un directorio “templates”” en cada aplicación del proyecto. Los TEMPLATE_LOADERS definidos en settings realizan la búsqueda cuando la vista lo requiera.

Una referencia detallada se encuentra en la documentación de Django

2.3.4 Estructura de una aplicación Django

Una aplicación es un paquete python, que consta de un módulo models y un módulo views. Para crear una aplicación se utiliza el comando **startapp** del modulo *manage* de la siguiente manera:

```
$ python manage.py startapp mi_aplicacion # Crea la aplicación
```

El resultado de este comando genera la siguiente estructura en el proyecto:

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

Como antes se mencionó puede crearse un módulo urls dentro de la aplicación para que delegue el tratado desde el urlpatterns raíz sobre este.

Módulo models

Al crear una aplicación es generado un módulos models.py en cual se definen las clases que extienden de *django.db.models.Model*. Estas definiciones son transformadas por el ORM en tablas relacionales [12].

Ej:

```
class Persona(models.Model):
    nombre = models.CharField(max_length = 50)
    apellido = models.CharField(max_length = 50)
```

Módulo views

Cada aplicación posee un módulo views, donde se definen las funciones que responden al cliente y se activan por medio del 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 obtenidos de los grupos nombrados de la url.

El siguiente ejemplo integra un fragmento de los módulos urls y views.

```
# Tras un mapeo como el siguiente
(r'^persona/(?P<id_persona>\d{1,4})/$', 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)
```

[12] Definidas bajo TEMPLATE_LOADERS en el módulo settings

2.3.5 Contenido dinámico y estático en Django

Django no se ocupa de generar contenido estático, sin embargo éste es necesario para el desarrollo completo de una aplicación web (imágenes, javascript y hojas de estilo). Django delega esta tarea al servidor web [DjangoDoc2009], sin embargo para desarrollo se provee un una vista que devuelve contenido estático (*django.views.static.serve*).

Al momento de la puesta en producción del proyecto desarrollado en Django, se deberá realizar la configuración del soporte para ejecución de Python (en la presente tesis se opto por el estudio de WSGI ya que es un estándar Pythonico), en este momento se reemplaza el uso de las static views para ser delegadas al server web, tal como antes se menciono.

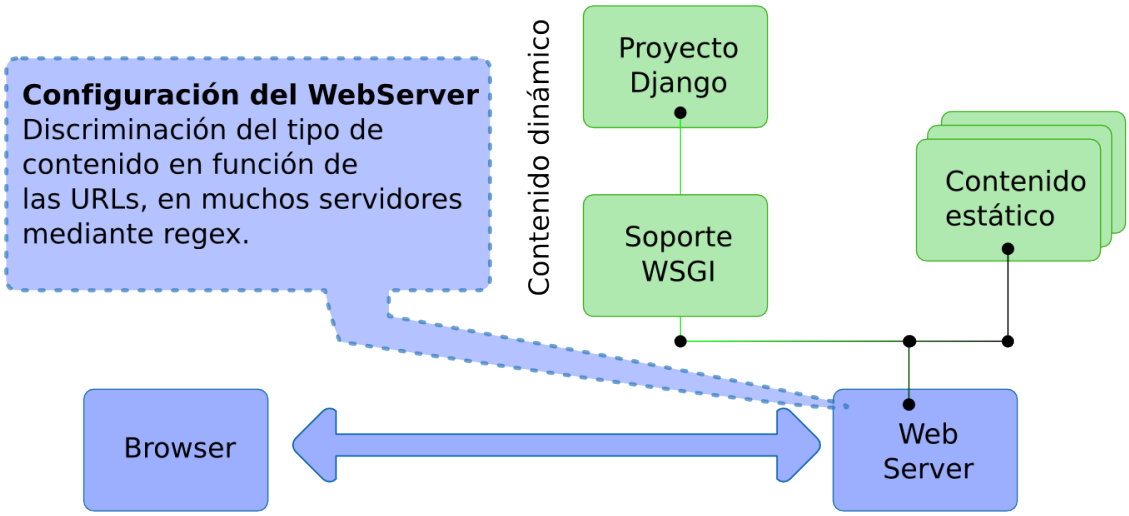


Figura 2.5: Deployment de Django

Una aplicación escrita en Django puesta en producción consta de contenido dinámico manejado por WSGI y contenido estático servido por el web server.

2.3.6 El ciclo de una petición

Se describe a continuación de manera más detallada el ciclo de una petición en Django. Durante este proceso existen clases que realizan tareas transversales, algunas de bajo y otras de alto nivel como caché, sesión y autenticación, manejo de cabeceras HTTP, etc. y reciben el nombre de clases de Middleware.

Según que métodos se implementen pueden anteponerse o anexarse a una o más etapas del proceso del request.

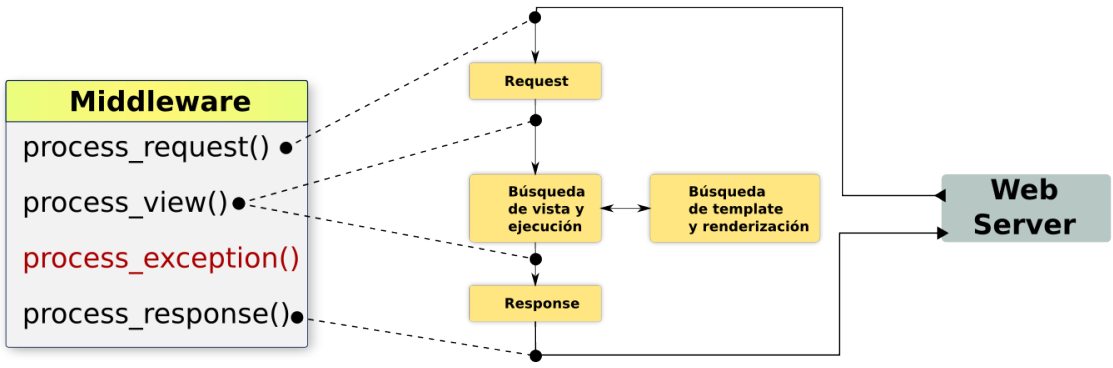


Figura 2.6: Intracción de los métodos de una clase de middleware en el proceso del request

El ciclo de una petición se completa mediante los siguientes pasos:

1. **Entrada por Middlewares de Request** El primer componente en tratar la solicitud es el conjunto de middlewares de request. Se encargan de envolver la petición en una instancia de la clase HttpRequest (*django.middleware.common.CommonMiddleware*), adicionar la sesión al request (*django.contrib.sessions.middleware.SessionMiddleware*) y la relación entre la sesión y el usuario de la aplicación de autenticación (*django.contrib.auth.middleware.AuthenticationMiddleware*).

2. **URLConf y View Middlewares** El siguiente paso es la búsqueda de la vista a partir de la url de la solicitud (que se encuentra en el atributo *path* de la instancia de *HttpRequest* generada por el middleware de request) en los patrones de urls asociados a las vistas. La constante ROOT_URLCONF (del módulo settings del proyecto) determina donde comienza dicha búsqueda. Una vez obtenida la vista, los middleware de vistas pueden posicionar su ejecución antes o después de ésta.

   Un middleware de vista importante es el transacciones. Previa a la ejecución de la vista inicia una transacción, al completarse de manera exitosa dicha vista se realiza un COMMIT. De existir algún error se cierra la transacción con un ROLLBACK.

3. **Salida mediante Middleware de Response** Finalmente, cunado la vista a generado una respuesta, ésta es procesada por los middlewares de respuesta (*Response Middlewares*). Algunas tareas que se pueden realizar con este tipo de middlewares es compresión de la salida, inclusión de librerías de javascript muy modulares, como YUI, etc.

4. **Middlewares de Excepciones** De lanzarse una excepción que no es capturada en la vista, modelos o urls, el middleware de excepciones la captura y genera una respuesta informando el error (cuando la bandera DEBUG del modulo settings está activada, se genera un traceback del error).

Las clases de middleware están provistas por Django y son incluidas durante la generación del proyecto en el módulo settings. Una clase de middleware puede implementar uno o más métodos de los mostrados en la *figura*

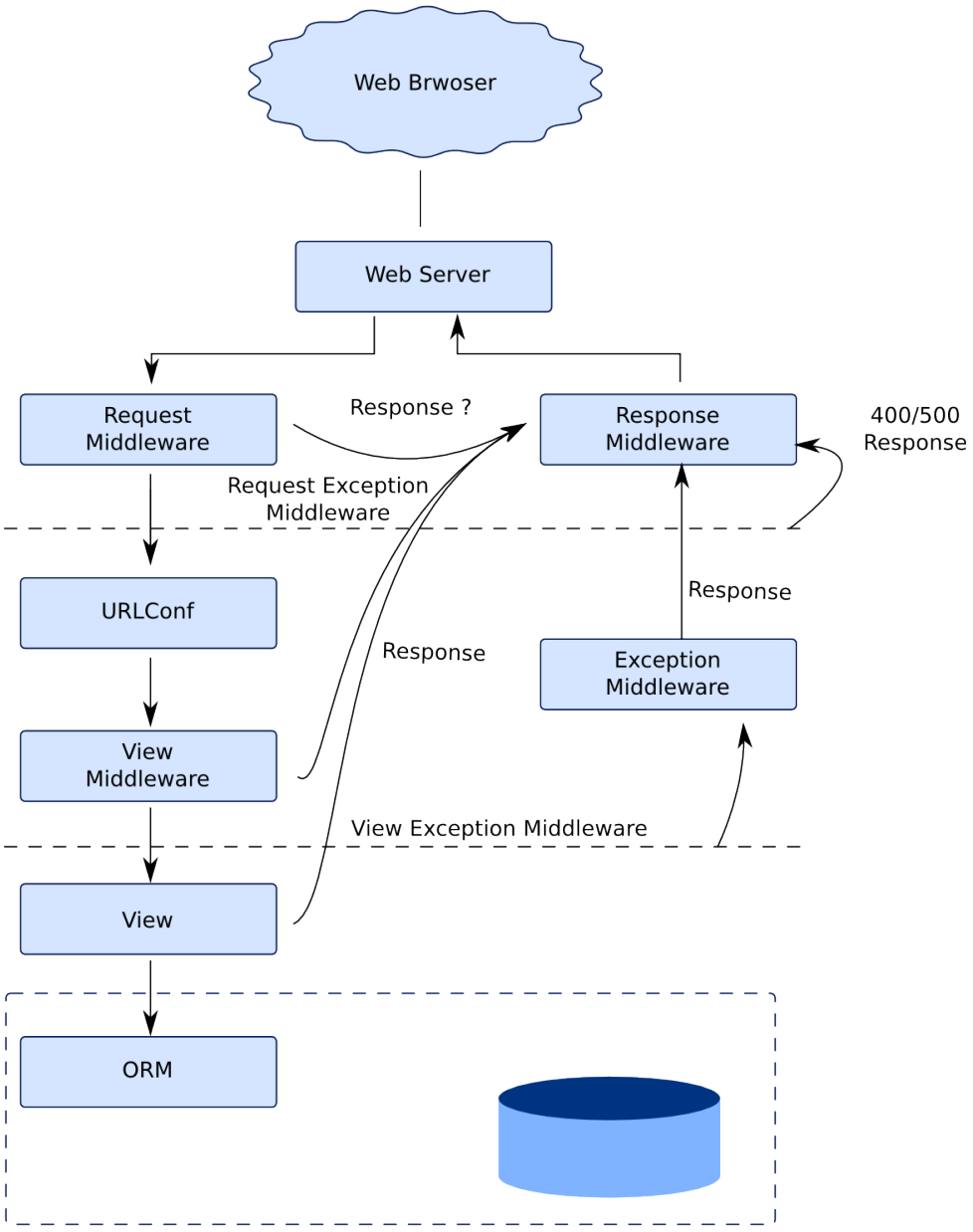


Figura 2.7: Vista alternativa del procesamiento de un request con la interacción de los middlewares.

2.3.7 La API de modelos (ORM)

Un modelo de Django es una descripción de alto nivel de la estructura de la base de datos. Esta descripción se realiza en lenguaje Python en el módulo models.py de cada aplicación, en vez de realizarse en SQL.

De contarse con una base de datos preexistente, Django permite inferir la definición de los modelos, con el fin de adaptar el ORM. Dentro del módulo se define cada entidad como una clase que extiende a *django.db.models.Model*.

Si bien la entidad queda identificada con el nombre de clase, pueden existir colisiones entre nombres iguales en diferentes aplicaciones. Por esto, el *nombre completo de las entidades* se define como una cadena compuesta por el nombre de la aplicación y el nombre de la entidad unidos por un punto (Ej: `"personal.Persona"`, `"auth.User"`, `"ventas.Producto"`, etc.).

Nota: Nombre *Modelo*:

> Si bien se utilizó el término *entidad* para definir los integrantes del modelo de una aplicación, es posible utilizar la palabra *modelo* en singular.

Las tareas del ORM son:

- **Creación de Tablas Relacionales** Esta operación realiza mediante el DDL [13] de SQL. Se toma como nombre de tabla el nombre completo de entidad reemplazando el punto por un guión bajo. Si existiesen relaciones que requieran tablas intermedias, son creadas.

  Esta operación se activa mediante el comando *syncdb* del modulo *manage* del proyecto.

  En el *ejemplo* el ORM se encarga de generar el siguiente código SQL al momento de la ejecución del comando syncdb.

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

- **CRUD**

  - **Creación, Modificación y Eliminación** La creación de entradas en la base de datos se realiza cuando una instancia de una clase del modelo recibe el mensaje `save()` (*INSERT*). Sobre una instancia existente, el mensaje `save()` actualiza la entidad con los valores (*UPDATE*) Cuando una instancia existente en la base recibe el mensaje `delete()` se elimina (*DELETE*).

  - **Recuperación (Administradores de consultas)** La recuperación de datos de una base de datos relacional se realiza típicamente mediante la sentencia SQL *SELECT*, pero los criterios de búsqueda pueden llegar a ser complejos, sobre todo

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

cuando existen relaciones, por esto, Django agrega un objeto (manager) que simplifica la tarea de recuperación de instancias. Existe un manager por cada entidad y cada relación que esta posea.

- **Sistema de señales** El sistema de señales permite registrar funciones para ser ejecutadas antes o después de ciertos eventos, como la creación de instancias del modelo, la eliminación, o la creación del esquema (*syncdb*). Son equivalentes a triggers en la base de datos.
- **Inclusión de metadatos** Existen ciertos metadatos que no pueden incluirse fácilmente en SQL de manera estándar, como el nombre visible para el usuario de una entidad, en plural y en singular, o la url absoluta del elemento en el sistema. Esta información puede almacenarse mediante una clase interna Meta y métodos de instancia.
- **Herencia de objetos** Django permite crear jerarquías de herencia en los modelos.
- **Relaciones genéricas** Si bien en los RDBMS se requiere especificar el tipo de los campos relacionados (tipado estático fuerte) al momento de definir una relación, mediante el field `GenericRelation``(provisto por la aplicación genérica ``ConentType` [DjangoDocsConentType2009]), se pueden realizar relaciones contra instancias de cualquier tipo.

Claves en los modelos

Los modelos en Django poseen un solo campo clave. Si ningún campo es definido como clave, con el argumento `primary_key = True`, se agrega de manera automática un campo `id` de tipo entero auto incremental [DjangoDocsModelsKey2009] .

Septiembre 2009, http://docs.djangoproject.com/en/dev/ref/models/instances/#the-pk-property

El atributo `pk` es un alias del campo `id` o del campo que se halla definido como clave primaria.

Django no soporta claves compuestas, pero permite definir combinaciones únicas de campos para suplir esta carencia. Dentro de la clase de metadatos `Meta` se pueden definir campos `unique` y `unique_together`.

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

class Meta:
        unique_together = ('nombre', 'apellido', )
```

Acceso a la API de modelos desde la consola interactiva

Uno de los comandos provistos por el módulo manage es `shell`, que invoca el intérprete de python de manera interactiva agregando al PythonPath el proyecto. De esta manera se pueden importar las aplicaciones, vistas, y modelos.

El siguiente listado ejemplifica la importación de la clase Persona desde la aplicación `mi_aplicacion`:

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

El ejemplo anterior permite observar los siguientes puntos

1. **Para crear un objeto, se importa la clase del modelo apropiada y se crea** una instancia asignando valores para cada campo.
2. Para guardar el objeto en la base de datos, se usa el método `save()`.
3. Para recuperar objetos de la base de datos, se usa `Persona.objects`, que constituye el manager de la entidad.

Managers

Los managers se encargan de realizar las querys sobre las entidades de la base de datos. Presentan una API para realizar las consultas y devuelven instancias de objetos `QuerySet`. Un QuerySet representa una consulta *lazy*, es decir, la consulta que se hace efectiva cuando es necesario acceder a los datos[*]_.

Dado el siguiente ejemplo:

```
    from django.db import models

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

class Vehiculo(models.Model):
    marca = models.CharField(max_length = 4)
    modelo = models.DateField()
    propietario = models.ForeignKey(Persona)
```

Los managers están presentes en:

- En el atributo `objects` de cada entidad del modelo.

El programador puede definir nuevos managers para consultas frecuentes.

Ej:

```
Persona.objects.all() # Recupera todas las personas
Vehiculo.objects.all() # Recupera todos los vehículos
```

- En cada relación, ya sea `1 a N` o `N a N`

  Ej:

```
p = Persona.obects.get(nombre = "nahuel")
Vehiculo.objects.filter( propietario = p )

# O lo que es lo mismo
Persona.obects.get(nombre = "nahuel").vehiculo_set() # Relación inversa re
```

API de los Managers

La API provista por los managers consiste en dos grupos de métodos: los que retornan instancias de QuerySet y los que no. A continuación se listan los métodos que constituyen la API. Si son invocados sobre un manager devuelven un QuerySet, y si son ejecutados sobre un QuerySet, devuelven otra instancia del mismo tipo.

El primer grupo está definido como sigue:

- **`filter( **argumentos )`** [14] Solo incluye las instancias que cumplen con el criterio definido en `argumentos`
- **`exclude( **argumentos )`** Excluye elementos que cumplan con el criterio definido en argumentos
- **`order_by(*campos)`** Ordena el resultado por el/los campos `campos`
- **`distinct()`** Solo admite una instancia de cada elemento.
- **`values(*campos)`** Retorna solo los campos `campos` como un diccionario.
- **`dates(campo, tipo, order='ASC')`** Retorna fechas
- **`none()`** Evalua a QuerySet vacío.
- **`select_related()`** Preselecciona en la consulta los datos relacionados para evitar el overhead de múltiples consultas sobre QuerySets grandes y con muchas relaciones.
- **`extra(select=None, where=None, params=None, tables=None)`** Permite realizar una consulta de bajo nivel

El conjunto que no devuelve ‘‘QuerySet‘‘s es el siguiente:

- **`get(**campos)`** Obtiene una única entidad identificada por `campos`
- **`create(**campos)`** Crea una entidad basada con los valores `campos`
- **`get_or_create(**kwargs)`** Crea una entidad basada con los valores `campos` o la modifica si existe.
- **`count()`** Retorna la cantidad de elementos del QuerySet.
- **`in_bulk(lista_de_identificadores)`** Retorna un diccionario donde la clave es cada identificador y el valor es la instancia.
- **`latest(campo = None)`** Retorna la última instancia por el campo `campo`. Debe ser del tipo DateField.

2.3.8 Formularios

El elemento de entrada tradicional de las aplicaciones web son los formularios. En un sistema de información, la forma de realizar las operaciones CRUD es a través de estos.

El ciclo de trabajo normal para la entrada de datos generalmente es:

El usuario:

1. Carga una página en la cual existe un formulario
2. Completa los campos (pueden existir campos opcionales)
3. Envía el formulario, utilizando típicamente un botón, el navegador

el cual se encarga de codificar los datos y enviarlos a la URL indicada en el campo "action" del :ref:markup del formulario.

El servidor: Recibe los datos y los valida. Si los datos son correctos, generalmente se envía una respuesta 300 indicando que la carga ha sido exitosa. Por el contrario si la validación es incorrecta se devuelve al usuario el formulario con los datos que hallan sido válidos, y se muestran los mensajes pertinentes de los errores que hallan ocurrido, volviendo al paso 2.

La validación de los formularios puede ser una tarea compleja, Django provee un mecanismo para generar formularios en el módulo *django.forms* que ameniza la tarea.

Un formulario es una entidad que posee un conjunto de campos, cada uno de los cuales tiene la responsabilidad de validar los propios datos. A posteriori el formulario realiza una comprobación de la coherecia global, tras la cual se obtiene el resultado de la validación.

Mediante los *widget* de los campos que componen el formulario, este puede de renderizarse como HTML.

Hasta acá

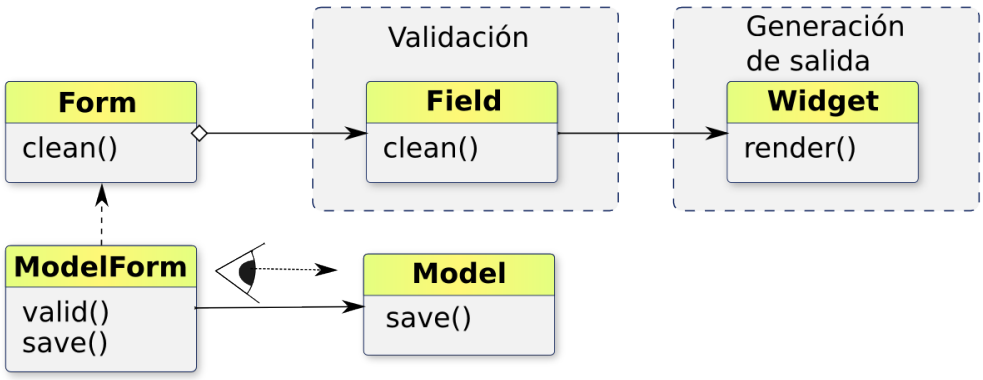


Figura 2.8: Estructura de objetos de los formularios en Django

CAPÍTULO 3

Tecnologías del cliente

3.1 Web dinámica desde la perspectiva del cliente

Desde la perspectiva de un usuario, la Web consiste en una enorme cantidad de documentos interconectados a nivel mundial llamados *paginas Web*. En el cliente las paginas se ven mediante un programa llamado navegadores

JavaScript ha tenido durante mucho tiempo la reputación de lenguaje torpe e inadecuado para el desarrollo serio. Esto ha sido en gran parte debido a las implementaciones incompatibles del lenguaje y del DOM en varios navegadores, y al uso extenso de código "pegado" lleno de errores por parte de los aficionados. Los errores en tiempo de ejecución eran tan frecuentes y difíciles de solucionar que pocos programadores intentaban corregirlos.

La reciente aparición de navegadores con un soporte adecuado de los estándares web, frameworks para JavaScript tales como Prototype, y herramientas de depuración de alta calidad han facilitado enormemente la creación de código organizado y escalable en JavaScript, y la aparición de AJAX lo ha hecho esencial. Mientras hasta hace poco JavaScript era solo utilizado para las tareas relativamente simples y no críticas (validación de formularios y decoraciones llamativas), actualmente se está utilizando para escribir código complejo que a menudo es responsable de buena parte de la funcionalidad básica de un sitio. Los errores en tiempo de ejecución y el comportamiento imprevisible ya no son molestias de menor importancia; son errores fatales.

3.2 Estructura de un navegador

3.2.1 Navegador Web

Nota: Acá abordamos desde la perspectiva del navegador HTTP, HTML y JavaScript

Un navegador web o *web browser*, es un software encargado de representar documentos de hipertexto al usuario, siendo los lenguajes de codificación de hipertexto más populares HTML y XHTML. Un navegador no solo representa los documentos de hipertexto, sino que puede representar otros tipos de documentos, como imágenes (:term:JPEG, :term:GIF, :term:PNG, etc.), sonido (:term:WAV, :term:MP3, :term:OGG) y contenido multimedia como vídeo (:term:MPEG, :term:H264, :term:RM, :term:MOV), e interactivos como es el caso de Macromedia Flash, applets Java o controles ActiveX en la plataforma Windows. Debido a la cantidad de recursos que debe manejar un navegador, el servidor web agrega a cada respuesta al cliente una cabecera donde le indica el tipo de recurso que está entregando. Esta especificación se realiza con el estándar :term:MIME.

Un navegador web acepta como entrada del usuario una URL, comúnmente conocida como dirección de Internet. Una vez validada la URL, el navegador web descarga el recurso apuntado por la URL mediante el protocolo HTTP.

Una URL tiene el siguiente esquema, donde podemos diferenciar varios componentes

Figura 3.1: Forma de una URL.

Los componentes de una URL son:

- esquema

  Especifica el mecanismo de comunicación. Generalmente HTTP y HTTPS en una comunicación asegurada mediante TLS [1] .

- anfitrión

  Especifica el nombre de dominio del servidor en Internet, por ejemplo: *google.com*, *nasa.gov*, *wikipedia.com*, etc. Se popularizó la utilización de el subdominio "www" para identificar el anfitrión que ejecuta el servidor web, dando lugar a direcciones del tipo *www.google.com*, *www.nasa.gov*, etc.

  El *puerto* es un parámetro de conexión TCP, y suele ser omitido debido a que el esquema suele determinarlo, siendo 80 para HTTP y 443 para HTTPS.

- recurso

  Especifica dentro del servidor, la ruta para acceder al recurso

- query

[1] *Transport Security Layer* es el sucesor de *Secure Socket Layer* (SSL),

El parámetro query tiene sentido cuando el recurso apuntado por la ruta no se trata de una página estática y sirve para el pasaje de parámetros. El programa que genera el recurso puede recibir como argumentos estos parámetros, por ejemplo, cuando se ingresa la palabra *foo* en el buscador google, la URL que provee el resultado de la búsqueda es la siguiente:

```
http://www.google.com/search?*q=foo*
```

- enlace

  Dentro de un documento de hipertexto pueden existir destinos de enlaces, o enlaces internos. Gracias a este parámetro se puede enlazar a una sección específica de un documento, permitiendo al navegador ubicarse visualmente.

es un protocolo criptográfico que provee conexiones seguras a través de una red, típicamente Internet [WikiSSL2009] .

Agosto 2009.

Cuando el recurso apuntado se encuentra en Internet, el navegador realiza una conexión hacia el servidor web indicado por el dominio y mediante el protocolo HTTP le informándole a que recurso debe acceder.

3.2.2 HTTP

Para acceder al recurso *~ndefosse/introducion_lenguaje_python.html* en el servidor *students.unp.edu.ar*, de la url http://students.unp.edu.ar/~ndefosse/introducion_lenguaje_python.html, el navegador conforma la siguiente consulta:

```
GET /~ndefosse/introduccion-lenguaje-python.html HTTP/1.1
Host: students.unp.edu.ar
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-ar,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Cookie: user_id=G7NVG5YY51I9DZAIJDEDQIXYQSRF0CTL
```

Esto conforma lo que se conoce como una consulta HTTP o *HTTP Request*. En la primera linea se conforma de el método HTTP y el nombre del recurso, finalizando con la versión del prtocolo que soporta el navegador (o cliente):

```
GET /~ndefosse/introduccion-lenguaje-python.html HTTP/1.1
```

La sgunda linea es el host al cual se accede. Un mismo servidor web puede estar publicado en varios dominos, mediante esta linea se puede discriminar desde cual se intenta acceder al recurso:

```
Host: students.unp.edu.ar
```

El siguiente componente del *request* es la linea que identifica al cliente, en este caso el navegador informa que se trata de Mozila, versión 5:

```
User-Agent: Mozilla/5.0
```

Una vez que el servidor web a localizado y accedido al recurso, precede a enviar la respuesta

3.2.3 HTML

HTML es un lenguaje de marcado que teiene como objetivo describir un documento de hipertexto. Un documento HTML se conforma por una serie de *tags* o etiquetas.

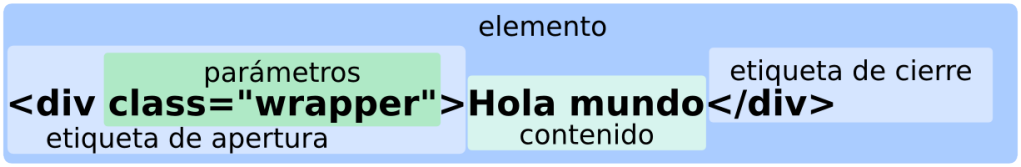


Figura 3.2: Una etiqueta HTML.

Un documento HTML está delimitado por las etiquetas o *tags* html y contiene una cabecera delimitada por *head* y un cuerpo, delmitado por *body*.

```
<html>
        <head>
                <title>Mi pagina</title>
        </head>
        <body>
                <h1>Título principal</h1> <!-- comentario -->
                <p>Párrafo</p>
        </body>
</html>
```

Un documento HTML contener enlaces a recursos entendibles para el navegador, como los enlaces a hojas de estilos o código javascript.

La inclusión de una hoja de estilo se realiza mediante el tag *link*, de la siguiente manera:

```
<link type="text/css" rel="stylesheet" href="hoja_de_estilos.css">
```

Se puede además embeber en la página el estilo, de la siguente manera

```
<style type="text/css">

        BODY {
                font-family: "Verdana";
                font-size: 12pt;
                padding: 2px 2px 3px 2px;
        }
</style>
```

Al haberse incrustado el estilo en una página en particular, este solo tiene validés para ese recurso en particular.

Otro tipo de recurso entendible para el navegador constituye la inclusión de código de ejecución en el clinete, mediante el tag script.

```
<script type="text/javscript" src="mi_codigo.js"></script>
```

Además de la inclusión de recursos javascript externos, el código javascript se puede embeber en el código html de varias maneras [StephenChapmanJS2009], entre ellas:

```
<script type="text/javascript">
<!-- // Javascript en un comentario, para pasar validadores XHTML y ocultar
```

de navegadores

// que no soporten Javascript var x = 2; var y = 4;

// –> </script>

último acceso Agosto 2009, http://javascript.about.com/library/blxhtml.htm

3.2.4 CSS

Una hoja de estilo en cascada especfica como se va a mostrar un documento en pantalla, como se va a imprimir o inclusive como se realiza la pronunciación a través de un dispositivo de lectura [W3cCSS2009] .

El objetivo de CSS es separar el contendio de la presentación de un documento HTML o XML. Una hoja de estilo puede ser enlazada desde varias paginas, permitiendo mantener coherencia y consistencia en el estilo.

http://www.w3c.es/divulgacion/guiasbreves/HojasEstilo

3.2.5 JavaScript

Javascript es un lenguaje de programación interpretado creado originalmente por Brendan Eich para la empresa Netscape con el nombre de Mocha. El lenguaje surge a principios de 1996, como un lenguaje de scripting para la Web y apuntado a la interacción directa con el usuario.

Con una sintaxis semejante a la de Java y C, JavaScript dista mucho de ser Java y debe su nombre más a cuestiones de marketing que a principios de diseño, de hecho se reconocen mayores influencias de lenguajes como Self, Scheme, Perl e incluso en versiones modernas de Python.

En junio de 1997 la European Computer Manufacturers' Association ECMA, adopta como un estándar a JavaScript, con el nombre de ECMAScript. Este importante paso marco un inicio para la compatibilidad entre navegadores y tras él las empresas comenzaron a desarrollar sus propias versiones del lenguaje, JScript es la implementación de Microsoft, muy similar al la de Netscape, pero con ciertas diferencias en el modelo de objetos del navegador [2]; con la finalidad de evitar la incopatibilidad que se avecinaba, el World Wide Web Consortium diseñó el estándar Document Object Model (DOM) que en conjunto con el lenguaje resuelven la modificación de la página Web en el cliente. La batalla entre los navegadores se continua luchando y probablemente de para unos cuantos años más, pero estos dos estandares representan la base para un desarrollo web compatible.

En la actualidad, todo computador personal en el mundo tiene, al menos, ún interprete de JavaScript instalado y activo en el. A pesar de su popularidad, solo algunos saben que JavaScript es un muy buen lenguaje de programación dinámico, orientado a objetos y de proposito general.

Incluir JavaScript en un documento

Cualquier documento HTML puede incluir JavaScript y el método correcto que define la W3C es incluir javascript como un archivo externo, tanto por cuestiones de accesibilidad, como practicidad y velocidad en la navegación. Para esto basta con escribir en el documento HTML:

```
<script type="text/javascript" src="[URL]"></script>
```

Siendo [URL] la URL relativa o absoluta del recurso con código JavaScript. Cuando el navegador descarga el documento y comienza su lectura al encontrar esta etiqueta solicita al servidor el archivo referenciado por URL y lo interpreta, para continuar luego con la lectura del resto de las etiquetas.

El otro método para incluir código, es hacerlo directamente en el documento entre los elementos <script> y </script>, aunque esta practica no es la recomendada:

```
<script type="text/javascript">
<!--
    // código JavaScript
```

[2] Por que sera?.

```
-->
</script>
```

De forma similar a la anterior, el navegador comienza con la interpretacion del codigo al encontrar la etiqueta y luego continua con la lectura del resto de las etiquetas.

Orientado a Objetos

JavaScript es un lenguaje orientado a objetos, es más, todo en este lenguaje es un objeto. Normalmente a muchos programadores de Java o C++ les cuesta comprender la implementación de objetos que provee JavaScript y esto es así porque el lenguaje esta orientado a prototipos.

Algunos argumentan que JavaScript no es verdaderamente orientado a objetos porque no provee ocultación de la información. Pero resulta ser que los objetos de JavaScript pueden tener variables privadas y métodos privados gracias a las clausuras.

A continuacion se explica como crear un objeto:

```
uno = new Object();
dos = {};
```

En este primer ejemplo se ver que hay dos formas de crear objetos, en particular la segunda forma hace uso de una característica propia del lenguaje para declarar estructuras de datos y resulta de mucha utilidad en los casos de intercambio de mensajes mediante el estándar *JSON*. Otros tipos de objetos pueden ser declarados de esta manera, como:

```
cadenas = "";
arreglos = [];
funciones = function() {};
```

Las caracteristicas dinámicas de lenguaje posibilitan la modificación de los objetos en tiempo de ejecución, esto quiere decir que a un objeto, luego de ser creado, se le pueden agregar o eliminar métodos y propiedades:

```
uno.variable = “hola”;
dos.metodo = new Function();
```

De esta forma se tienen dos objetos personalizados uno con un “atributo” y otro con un “metodo”, de esta manera se pueden continuar agregando miembros de cualquier tipo y hacer crecer los objetos. Sin embargo, crear objetos así no es cómodo, ya que si se quisiéra crear una copia de este objeto se deberían declarar nuevamente todos sus miembros.

Creando ‘Clases’:

La forma en que los lenguajes orientados a objetos comúnmente resuelven el problema anterior es mediante el uso de clases, en JavaScript no es posible declarar clases, pero si es posible instanciar objetos a partir de un constructor.

El objeto Function se utiliza como objeto instanciable en JavaScript, y el cuerpo de la función es el constructor del objeto. Para emular el comprortamiento propio de una clase el método mas difundido es aprovechar el funcionamiento de la plabra reservada *this* dentro del constructor y las funciones miembro. Cuándo una función es llamada con el operador *new*, *this* hace referencia al objeto que será retornado.

```
/*clase de ejemplo*/

Clase = function() {
    this.propiedad = "hola!";
    this.metodo = function(){
        alert(this.propiedad);
    }
}
objeto = new Clase(); // instanciamos 'Clase'
```

Podria parecer que aquí se terminan los problemas pero no es así, esta forma de crear clases provoca que cada objeto tenga una versión unica de sus 'metodos', cuando lo ideal seria que todos los objetos compartieran la misma función.

La propiedad 'Prototype':

Todas las funciones tienen una propiedad llamada prototype, esta propiedad es un objeto que será utilizado como 'modelo' inicial de todos los objetos que sean creados con esta función cuando sea utilizada como constructor.

Reescribiendo el ejemplo anterior el código quedaría así:

```
/* creamos un constructor limpio */

Clase = new Function();
Clase.prototype.propiedad = "hola!";
Clase.prototype.metodo = function(){
    alert(this.propiedad);
}
objeto = new Clase(); // instanciamos 'miClase'
```

Ahora todos los objetos creados a partir de 'Clase' comparten inicialmente las mismas referencias en todas sus propiedades, esto significa que, todos los objetos van a compartir la misma versión de 'metodo'.

Adicionalmente, al utilizar la propiedad prototype, se obtiene otra ventaja y es poder usar la palabra reservada instanceof para determinar si un objeto es instancia de un constructor.

> http://javis.wordpress.com/2006/10/23/javascript-orientado-a-objetos/

3.3 Evolución del JavaScript

3.3.1 Herencia

JavaScript puede no tener herencia orientada a clases, pero tiene herencia "prototipal". Existen muchos autores que implementan su forma de emular la herencia en JavaScript y entre los métodos mas utilizados se detallan a continuacion dos prinsipios basicos ampliamente difundidos.

Object masquerading

Este método saca provecho del comportamiento de la palabra reservada this dentro de los constructores. El funcionamiento es el siguiente: Un constructor asigna propiedades y métodos a un objeto referenciándolo con la palabra clave this, como un constructor es simplemente una función, se puede usar el constructor de una clase A como método de una clase B.

```
function ClaseA(nombre){
    this.nombre = nombre;
    this.identificarse = function(){
        alert(this.nombre);
    }
}
```

Recordemos que en un constructor, this hace referencia al nuevo objeto que será retornado. Pero en un método, this hace referencia al objeto desde el cual fue llamado.

```
function ClaseB(nombre){
    this.superClase = ClaseA;
    this.superClase(nombre);
    delete this.superClase;
}
```

En el código anterior, el constructor 'ClaseA', es llamado como método del nuevo objeto que se esta creando en 'ClaseB', por lo tanto, todas las propiedades y métodos que se crean en ClaseA se van a agregar al nuevo objeto de ClaseB.

Quizás el punto mas interesante del Object masquerading, frente a otros métodos de emular herencia, es que soporta la herencia múltiple, esto significa que un objeto puede heredar de mas de una clase al mismo tiempo. Solo basta con llamar a cuantos constructores sean necesarios dentro del constructor de la clase hija.

Prototype chaining

Anteriormente se mostro como definir clases utilizando el objeto prototype. Prototype chaining se basa en este objeto y es el método recomendado por el standard ECMA Script. Prototype es una

propiedad del objeto Function, que actúa como un template sobre el cual se van a crear nuevos objetos. Mas precisamente, las propiedades y métodos del objeto prototype van a ser pasados a todas las instancias de esa clase.

El ejemplo anterior utilizando prototype chaining quedaría de la siguiente manera:

```
function ClassA() {}
ClassA.prototype.nombre = "";
ClassA.prototype.identificarse = function() {
    alert(this.nombre);
}
function ClassB() {}
ClassB.prototype = new ClassA();
```

La última línea del ejemplo muestra el funcionamiento del prototype chaining. Lo que ocurre es que asignamos al prototipo de 'ClassB' una nueva instancia de 'ClassA'. De ahora en adelante, todos los objetos creados con 'ClassB' van a tener, también, los mismos métodos y propiedades de la instancia de 'ClassA'. Y si queremos agregar mas métodos y propiedades, lo único que tenemos que hacer es agregárselos al prototype de 'ClassB'.

Lo malo de este método para emular herencia, es que no se puede pasar parámetros a la clase base, como hicimos en el ejemplo de Object masquerading.

Lo bueno, es que el operador instanceof funciona de una manera única: por cada instancia de ClaseB, instanceof nos retorna true tanto con 'ClaseA' como con 'ClaseB':

```
var miobjeto = new ClassB();
alert(miobjeto instanceof ClassB); // true
alert(miobjeto instanceof ClassA); // true
```

3.3.2 AJAX

La expansión y masificación de la Web, llevó a que el ciclo de petición respuesta en el navegador resultara insuficiente para proveer al usuario de aplicaciones interactivas o :term:´RIA´, por este motivo se comenzo a trabajar sobre nuevas técnica de desarrollo que culminaron el en nacimiento de AJAX, acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML).

Una aplicacion Web que trabaje con AJAX se ejecutan en el cliente o navegador y mantiene una comunicación asíncrona con el servidor en segundo plano. De esta forma le es posible realizar cambios sobre las páginas sin necesidad de recargarla, lo que significa un aumento de la interactividad, velocidad y usabilidad en las aplicaciones.

AJAX es una tecnología asíncrona, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. JavaScript es el lenguaje en el que normalmente se efectúan las funciones de llamada de

AJAX mientras que el acceso a los datos se realiza mediante XMLHttpRequest. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.

En la practica AJAX no constituye una tecnología en sí, sino que es un término que engloba a éstas cuatro tecnologías:

- XHTML (o HTML) y hojas de estilos en cascada (CSS) para el diseño que

acompaña a la información.

- Document Object Model (DOM) accedido con un lenguaje de scripting por

parte del usuario, especialmente implementaciones ECMAScript como JavaScript y JScript, para mostrar e interactuar dinámicamente con la información presentada.

- El objeto XMLHttpRequest para intercambiar datos de forma asíncrona con el

servidor web. En algunos frameworks y en algunas situaciones concretas, se usa un objeto iframe en lugar del XMLHttpRequest para realizar dichos intercambios.

- XML es el formato usado generalmente para la transferencia de datos

solicitados al servidor, aunque cualquier formato puede funcionar, incluyendo HTML preformateado, texto plano, JSON y hasta EBML.

Funcionamiento de AJAX

Se crea y configura un objeto XMLHttpRequest. # El objeto XMLHttpRequest realiza una llamada al servidor. # La petición se procesa en el servidor. # El servidor retorna un documento XML que contienen el resultado. # El objeto XMLHttpRequest llama a la función callback() y procesa el resultado. # Se actualiza el DOM (Document Object Model)de la página asociado con la petición con el resultado devuelto

Las ventajas de AJAX

- Mayor interactividad
- Recuperación asíncrona de datos, reduciendo el tiempo de espera del usuario
- Facilidad de manejo del usuario
- El usuario tiene un mayor conocimiento de las aplicaciones de escritorio
- Se reduce el tamaño de la información intercambiada
- Portabilidad entre plataformas
- No requieren instalación de complementos, applets de Java, ni ningún otro

elemento * Código público

Las desventajas de AJAX

- Usabilidad: Comportamiento del usuario ante la navegación
- Botón de volver atrás del navegador
- Empleo de iframes ocultos para almacenar el historial
- Empleo de fragmento identificador del URL ('#') y recuperación mediante

JavaScript * Problema al agregar marcadores/favoritos en un momento determinado de la aplicación * Problemas al imprimir páginas renderizadas dinámicamente * Tiempos de respuesta entre la petición del usuario y la respuesta del servidor * Empleo de feedback visual para indicar el estado de la petición al usuario * Requiere que los usuarios tengan el JavaScript activado en el navegador * En el caso de Internet Explorer 6 y anteriores, que necesita tener activado el ActiveX

Nota: terminar este tema

A pesar de que el término «AJAX» fuese creado en 2005, la historia de las tecnologías que permiten AJAX se remonta a una década antes con la iniciativa de Microsoft en el desarrollo de Scripting Remoto. Sin embargo, las técnicas para la carga asíncrona de contenidos en una página existente sin requerir recarga completa remontan al tiempo del elemento iframe (introducido en Internet Explorer 3 en 1996) y el tipo de elemento layer (introducido en Netscape 4 en 1997, abandonado durante las primeras etapas de desarrollo de Mozilla). Ambos tipos de elemento tenían el atributo src que podía tomar cualquier dirección URL externa, y cargando una página que contenga javascript que manipule la página paterna, pueden lograrse efectos parecidos al AJAX.

El Microsoft's Remote Scripting (o MSRS, introducido en 1998) resultó un sustituto más elegante para estas técnicas, con envío de datos a través de un applet Java el cual se puede comunicar con el cliente usando JavaScript. Esta técnica funcionó en ambos navegadores, Internet Explorer versión 4 y Netscape Navigator versión 4. Microsoft la utilizó en el Outlook Web Access provisto con la versión 2000 de Microsoft Exchange Server.

La comunidad de desarrolladores web, primero colaborando por medio del grupo de noticias microsoft.public.scripting.remote y después usando blogs, desarrollaron una gama de técnicas de scripting remoto para conseguir los mismos resultados en diferentes navegadores. Los primeros ejemplos incluyen la librería JSRS en el año 2000, la introducción a la técnica imagen/cookie[1] en el mismo año y la técnica JavaScript bajo demanda (JavaScript on Demand)[2] en 2002. En ese año, se realizó una modificación por parte de la comunidad de usuarios[3] al Microsoft's Remote Scripting para reemplazar el applet Java por XMLHttpRequest.

Frameworks de Scripting Remoto como el ARSCIF[4] aparecieron en 2003 poco antes de que Microsoft introdujera Callbacks en ASP. NET.[5]

Desde que XMLHttpRequest está implementado en la mayoría de los navegadores, raramente se usan técnicas alternativas. Sin embargo, todavía se utilizan donde se requiere una mayor compatibilidad, una reducida implementación, o acceso cruzado entre sitios web. Una alternativa, el Terminal SVG[6] (basado en SVG), emplea una conexión persistente para el intercambio continuo entre el navegador y el servidor.

3.3.3 JSON

JSON brinda 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 JSON http://www.json.org/

Nota: Iteradores

3.4 Google Gears

3.4.1 Introducción

Google Gears es un software de código abierto distribuido por Google que añade una nueva capa de aplicación a los navegadores.

Una vez instaldo como una extensión en el navegador, el producto agrega una API que permite programar en JavaScript interacciones con los componentes que contiene.

Los tres componentes principales que incorpora gears son:

- Local Server

  Permite almacenar localmente datos correspondientes a las páginas webs. Tanto HTML, JavaScript e imágenes entre otros, pueden ser almacenados localmente por el cliente e interponerse entre el requerimiento del navegador al servidor en conslutas posteriores, evitando asi la solicitud HTTP y optimizando el tiempo de respuesta de la aplicacion.

  Pese a que su funcionamiento es muy similar al de la cache del navegador la diferencia fundamental esta en que la actualizacion de los recursos que almacena

es mantenida y realizada por el desarrollador.

- Database

  Permite almacenar localmente datos que no correspondan a una página web pero son parte de la logica de la aplicacion y requieren de un almacenamiento presistente.

  El motor de base de datos utilizado es SQLite con algunos agregados y restricciones para brindar seguridad y formas de busqueda.

  Luego de que el usuario de la aplicacion web otroge el permiso explicito de creacion de la base, el desarrollador, disponer de un almacenamiento del tipo relacional en la máquina huésped.

- Worker Pool

  De manera similar a los "hilos" del sistema operativo, el manejador de hilos permite ejecutar acciones en segundo plano sin bloquear la ejecución del hilo principal del navegador.

  Hay que destacar que el manejador no corre en forma paralela a la ejecución del navegador, sino que se ejecuta cuando la página web se mantiene activa, por lo cual el refresco de página o la salida de la misma provoca que este se detenga o no se ejecute directemente.

Basicamente Gears y sus principales componentes estan enfocados en permitir al programador ejecutar sus aplicaciones cuando el navegador no está conectado al servidor. El lider del grupo de desarrollo Bret Taylor dijo que buscaba ser capaz de acceder al Google Reader mientras usaba la conexión de la compañía, la cual frecuentemente tenia un acceso defectuoso a Internet.

Gears esta incluido en el nuevo navegador de Google (Google Chrome) y posee extensiones para instalarse en Internet Explorer 6.0+, Mozilla Firefox, Safari y Opera Mini, y funciona en los sistemas operativos Windows 2000, XP y Vista, Windows Mobile 5 y 6, MacOS y Linux de 32 bits.

Con susecivos lanzamientos el producto sea visto mejorado y favorecido en varios aspectos a partir de la versión 0.4 del Gears se puede hablar ya de:

- API para GIS, que permite acceder a la posición geográfica del usuario.
- El API Blob, que permite gestionar bloques de datos binarios.
- Accede a archivos en el equipo cliente a través del API de Google Desktop.
- Permite enviar y recibir Blobs con el API HttpRequest.
- Localización de los cuadros de diálogo de Gears en varios idiomas.
- API para canvas, que permite manipular imagenes desde JavaScript.

En todos los casos que se requiera tener instalada una aplicacion soportada por Gears vale la pena aclarar que el cliente debe haber accedido al menos una vez al servidor de la aplicacion web y

otorgado los permisos de instalacion correspondiente.

CAPÍTULO 4

Introducción al desarrollo

Nota: Un mejor titulo no vendría mal

4.1 Soporte de lenguajes de programación en el browser

Tras la elección de Django como framework web, se emprendió el análisis de las posibilidades de ejecución de una aplicación web desarrollada en Django en el browser. Surgen puntos de interés como:

- ¿Es posible ejecutar el esquema de servidor web en un cliente?
- ¿Es posible ejecutar Python y mod_wsgi en el cliente?
- ¿Que posibilidades hay de brindar seguridad sobre los datos que se transfieran al cliente?
- Una vez lograda la transferencia de una aplicación web al cliente, ¿Que posibilidad existe de sincronizar los datos con la aplicación "madre"?

entre otras que se analizaran más adelante en el texto.

El enfoque inicial fue realizar un "espejo" de la aplicación que se ejecuta en el servidor en el cliente.

Django está escrito en Python y si bien la ejecución de Python es posible en un browser, la solución es engorrosa y no es cross-browser. En la plataforma Mozilla, la integración se puede realizar mediante PyXPCOM [1] , PyShell [2] y también existe una extensión para XUL [3]

[1] *PyXPCOM*, conexión del modelo de objetos multiplataforma de Mozilla con Python, https://developer.mozilla.org/en/PyXPCOM

[2] *PyShell*, consola interactiva

[3] *Luxor, Python for XUL* http://mail.python.org/pipermail/python-announce-list/2003-March/002084.html

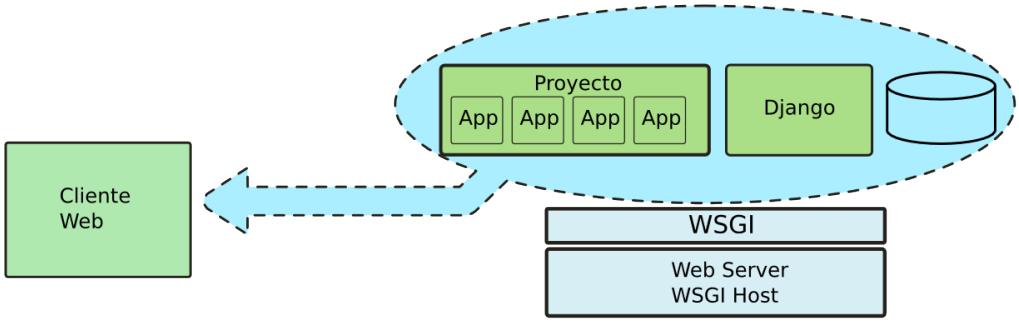


Figura 4.1: Elementos a transferir desde la aplicación online

4.2 Soluciones existentes

Sobre la plataforma Windows, existen dos formas de ejecutar Python en el navegador, la primera consiste en la ejecución de un control ActiveX sobre el browser que cuente con el intérprete de Python embebido. Un control ActiveX es un componente ejecutable empotrable, que puede ser dibujado en una página web. Los controles ActiveX son peligrosos en el ámbito de la web debido a que fueron ideados para ser utilizados como elemento incrustable entre aplicaciones o para el uso en entornos confiables. Un control ActiveX cuenta con privilegios similares a los de una aplicación tradicional sobre el equipo del cliente. La mayoría de los antivirus y herramientas de seguridad los eliminan o hacen responsable de la seguridad al usuario a partir de la ejecución del ActiveX. Si bien esta técnica se presenta atractiva gracias a que Python es un lenguaje que ha sido diseñado para ser embebido, los controles ActiveX no cumplen con las garantías de seguridad necesarias para el desarrollo de aplicaciones para la web. Es posible considerar esta solución "cross-browser" gracias a proyectos como un *host para ActiveX sobre la plataforma mozilla* [4] pero no es multiplataforma.

La segunda alternativa es utilizar el framework Silverlight de Microsoft, que permite generar aplicaciones para browsers basados en la plataforma *.Net*. Silverlight es un runtime similar al popular Adobe Flash Player, pero las aplicaciones pueden ser creadas en cualquier lenguaje de la plataforma .Net, incluyendo Python y Ruby [5] [6] a partir de la versión 4, bajo sus implementaciones sobre CLR.

Nota: Preguntar en el IRC de *pyar* desde cuando tiene la API standard IronPython si no volar el párrafo

Esta tecnología exige que el navegador cuente con un plugin para la plataforma, siendo el más completo el desarrollado por Microsoft, propulsor de la plataforma .Net. IronPython es una implementación de Python sobre .Net que no cuenta con la API standard [7] de la versión CPython, por lo

[4] *ActiveX para Mozilla* http://www.iol.ie/~locka/mozilla/plugin.htm

[5] *Silverlight, a new way for Python?* http://mail.python.org/pipermail/python-list/2007-May/610021.html

[6] *Sitio de Michael Frood, donde explica como ejecutar IronPython sobre .Net* http://www.voidspace.org.uk/ironpython/silverlight/index.shtml#id2

[7] *Listado de Módulos de la API standard* http://docs.python.org/modindex.html

que no se puede ejecutar django de manera nativa.

Aunque en un principio no era posible ejecutar Django sobre la IronPython, gracias a la popularidad del framework, con la versión 2.0 se superó esta dificultad [8].

En

Gracias a la posibilidad de acceso a DOM por medio de una aplicación construida con Silverlight [9] y al almacenamiento local en el cliente introducido en en Silverlight 2.0, se hace posible ejecutar Django en el cliente con acceso a una base de datos local, sin embargo, la arquitectura de software necesaria para desplegar este tipo de aplicaciones se hace compleja, que en cierta forma apunta contra los ideales de Python y Django:

- **Plugin necesario** Es necesario un plugin en el browser que no se encuentra disponible para todas las plataformas *no cross-browser*. O al menos, no en su estado más maduro
- **Madurez de la herramientas fuera de la plataforma Windows** Las herramientas de desarrollo solo están en su estado más maduro sobre la plataforma Windows Si bien existen compiladores gratuitos, las herramientas son propietarias y las IDEs que permiten un desarrollo más eficiente son pagas y propietarias.
- No existe soporte para IronPython en la IDE de facto, VisualStudio.
- La implementación de Python no es la estándar, y por ahora poco soportada [IronPython-FAQ2009] .

Tras el análisis de Silverlight como tecnología de soporte, se decide analizar las posibilidades nativas de los navegadores web. En particular se focalizó el análisis sobre las implementaciones de Javascript como lenguaje de soporte para la programación del lado del cliente.

4.3 Un lenguaje adecuado para ejecución de la aplicación en el cliente

En principio, Javascript y Python parecen lenguajes bastante diferentes en su sintaxis, sin embargo, comparten ciertas características como ser orientados a objetos basados en prototipos y permitir la definición de clausuras [AtulVarma2009] .

Más allá de la ejecución de Python, el servidor necesita un medio para almacenar los datos, típicamente una base de datos y un mecanismo para servir estáticamente los archivos de medios. Transportar la aplicación a un cliente web requiere un mecanismo de almacenamiento que brinde estos dos elementos en conjunto.

Google desarrolló un plugin multiplataforma y cross-browser llamado *Google Gears* que tiene como objetivo facilitar el sporte offline de aplicaciones web. Gears se compone de 3 componentes

[8] *Django On IronPython* http://www.infoq.com/news/2008/03/django-and-ironpython

[9] *Silverlight Tutorial - Interaction With The DOM* http://www.switchonthecode.com/tutorials/silverlight-tutorial-interaction-with-the-dom

básicos:

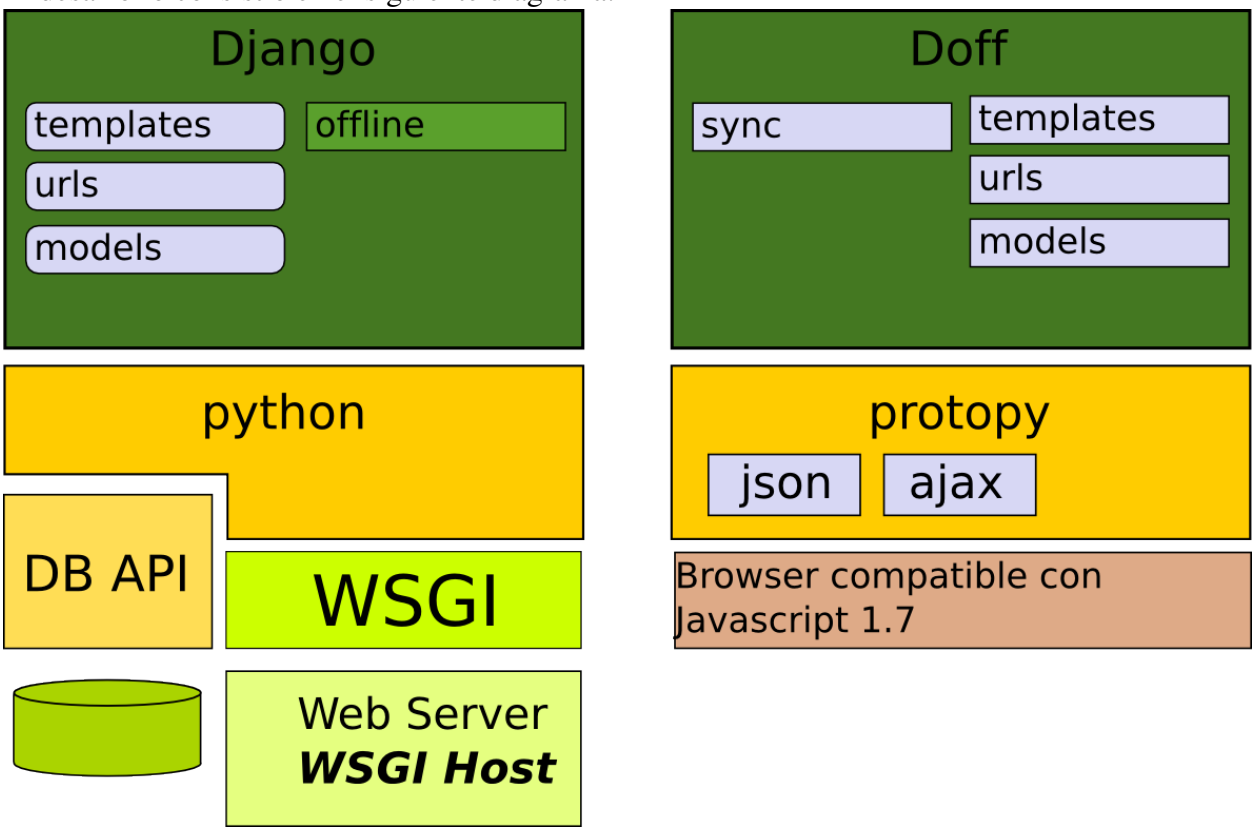
- **LocalServer** Un servidor de archivos estáticos en el clinete
- **DataBase** Una base de datos relacional
- **WorkerPool** Un sistema de procesos, para realizar tareas concurrentes de manera aislada y no provocar retrasos en el dibujado de la pantalla.

Generar este espejo conlleva poseer un equivalente al framework Django que pueda ser ejecutado en un browser, brindando los componentes básicos de Django:

- Mapeador Objeto Relacional
- Renderización de templates
- Asociación de expresiones regulares a funciones.

Esta idea surge en parte gracias al proyecto "Gars On Rails" [10] , un equivalente al objetivo de la presente tesis sobre el framework Rails.

El desarrollo consistió en el siguiente diagrama:



[10] *Isolated Storage in Silverlight 2.0* http://www.ddj.com/windows/208300036

CAPÍTULO 5

Protopy

5.1 Introducción

Nota: Ver donde poner estos dos parrafos, que estan buenos pero creo que no van aca :)

En apartados teóricos se menciono, entre otras cosas, que la creación de un framework generalmente surge de la identificación de objetos reusables en el desarrollo de software. Posteriormente los objetos identificadas decantan en componentes que forman parte de la arquitectura del framework y a los cuales se accede mediante la extención y la configuración. Esta forma de obtener un producto que permita arquitecturar proyectos de caracteristicas similares a los que lé dieron origen, implica pasar por varias etapas de maduración en el desarrollo, pero requieren de un punto de partida primordial y es justamente un proyecto que oriente la identificacion de las partes reusables. Nuestro caso se complica en este punto, ya que la intención panteada desde el principio es la de "desarrollar un framework" y la realidad del desarrollo es que muchas de las partes surguieron en paralelo y de la mano de aplicaciones de prueba.

Si bien la intención primaria es la de brindar un prodcuto que permita ejecutar "aplicaciones desconectadas" en los navegadores, existen aspectos complejos en el ambiente de un navegador que dificultan el desarrollo y que requieren de una capa soporte. Entre estas podemos mencionar:

- Organizar y obtener código.

  Un framework con funciones minimas, como una API de base de datos y un motor de plantillas, requiere de varias lineas de código para su implementación. Es por esto que resulta deseable que existan formas de organizar y obtener código en el cliente; liberando así al desarrollador de estas tareas tediosas y permitiendole enfocarse en la funcionalidad.

- Reuso y extensión.

  No es una buena practica modificar un framework y su utilizacion debe estar basa en la extención y la configuración. JavaScript es el lenguaje para el framework y

si se quiere promover el reuso y la extensión es conveniente proveer al lenguaje de objetos extensibles o contrucciones que fasilmente sean asimiladas por el desarrollador.

- Guerra de los navegadores.

  Si bien se han logrado muchos avanzes con la estandarizacion de APIs, en general existen muchas diferencias entre navegadores para trabajar con el DOM o con los objetos de JavaScript. Una capa uniforme de acceso al DOM, funciones y objetos que trabajen sobre APIs heterogeneas, entre otras cosas, resultan buenas herramientas para el desarrollo client-side.

- Interacción entre cliente servidor.
- 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. * 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 biblioteca en JavaScript, que brinde el soporte necesario al framework y a buena parte de los items expuestos.

5.1.1 Biblioteca

Protopy es una biblioteca 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*

5.1.2 Historia

Si bien el desarrollo de la biblioteca 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 Biblioteca en 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 codigo [1] el desarrollo del framework torna el enfoque inicial poco sustentable, requiriendo este de funciones más Python-compatibles se desecha la libreria base y se continua con un enfoque "pythonico", persiguiendo de esta forma acercar la semántica de JavaScript 1.7 a la del lenguaje de programacion Python.

[1] Forma en que los informaticos miden el paso del tiempo.

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 acercando, 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.

5.2 Organizando el codigo

Como ya se vio en la sección dedicada a *JavaScript*, una de las formas tradicionales y recomendada de incluir funcionalidad 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; en una instancia posterior, cuando el cliente accede al recurso, carga el archivo con las sentencias JavaScript y las interpreta en el contexto del documento.

El enfoque tradicional resulta sustentable para pequeños proyectos, donde el lenguaje brinda mayormente soporte a la interacción con el usuario (validacion, accesibilidad, etc) y los fragmentos de código que se pasan al cliente son bien conocidos por el desarrollador, pero 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 esto que para Protopy se busco como primera medida una forma de organizar y obtener el código del servidor que resulte sustentable y escalable.

Similar al concepto de *modulos en Python*, el desarrollo de Protopy se oriento en pequeñas unidades funcionales llamadas **modulos**.

Además de la sanidad mental que implica organizar el código en distintos archivos, los módulos representan un cambio muy importante en la obtención de funcionalidad; ya no es el documento HTML el que dice al cliente que archivo cargar del servidor, sino que el mismo código interpretado obtiene la funcionalidad a medida que la requiere.

> "Namespaces are one honking great idea – let's do more of those!"
>
> —import this

http://www.gulic.org/almacen/diveintopython-5.4-es/html_processing/locals_and_globals.html

El enfoque modular no es nuevo en programación y basicamente, la implementación de Protopy, implica llevar el concepto de "divide y vencerás" ó "análisis descendente (Top-Down)" al hambito de JavaScript.

Un módulo 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.

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 está disponible como el valor de la variable *__name__*.

5.2.1 Obtener un módulo

La función *require* es la encargada de obtener un módulo del servidor e incorporarlo al *espacio de nombres* del llamador. Por ejemplo, cuando un módulo llamado spam es requerido, Protopy busca un archivo llamado spam.js en la url base [2], de no encontrar el archivo el error `LoadError` es lanzado a la función que requirio el módulo.

Otra forma de obtener módulos es usando nombres de **paquetes**. A diferencia de Python un paquete no incluye funcionalidad en si mismo y su funcionalidad principal es la de establecer las bases en la cual buscar modulos. De forma similar a la anterior cuando un módulo llamado foo.spam es importado, Protopy busca en el objeto sys.paths si existe una url asociada al paquete foo, de encontrar la url base para foo el archivo spam.js es buscado en esa ubicación, por otra parte si sys.paths no contiene una url asociada a foo el archivo foo/spam.js es buscado en la url base.

El uso del objeto sys.paths permite a los modulos de JavaScript que saben lo que están haciendo modificar o reemplazar el camino de búsqueda para los módulos. Nótese que es importante que el script no tenga el mismo nombre que un *módulo estandar*.

Las formas en que el modulo obtenido es presentado al llamador difiere en funcion de los parametros pasados a *require*. Estas formas son:

- Un modulo puede ser obtenido como un objeto,
- Se puede obtener solo determinada funcion de un modulo,
- O se pueden importar todas las definiciones del módulo en el espacio de

nombres del llamador.

Para ver más *Apendice Protopy*.

5.2.2 Publicar un módulo

La acción de publicar un modulo implica exponer la funcionalidad que este define. En Python no es necesario explicitar que funcionalidad del módulo se expone a los llamadores, ya que todo lo definido en él es público; pero los módulos en Protopy se evaluan dentro de una clausura y los llamadores no podran acceder a sus funciones si no son publicadas.

La funcion *publish* es la encargada de relizar la tarea de publicar el contenido del modulo en Protopy.

Un modulo puede contener sentencias ejecutables y definición de funciones, generalmente las sentencias son para inicializar el módulo ya que estas se evaluan solo la primera vez que el módulo es requerido a alguna parte y las definiciones son las que efectivamente seran publicadas como funcionalidad.

[2] Ruta base desde la cual la biblioteca Protopy carga los módulos, por defecto esta es la url base del archivo protopy.js más el sufijo *packages*.

A continuación se presenta un fragmento de código que ejemplifica el uso de las dos funciones presentadas.

```
 /* Obtengo la funcion copy y deepcopy del modulo copy
    estas funciones son para copia de objetos
    superficiales y en profundidad respectivamente. */
require('copy', 'copy', 'deepcopy');

/* Representa la cantidad maxima de caches */
var MAX = 1000;

/* Objeto para guardar las cache */
var cache = {};

...

/* Estructura de datos que representa a una cache */
var Cache = type('Cache', [ Dict ], {
    ...
});

/* Funcion que retorna una cache */
function get_cache(name) {
    if (len(cache) > 1000)
        throw new Exception('%s caches creadas'.subs(MAX));
    var c = cache[name];
    if (!isundefined(c)) {
        c = new Cache();
        cache[name] = c;
    }
    return d;
}

/* Publico la cantidad maxima de caches
   la funcion para obtener caches. */
publish({
    MAX: MAX,
    get_cache: get_cache
});
```

Asumiendo que el código presentado está en un archivo llamado *caches.js* en la url base, éste representa un módulo en sí mismo y el acceso se obtiene mediante la invocación de 'require('caches')'.

Nota: Poner algo de espacio de nombres antes de entrar de lleno con escribir modulos y leer modulos

5.2.3 Modulos nativos

Nota: describir de forma rápida los módulos principales

- builtin
- dom
- sys
- event
- ajax
- exceptions
- timer

5.3 Creando tipos de objeto

En el apartado teórico se toco el tema de construcción de objetos en base a "clases" y de la emulación de herencia en JavaScript. Aunque muchos autores cuestionan estas practicas alegando, con justa razon, que no tiene sentido emular un paradigma dentro de otro; en la practica tener una implementación de objetos tipados en JavaScript ayuda al reuso de código y en gran medida a que los programadores se acerquen al lenguaje.

La forma de crear nuevos "tipos de objetos" en Protopy es a traves de la función *type*. Esta funcion no fue parte de la biblioteca hasta que no se observo la necesidad de otrogar mayor "poder" al constructor de clases que brindaba Prototype, y su aparción posibilito nuevas formas de construir tipos, similares a las construcciones de tipos que brinda la funcion homónima en Python, a la cual debe su nombre.

A continuación se presenta un fragmento de código que ejemplifica la creación de tipos en Protopy.

```
var Dict = type('Dict', object, {
    ...
});

var SortedDict = type('SortedDict', [ Dict ], {
    __init__: function(object) {
        this.keyOrder = (object && isinstance(object, SortedDict))? copy(object.key
        super(Dict, this).__init__(object);
    },
    __iter__: function() {
        for each (var key in this.keyOrder) {
            var value = this.get(key);
            var pair = [key, value];
            pair.key = key;
```

```
15              pair.value = value;
16              yield pair;
17          }
18      },
19      __deepcopy__: function() {
20          var obj = new SortedDict();
21          for (var hash in this._key) {
22              obj._key[hash] = deepcopy(this._key[hash]);
23              obj._value[hash] = deepcopy(this._value[hash]);
24          }
25          obj.keyOrder = deepcopy(this.keyOrder);
26          return obj;
27      },
28      __str__: function() {
29          var n = len(this.keyOrder);
30          return "%s".times(n, ', ').subs(this.keyOrder);
31      },
32      set: function(key, value) {
33          this.keyOrder.push(key);
34          return super(Dict, this).set(key, value);
35      },
36      unset: function(key) {
37          without(this.keyOrder, key);
38          return super(Dict, this).unset(key);
39      }
40  });
```

Rapidamente, este ejemplo presenta la definición del tipo SortedDict, el cual es una especialización del tipo base Dict. Como se observa la función constructora recibe, como primer argumento el nombre para el nuevo tipo, seguidamente un arreglo con los tipo base y para terminar un objeto con los atributos y metodos para los objetos de ese tipo.

Con los constructores así definidos es posible crear "objetos tipados" que se comporten en función de sus respectivas definiciones.

```
>>> d = new SortedDict({'uno': 1})
>>> d.set('dos', 2)
>>> d.get('dos')
2
>>> d.items()
[["uno", 1 ], ["dos", 2 ]]
```

Como se observa, para instanciar un nuevo tipo se utiliza el operador *new* de JavaScript, este operador crea el nuevo objeto e invoca a la función *__init__*.

En los métodos la palabra reservada *this* tiene el comportamiento esperado, presentado en la parte teorica, el cual es hacer referencia al objeto instanciado con *new*.

Internamente *type* utiliza el objeto *prototype* para la construcción, con lo cual el operador *instanceof* presenta un comportamiento coherente, pese a esto se recomienda usar la funcion *isinstance* que trae Protopy, ya que esta permite navegar por la cadena de herencia.

```
>>> d instanceof SortedDict
true
>>> d instanceof Dict
false
>>> isinstance(d, Dict)
true
```

5.3.1 Inicialización

En el ejemplo se muestra la inicialización del tipo SorteDict usando una funcion llamada *__init__*. Este método es llamado por el operador *new* inmediatamente despues de crear una instancia. Sería tentador decir que es el "constructor" de la clase. Si bien se parece un constructor de Java, actúa como si lo fuese ya que el objeto ya esta instanciado cuando se llama a esta función.

Los métodos *__init__* son opcionales, pero cuando se define uno, se debe recordar llamar explícitamente al método *__init__* del ancestro.

5.3.2 Otros métodos

Además de los métodos normales, Protopy prevee algunos métodos especiales para los objetos, estos en lugar de ser llamarlos directamente, se invocan por la biblioteca en circunstancias particulares o cuando se use una sintaxis específica.

- *__str__*

  Este metodo se llama cuando es necesario proveer de una reprecentacion en texto del objeto, JavaScript provee para este objetivo el metodo toString, pero por cuestiones de nombres se prefirio usar este metodo en su lugar y hacer internamente una relacion entre las funciones.

- *__iter__*

  Protopy se vale de versiones modernas de JavaScript y brinda soporte a iteradores, este es el método que el desarrollador debe definir si quiere objetos iterable, la funcion debe retornar un objeto que implemente el metodo next (un iterador o un generado), los bucles for hacen esto automáticamente, pero también se puede hacer manualmente.

- *__cmp__*

  Llamado al comparar dos instancias de tipo con las funciones *equal* o *nequal*.

- *__len__*

Se invoca con la llamada a la funcion *len*. La función incorporada len devuelve la longitud de un objeto. Funciona sobre cualquier objeto del que se pueda pensar razonablemente que tiene longitud.

- *__copy__* y *__deepcopy__*
- *__json__* y *__html__*

Protopy tiene otros métodos especiales, general todos estos apuntan a emular algun comportambiento de Python en JavaScript.

5.3.3 Herencia

Como ya se menciono, el codigo de un Framework no debe ser modificado y su utilizacion esta respaldada en algun mecanimos de extención, redefiniendo o especializando componentes. Es en este punto donde proveer de una forma de herencia al constructor de tipos resulta un paso en la dirección correcta, eventualmente el desarrollador que requiera componentes especificos podra heredar de los tipos correspondientes e implementar solo lo que haga falta.

Protopy utiliza la herencia del tipo **Prototype chaining**, aunque lo hace de una forma un poco "rebuscada" con el objeto de dar soporte a una herencia multiple.

Para implementar la herencia y en particular la herencia multiple, el contructor de tipos recibe una lista de los tipos base e internamente crea un objeto que agrega de derecha a izquierda todos los metodos de las bases, posteriormente crea el tipo requerido tomando como base el objeto generado. Es en este punto donde se pierde el poder del operador *instanceof* y es por eso que Protopy provee una funcion para determinar la correspondencia entre instancias y tipos llamada *isinstance*.

Otra funcion interesante importada de Python es *issubclass*, bajo determinadas condiciones resulta util determinar si un tipo es una sub-clase de otro y para ello esta funcion inspecciona la cadena de herencia hacia adelante entre los "hijos" de un tipo.

Bien, ya se tiene un mecanismo de herencia casi completo, solo falta ver como se accede a las funciones de tipos base cuando se esta redefiniendo un método, para esto existe la funcion *super*, nuevamente y similar a Python esta función asocia un tipo con una instancia, con lo cual logra el objetivo de llamar a un método que se encuentre en un tipo base. Este comportamiento, es el equivalente, en JavaScript, al de llamar al metodo del tipo base con la funcion *apply* o *call*.

Para terminar con herencia se debe destacar que de no especificar por lo menos un tipo base, Protopy establce por defecto como base al tipo *object*. Object como provee de los principales metodos (*__init__*, *__str__*) a todos los nuevos tipos.

5.3.4 Metodos y atributos de tipo

Hasta aqui se vio como crear nuevos tipos de objetos y la posterior construccion de instancias de un determinado tipo, Protopy tambien prevee una forma de definir metodos y atributos de tipo, estos pertenecen al tipo en si, y no a las instancias.

La forma de agregar atributos y metodos de tipo es a traves de un objeto opcional que se pasa a la función constructora antes del objeto de instancia.

```
var Planeta = new type('Planeta', [ object ], {
    // Atributos y metodos de tipo
    count: 0,
    reset: function() {
        this.count = 0;
    }
}, {
    // Atributos y metodos de instancia
    __init__: function(name) {
        this.name = name;
        this.count = Planeta.count++;
        // Otra forma puede ser con this.__class__.count++
    }
});
```

El ejemplo presentado define el tipo "Planeta", este tipo internamente lleva un contador que las instancias utilizan para numerarse en la construccion y una funcion de reset para reiniciar el contador. Dentro de las funciones de tipo la palabra reservada *this*, como es de esperar, hace referencia al tipo. A continuacion se ve como usar el tipo.

```
>>> p = new Planeta('Tierra')
window.Planeta name=Tierra count=0 __name__=Planeta
>>> Planeta.reset()
>>> sol = new Planeta('Sol')
window.Planeta name=Sol count=0 __name__=Planeta __module__=window
>>> mercurio = new Planeta('Mercurio')
window.Planeta name=Mercurio count=1 __name__=Planeta
>>> venus = new Planeta('Venus')
window.Planeta name=Venus count=2 __name__=Planeta
>>> tierra = new Planeta('Tierra')
window.Planeta name=Tierra count=3 __name__=Planeta
```

Existe una funcion muy importante que se puede definir para el tipo, buscando la emulacion de Python se adopto el metodo *__new__*. Este metodo permite al desarrollador tomar parte en la constuccion del tipo, los parametros son de una forma similar a Python, el nombre del nuevo tipo, el arreglo con los tipos base y el objeto con atributos y metodos de instancia.

5.3.5 Objetos nativos

Algunos objetos que se encuentran el la biblioteca son:

- Dict

Si bien la estructura "hasheable" nativa de JavaScript en un objeto, los diccionarios de Protopy permiten mejores formas de trabajar con la dupla clave-valor, posibilitando ademas el uso de objetos como claves en lugar de solo cadenas.

- Set

  Los sets o conjuntos tomados de las matematicas, permiten crear bolasa de objetos únicos y trabajar con las operaciones propias, union, intersección, diferencia, etc.

- Arguments

  Las funciones en JavaScript pueden recibir opcionalmente cualquier cantidad de argumentos, los objetos del tipo Arguments encapsulan y uniforman los argumentos pasados a una funcion y permiten establecer entre otras cosas valores por defecto.

5.4 Extendiendo DOM y JavaScript

Si bien el *DOM* ofrece ya una *API* muy completa para acceder, añadir y cambiar dinámicamente el contenido del documento HTML, existen funciones muy utiles y comunes en los desarrollos que los programadores incorporan al HTML y se intentaron englobar y mantener dentro del nucleo de Protopy. Funciones para modificar e incorporar elementos al documento son muy comunes y estan disponibles en Protopy, en conjunto con otras para el manejo de formularios, como serializacion, obtencion de valores, etc.

Otra extension interesante de mencionar es la de los tipos de datos en JavaScript, en manejo de cadenas incorpora nuevas funciones que simplifican tareas comunes, los numeros y fechas tambien tiene su aporte.

Los eventos estan uniforamdos bajo un modulo de manejo de eventos, que permite conectar eventos del *DOM* con funciones en JavaScript, así tambien como funciones entre si.

Nota: Poner los nombres de las funciones para destacar el laburo

5.5 Envolviendo a gears

La biblioteca puede funcionar independiente de la instalación de Google Gears, debido a que su principal funcionalidad como ya se menciono es la de extención de JavaScript y posterior soporte al framework, pese a esto Protopy provee mecanismos para uniformar el acceso y extender los objetos de Gears cuando éste se encuentra instalado en el navegador.

El acceso al *Factory* de Gears esta centralizado y controlado en el objeto *gears* dentro del modulo *sys*, mediante este objeto es posible conocer el estado de Gears y sus permisos. El objeto informa al desarrollador si Gears esta instalado, la versión, si los permisos son corrector e incluso simplifica el proceso de instalación de la extensión de no estar presente en el navegador entre otras cosas.

El metodo *create* del objeto *sys.gears* es el encargado de crear y retornar los objetos Gears. Esta función se ayuda de módulos presentes en el paquete *gears* para asistir la creación de los objetos. El objeto que se retorne dependera de la precencia del módulo con el mismo nombre en dicho paquete; de no encontrar un modulo que asista la cración de un objeto Gears el objeto en sus estado "puro" es retornado al llamador.

Si bien no es necesario que los módulos obtengan el acceso a Gears a traves de Protopy, es recomendable que asi se haga; ya que la biblioteca provee los mecanimos de extención para los objetos en *create*. Esto no fue así desde el comienzo de del desarrollo y fue una idea que se maduro luego de observar que resultaba complejo y costoso requerir los módulos que involucraban a Gears desde distintos lugares.

El desarrollo del framework implico extender algunos objetos Gears, concretamente al paquete *gears* se incorporaron los siguientes módulos:

- desktop

  El objeto *desktop* de Gears permite interactuar con el escritorio del cliente. Aquí se extendio la creacción de accesos directos para simplificar la generación de los mismos y agregar la posibilidad de manejar *Icon* y algunos *IconTheme*.

- database

  Sobre el objeto *database* se agrego funcionalidad uniformar el acceso a la base de datos por los módulos de Protopy y encapsular los *ResulSet* en cursores a los que se incorporo iteradores, registro de funciones para tipos de datos, etc.

5.6 Auditando el codigo

Una queja recurrente de los desarrolladores que trabajan con JavaScript es lo complejo que resulta el lenguaje para depurar errores. Encontrar errores en el código resulta molesto y mas todavía si la salida de los mismos no esta en un formato adecuado y encausada en un lugar especifico.

Tradicionalmente lo que se hace para detectar errores es valerse de funciones *alert* diseminadas por el código con textos del estilo "Paso por aquí", pero luego de cerrar unas diez o quince ventanas de este estilo, generalmente se pierde referencia de donde esta ocurriendo el error y se cae en la tentación de comenzar a comentar alerts a mansalva esperando dar con el indicado. Este es un claro ejemplo de que tanto la salida como el sistema de detección de errores es molesto e infructifero.

Una clara ventaja sobre el sistema tradicional y poderosa herramienta de desarrollo es el plug-in Firebug, que integra entre otras cosas una consola JavaScript y un debuger al ambinete del navegador, permitiendo a los desarrolladores depurar el código mediante puntos de corte, inspección de variables, etc. En firebug la consola pasa a ser por defecto la principal salida de errores, gracias a la funcion *console.log* los desarrolladores pueden redirigir todos los "Paso por aquí" a una salida uniforme e incluso inspeccionar el valor de las variables.

Protopy lleva la depuración y auditoria del código un paso más lejos, integrando un sistema de

logging propio altamente configurable y con posibilidades de escribir en diferentes salidas y con diferentes formatos.

Similar a log4j el logger de Protopy trabaja con niveles de prioridad para los mensajes, distintos manejadores o *Handlers* y salidas en varios formatos. Todo esto configurable por el desarrollador.

Una vez configurado el sistema de logging, los módulos que requieran auditar el codigo solo deben requerir un logger en su sespacio de nombre e invocar a sus funciones.

```
var logging = require('logging.base');
var logger = logging.get_logger(__name__);

...

logger.debug('La query: %s\n Los parametros: %s', query, params, {});
```

En este ejemplo se requiere el módulo *logging* y posteriormente un logger para el módulo con el nombre __name__, de no encontrar configuración para este módulo se adopta la configuracion del modulo inmediato superior y asi consecutivamente hasta tomar la configuración del root logger.

Suponiendo que el módulo del ejemplo se llama doff.db.models.sql el siguiente archivo de configuracion perapraria este logger en modo DEBUG para auditar el código en la consola de firebug y en una url con distintos formatos.

```
{
    'loggers': {
        'root': {
            'level': 'DEBUG',
            'handlers': 'firebug'
        },
        'doff.db.models.sql': {
            'level': 'DEBUG',
            'handlers': [ 'firebug', 'remote'],
            'propagate': true
        },
    },
    'handlers': {
        'firebug': {
            'class': 'FirebugHandler',
            'level': 'DEBUG',
            'formatter': ' %(time)s %(name)s(%(levelname)s):\n %(message)s',
            'args': []
        },
        'remote': {
            'class': 'RemoteHandler',
            'level': 'DEBUG',
            'formatter': ' %(levelname)s:\n %(message)s',
            'args': ['/loggers/audit']
```

```
        },
        'alert': {
            'class': 'AlertHandler',
            'level': 'DEBUG',
            'formatter': '%(levelname)s:\n%(message)s',
            'args': []
        }
    }
}
```

Nota: Poner referencia a firebug

5.7 Interactuando con el servidor

Protopy permite al desarrollador trabajar con AJAX de forma simple y segura, encapuslando en objetos la lógica de petición y los valores de retorno del servidor.

Todo lo relacionado con AJAX se encuentra en el modulo *ajax*. El objeto de transporte para ajax es *XMLHttpRequest* e internamente se salvan las diferencias que existen entre los distintos navegadores. La forma de realizar una peticion es creando una instancia del objeto ajax.Request.

```
new ajax.Request('una/url', {method: 'get'});
```

La primera opcion es la url de la solicitud, y el segundo parametro es el hash de opciones, en este caso el metodo a utilizar es GET, si no se especifica, el metodo por defecto es POST.

Por defecto la respuesta del servidor es asincrona, para este caso se debe explicitar en el hash de opciones las funciones que manejaran los eventos disparados.

```
new ajax.Request('una/url', {
    method:'get',
    onSuccess: function(transport){
        var response = transport.responseText || "sin texto";
        alert("Success! \n\n" + response); },
    onFailure: function(){
        alert('Algo esta mal...'); }
});
```

En el ejemplo se presentan dos funciones, onSuccess y onFailure, para manejar los eventos correspondientes. A cada manejador se le pasa un objeto que representa la respuesta obtenida y que esta en relación con el evento capturando.

Otros manejadores que se pueden definir son:

- onUninitialized

- onLoading
- onLoaded
- onInteractive
- onComplete
- onException

Todos estos dependen de un estado del objeto *XMLHttpRequest*.

De igual manera que el resto de las opciones es posible agregar parametros a la peticion, estos pueden ser pasados como un objeto "hasheable" o como una cadena clave-valor.

5.8 Soporte para json

La idea detras del soporte para JSON en Protopy es la transimisión de datos generados offline por el cliente, en el momento de recuperar la conexion con el servidor el cliente debe serializar los datos y enviarlos al servidor; otro uso para es el intercambio de mensajes de control.

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 JSON como el formato de datos a utilizar, es por esto que Protopy incluye un modulo para trabajar con el mismo.

El soporte para JSON se encuentra en el modulo "json" entre los módulos 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 __json__, dejando de este modo en manos del desarrollador la reprecentacion en JSON de determinado objetos.

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.

5.8.1 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 JSON contra 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.

5.9 Ejecutando código remoto

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

CAPÍTULO 6

Doff

6.1 Introducción

————————- OJO CON ESTO QUE TIENE PONIES ————————

Este libro es sobre Django, un framework de desarrollo Web que ahorra tiempo y hace que el desarrollo Web sea divertido. Utilizando Django puedes crear y mantener aplicaciones Web de alta calidad con un mínimo esfuerzo.

En el mejor de los casos, el desarrollo web es un acto entretenido y creativo; en el peor, puede ser una molestia repetitiva y frustrante. Django te permite enfocarte en la parte divertida – el quid de tus aplicaciones Web – al mismo tiempo que mitiga el esfuerzo de las partes repetitivas. De esta forma, provee un alto nivel de abstracción de patrones comunes en el desarrollo Web, atajos para tareas frecuentes de programación y convenciones claras sobre cómo solucionar problemas. Al mismo tiempo, Django intenta no entrometerse, dejándote trabajar fuera del ámbito del framework según sea necesario.

El objetivo de este libro es convertirte en un experto de Django. El enfoque es doble. Primero, explicamos en profundidad lo que hace Django, y cómo crear aplicaciones Web con él. Segundo, discutiremos conceptos de alto nivel cuando se considere apropiado, contestando la pregunta "¿Cómo puedo aplicar estas herramientas de forma efectiva en mis propios proyectos?" Al leer este libro, aprenderás las habilidades necesarias para desarrollar sitios Web poderosos de forma rápida, con código limpio y de fácil mantenimiento.

En este capítulo ofrecemos una visión general de Django.

6.1.1 Framework

Doff en un Framework Web escrito en JavaScript que provee un marco de desarrollo para las aplicaciones Web cliente-side.

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

6.1.2 Historia

> **d** jango + **off** line = **doff**

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.

El framework fue construido con el objeto de ser compatible con los modelos y las plantillas de django, intentando acercar así al desarrollador que tenga experiencia en este framework.

Django nació naturalmente de aplicaciones de la vida real escritas por un equipo de desarrolladores Web en Lawrence, Kansas. Nació en el otoño boreal de 2003, cuando los programadores Web del diario *Lawrence Journal-World*, Adrian Holovaty y Simon Willison, comenzaron a usar Python para crear sus aplicaciones. El equipo de The World Online, responsable de la producción y mantenimiento de varios sitios locales de noticias, prosperaban en un entorno de desarrollo dictado por las fechas límite del periodismo. Para los sitios – incluidos LJWorld.com, Lawrence.com y KUsports.com – los periodistas (y los directivos) exigían que se agregaran nuevas características y que aplicaciones enteras se crearan a una velocidad vertiginosa, a menudo con sólo días u horas de preaviso. Es así que Adrian y Simon desarrollaron por necesidad un framework de desarrollo Web que les ahorrara tiempo – era la única forma en que podían crear aplicaciones mantenibles en tan poco tiempo – .

En el verano boreal de 2005, luego de haber desarrollado este framework hasta el punto en que estaba haciendo funcionar la mayoría de los sitios World Online, el equipo de World Online, que ahora incluía a Jacob Kaplan-Moss, decidió liberar el framework como software de código abierto. Lo liberaron en julio de 2005 y lo llamaron Django, por el guitarrista de jazz Django Reinhardt.

A pesar de que Django ahora es un proyecto de código abierto con colaboradores por todo el mundo, los desarrolladores originales de World Online todavía aportan una guía centralizada para el crecimiento del framework, y World Online colabora con otros aspectos importantes tales como tiempo de trabajo, materiales de marketing, y hosting/ancho de banda para el Web site del framework (http://www.djangoproject.com/).

Esta historia es relevante porque ayuda a explicar dos cuestiones clave. La primera es el "punto dulce" de Django. Debido a que Django nació en un entorno de noticias, ofrece varias características (en particular la interfaz admin, tratada en el **'Capítulo 6'_**) que son particularmente apropiadas para sitios de "contenido" – sitios como eBay, craigslist.org y washingtonpost.com que ofrecen información basada en bases de datos –. (De todas formas, no dejes que eso te quite las ganas – a pesar de que Django es particularmente bueno para desarrollar esa clase de sitios, eso no significa que no sea una herramienta efectiva para crear cualquier tipo de sitio Web dinámico –. Existe una diferencia entre ser *particularmente efectivo* para algo y *no ser efectivo* para otras cosas).

La segunda cuestión a resaltar es cómo los orígenes de Django le han dado forma a la cultura de su

comunidad de código abierto. Debido a que Django fue extraído de código de la vida real, en lugar de ser un ejercicio académico o un producto comercial, está especialmente enfocado en resolver problemas de desarrollo Web con los que los desarrolladores de Django se han encontrado – y con los que continúan encontrándose –. Como resultado de eso, Django es activamente mejorado casi diariamente. Los desarrolladores del framework tienen un alto grado de interés en asegurarse de que Django les ahorre tiempo a los desarrolladores, produzca aplicaciones que son fáciles de mantener y rindan bajo mucha carga. Aunque existan otras razones, los desarrolladores están motivados por sus propios deseos egoístas de ahorrarse tiempo a ellos mismos y disfrutar de sus trabajos. (Para decirlo sin vueltas, se comen su propia comida para perros).

6.2 Modelos

La forma de interactuar con la base de datos es a traves de los modelos, siendo consistente con Django, Doff abstrae la logica de negocios dentro de los modelos.

Continuando con la linea de ejemplos se presenta a continuación una configuración de datos básica sobre libro/autor/editor, esta estructura debe estar contenida en un archivo models.js dentro de una aplicacion:

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

var Publisher = type('Publisher', [ models.Model ], { name: new models.CharField({ maxlength: 30 }), address: new models.CharField({ maxlength: 50 }), city: new models.CharField({ maxlength: 60 }), state_province: new models.CharField({ maxlength: 30 }), country: new models.CharField({ maxlength: 50 }), website: new models.URLField()

});

var Author = type('Author', [ models.Model ], { salutation: new models.CharField({ maxlength: 10 }), first_name: new models.CharField({ maxlength: 30 }), last_name: new models.CharField({ maxlength: 40 }), email: new models.EmailField(), headshot: new models.ImageField({ upload_to: '/tmp' })

});

var Book = type('Book', [ models.Model ], { title: new models.CharField({ maxlength: 100 }), authors: new models.ManyToManyField(Author), publisher: new models.ForeignKey(Publisher), publication_date: new models.DateField()

});
```

Cada modelo es representado por un tipo de Protopy que es un subtipo de `doff.db.models.model.Model`. El tipo `Model` contiene toda la maquinaria necesaria para hacer que los nuevos tipos sean capaces de interactuar con la base de datos.

Cada modelo se corresponde con una tabla única de la base de datos, y cada atributo de un modelo con una columna en esa tabla. El nombre de atributo corresponde al nombre de columna, y el tipo de campo corresponde al tipo de columna de la base de datos. Por ejemplo, el modelo `Publisher` es equivalente a la siguiente tabla:

> **CREATE TABLE "books_publisher"** ("id" serial NOT NULL PRIMARY KEY, "name" varchar(30) NOT NULL, "address" varchar(50) NOT NULL, "city" varchar(60) NOT NULL, "state_province" varchar(30) NOT NULL, "country" varchar(50) NOT NULL, "website" varchar(200) NOT NULL
>
>);

La excepción a la regla una-clase-por-tabla es el caso de las relaciones muchos-a-muchos. En el ejemplo, `Book` tiene un `ManyToManyField` llamado `authors`. Esto significa que un libro tiene uno o más autores, pero la tabla de la base de datos `Book` no tiene una columna `authors`. En su lugar, se crea una tabla adicional que maneja la correlación entre libros y autores.

Para una lista completa de tipos de campo y opciones de sintaxis de modelos, ver el Apéndice B.

Finalmente, no se define explícitamente una clave primaria en ninguno de estos modelos. A no ser que se le indique lo contrario, Doff dará automáticamente a cada modelo un campo de clave primaria entera llamado `id`.

Para activar los modelos en el proyecto, la aplicación que los contine debe estar incluida en la lista de aplicaciones instaladas de Doff. Esto es, edita el archivo `settings.js`, y examina la variable de configuración `INSTALLED_APPS`

Posteriormente, cuando el usuario instala la aplicación en su navegador, el sistema recorre las aplicaciones en `INSTALLED_APPS` y genera el SQL para cada modelo, creando las tablas en la base de datos.

Doff provee entre las herramientas del desarrollador, un interprete de SQL sobre la base de datos del cliente para conslutas.

6.2.1 Acceso básico a datos

Una vez que se creo el modelo, Doff provee automáticamente una API JavaScript de alto nivel para trabajar con estos modelos:

```
>>> require('books.models', 'Publisher');
>>> p1 = new Publisher({ name: 'Addison-Wesley', address: '75 Arlington Street',
...     city: 'Boston', state_province: 'MA', country: 'U.S.A.',
...     website: 'http://www.apress.com/'});
>>> p1.save();
>>> publisher_list = Publisher.objects.all();
>>> print(array(publisher_list));
[<Publisher: Publisher object>]
```

Se puede hacer mucho con la API de base de datos de Doff y para mejorar la interactividad se recomienda implementar __str__ de Protopy. Con este método los objetos tendran su representación en "string", es imporante que eso sea así ya que el framework utiliza esta representacion en muchos lugares, como templates y salidas por consola.

6.2.2 Insertando y actualizando datos

En el ejemplo presentado anteriormente se ve cómo se hace para insertar una fila en la base de datos, primero se crea una instancia del modelo pasando argumentos nombrados y luego se llama al método `save()` del objeto:

En el caso de `Publisher` se usa una clave primaria autoincremental `id`, por lo tanto la llamada inicial a `save()` hace una cosa más: calcula el valor de la clave primaria para el registro y lo establece como el valor del atributo `id` de la instancia.

Las subsecuentes llamadas a `save()` guardarán el registro en su lugar, sin crear un nuevo registro (es decir, ejecutarán una sentencia SQL `UPDATE` en lugar de un `INSERT`).

6.2.3 Seleccionar objetos

La forma de seleccionar y tamizar los datos se consige a traves de los administradores de consultas, en el ejmplo la linea `Publisher.objects.all()` pide al administrador objects de Publisher que obtenga todos los registros, internamente esto genera una consulta SQL:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher;
```

Todos los modelos automáticamente obtienen un administrador `objects` que debe ser usado cada vez que se quiera consultar sobre una instancia del modelo. El metodo `all()` es un método del administrador `objects` que retorna todas las filas de la base de datos. Aunque este objeto se *parece* a una lista, es actualmente un *QuerySet* – un objeto que representa algún conjunto de filas de la base de datos. El Apéndice C describe QuerySets en detalle.

Cualquier búsqueda en base de datos va a seguir esta pauta general.

6.2.4 Filtrar datos

Aunque obtener todos los objetos es algo que ciertamente tiene su utilidad, la mayoría de las veces lo que vamos a necesitar es manejarnos sólo con un subconjunto de los datos. Para ello usaremos el método `filter()`:

```
>>> Publisher.objects.filter(name="Apress Publishing")
[<Publisher: Apress Publishing>]
```

`filter()` toma argumentos de palabra clave que son traducidos en las cláusulas SQL `WHERE` apropiadas. El ejemplo anterior sería traducido en algo como:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name = 'Apress Publishing';
```

Puedes pasarle a `filter()` múltiples argumentos para reducir las cosas aún más:

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress Publishing>]
```

Esos múltiples argumentos son traducidos a cláusulas SQL `AND`. Por lo tanto el ejemplo en el fragmento de código se traduce a lo siguiente:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A.' AND state_province = 'CA';
```

Notar que por omisión la búsqueda usa el operador SQL = para realizar búsquedas exactas. Existen también otros tipos de búsquedas:

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress Publishing>]
```

Notar el doble guión bajo entre `name` y `contains`. Del mismo modo que Python, Django usa el doble guión bajo para indicar que algo "mágico" está sucediendo – aquí la parte `__contains` es traducida por Django en una sentencia SQL `LIKE`:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE name LIKE '%press%';
```

Hay disponibles varios otos tipos de búsqueda, incluyendo `icontains` (`LIKE` no sensible a diferencias de mayúsculas/minúsculas), `startswith` y `endswith`, y `range` (consultas SQL `BETWEEN`). El Apéndice C describe en detalle todos esos tipos de búsqueda.

6.2.5 Obteniendo objetos individuales

En ocasiones desearás obtener un único objeto. Para esto existe el método `get()`:

```
>>> Publisher.objects.get(name="Apress Publishing")
<Publisher: Apress Publishing>
```

En lugar de una lista (o más bien, un QuerySet), este método retorna un objeto individual. Debido a eso, una consulta cuyo resultado sean múltiples objetos causará una excepción:

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
AssertionError: get() returned more than one Publisher -- it returned 2!
```

Una consulta que no retorne objeto alguno también causará una excepción:

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
DoesNotExist: Publisher matching query does not exist.
```

6.2.6 Ordenando datos

A medida que juegas con los ejemplos anteriores, podrías descubrir que los objetos so devueltos en lo que parece ser un orden aleatorio. No estás imaginándote cosas, hasta ahora no le hemos indicado a la base de datos cómo ordenar sus resultados, de manera que simplemente estamos recibiendo datos con algún orden arbitrario seleccionado por la base de datos.

Eso es, obviamente, un poco ***silly*** (tonto), no querríamos que una página Web que muestra una lista de editores estuviera ordenada aleatoriamente. Así que, en la práctica, probablemente querremos usar `order_by()` para reordenar nuestros datos en listas más útiles:

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>
```

Esto no se ve muy diferente del ejemplo de `all()` anterior, pero el SQL incluye ahora un ordenamiento específico:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name;
```

Podemos ordenar por cualquier campo que deseemos:

```
>>> Publisher.objects.order_by("address")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>

>>> Publisher.objects.order_by("state_province")
[<Publisher: Apress Publishing>, <Publisher: Addison-Wesley>, <Publisher: O'Reilly>
```

y por múltiples campos:

```
>>> Publisher.objects.order_by("state_provice", "address")
 [<Publisher: Apress Publishing>, <Publisher: O'Reilly>, <Publisher: Addison-Wesley
```

También podemos especificar un ordenamiento inverso antecediendo al nombre del campo un prefijo - (el símbolo menos):

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>
```

Aunque esta flexibilidad es útil, usar `order_by()` todo el tiempo puede ser demasiado repetitivo. La mayor parte del tiempo tendrás un campo particular por el que usualmente desearás ordenar. Es esos casos Django te permite anexar al modelo un ordenamiento por omisión para el mismo:

Este fragmento `ordering = ["name"]` le indica a Django que a menos que se proporcione un ordenamiento mediante `order_by()`, todos los editores deberán ser ordenados por su nombre.

6.2.7 Encadenando búsquedas

Has visto cómo puedes filtrar datos y has visto cómo ordenarlos. En ocasiones, por supuesto, vas a desear realizar ambas cosas. En esos casos simplemente "encadenas" las búsquedas entre sí:

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress Publishing>, <Publisher: Addison-Wesley>
```

Como podrías esperar, esto se traduce a una consulta SQL conteniendo tanto un `WHERE` como un `ORDER BY`:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

Puedes encadenar consultas en forma consecutiva tantas veces como desees. No existe un límite para esto.

6.2.8 Rebanando datos

Otra necesidad común es buscar sólo un número fijo de filas. Imagina que tienes miles de editores en tu base de datos, pero quieres mostrar sólo el primero. Puedes hacer eso usando la sintaxis estándar de Python para el rebanado de listas:

```
>>> Publisher.objects.all()[0]
<Publisher: Addison-Wesley>
```

Esto se traduce, someramente, a:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
ORDER BY name
LIMIT 1;
```

6.2.9 Eliminando objetos

Para eliminar objetos, simplemente se debe llamar al método `delete()` del objeto:

```
>>> p = Publisher.objects.get({ name: "Addison-Wesley" });
>>> p.delete();
>>> array(Publisher.objects.all());
[]
```

Se pueden borrar objetos al por mayor llamando a `delete()` en el resultado de una búsqueda:

```
>>> publishers = Publisher.objects.all();
>>> publishers.delete();
>>> array(Publisher.objects.all());
[]
```

Nota: Mixin en RemoteSite, los modelos se registran en RemoteSite o se pueden hacer a mano. Hacer la referencia correspondiente al anexo de modelos

6.3 Plantillas

Doff brinda soporte al sistema de plantillas de Django. La idea detras de esto es que las plantillas escritas para una apilcacion on-line en Django pueda ser utilizada en la aplicación off-line de una forma consistente.

Nota: Existen sin emabargo algunos factores a tener en cuenta a la hora de realizar plantillas para aplicaciones offline. Cuando una aplicación se ejecuta de manera desconectada, el browser solo realiza una carga de documento y la navegación se basa en la inserción y supresión de nodos sobre el elemento document. Esto implica que el estado de la aplicación

Retomando el ejemplo presentado en la seccion de plantillas en Django, continuamos desde aquí describiendo como es el trabajo con esta plantilla en Doff.

Rapidamente, la forma de obtener un producto de la plantilla es:

 # Crea un objeto `Template` brindando el código crudo de la plantilla como una cadena.

 # Llama al método `render()` del objeto `Template` con un conjunto de variables (o sea, el contexto). Este retorna una plantilla totalmente renderizada como una cadena de caracteres, con todas las variables y etiquetas de bloques evaluadas de acuerdo al contexto.

6.3.1 Creación de objetos Template

Doff provee su versión del objeto `Template` para crear plantillas, y esté puede ser importado del módulo `doff.template.base`, el argumento para la construccion del objeto es el texto en crudo de la plantilla.

```
>>> require('doff.template.base', 'Template');
>>> t = new Template("Mi nombre es {{ name }}.");
>>> print(t);
```

En este ejemplo *t* es un objeto template listo para ser renderizado. Si se obtubo el objeto es porque la plantilla esta correctamente analizada y no se econtraron errores en la misma, algunos errores por los que puede fallar la construccion son:

- Bloques de etiquetas inválidos
- Argumentos inválidos de una etiqueta válida
- Filtros inválidos
- Argumentos inválidos para filtros válidos
- Sintaxis de plantilla inválida
- Etiquetas de bloque sin cerrar (para etiquetas de bloque que requieran la etiqueta de cierre)

En todos los casos el sistema lanza una excepción `TemplateSyntaxError`.

6.3.2 Renderizar una plantilla

Una vez que se tiene el objeto `Template`, se esta en condiciones de obtener una salida procesada en un determinado *contexto*. Un contexto es simplemente un conjunto de variables y sus valores asociados. Una plantilla usa las variables para poblar la plantilla evaluando las etiquetas de bloque.

El contexto esta representado en el tipo `Context`, el cual se encuentra en el módulo `doff.template.base`. La construccion del objeto toma un argumento opcional: un hash mapeando nombres de variables con valores. La llamada al método `render()` del objeto `Template` con el contexto "rellena" la plantilla:

```
>>> requiere('doff.template.base', 'Context', 'Template');
>>> t = new Template("Mi nombre es {{ name }}.")
>>> c = new Context({"name": "Pedro"})
>>> t.render(c)
'My name is Pedro.'
```

El objeto `Template` puede ser renderizado con múltiples contextos, obteniendo así salidas diferentes para la misma plantilla. Por cuestiones de eficiencia es conveniente crear un objeto `Template` y luego llamar a `render()` sobre este muchas veces:

```
# Mal chico
for each (var name in ['John', 'Julie', 'Pat']) {
    var t = new Template('Hello, {{ name }}');
    print(t.render(new Context({'name': name})));
}

# Buen chico
t = new Template('Hello, {{ name }}');
for each (var name in ['John', 'Julie', 'Pat'])
    print(t.render(new Context({'name': name})));
```

Al igual que en Django el objeto contexto puede contener variables mas complejas y la forma de inspeccionar dentro de estas es con el operador .. Usando el punto se puede acceder a objetos, atributos, índices, o métodos de un objeto.

Cuando un sistema de plantillas encuentra un punto en una variable el orden de busqueda es el siguiente:

- Diccionario (por ej. `foo["bar"]`)
- Atributo (por ej. `foo.bar`)
- Llamada de método (por ej. `foo.bar()`)
- Índice de lista (por ej. `foo[bar]`)

El sistema utiliza el primer tipo de búsqueda que funcione. Es la lógica de cortocircuito.

Para terminar con este objeto, si una variable no existe en el contexto, el sistema de plantillas renderiza este como un string vacío, fallando silenciosamente. Es posible cambiar este comportamiento modificando el valor de la variable de configuración *TEMPLATE_STRING_IF_INVALID* en el módulo *settings*.

6.3.3 Cargador de plantillas

Se ve a continuación un ejemplo de una vista que retorna HTML generado por una plantilla:

```
require('doff.template.base', 'Template', 'Context');
require('doff.utils.http', 'HttpResponse');
require('doff.template.loader', 'get_template');

function current_datetime(request) {
    var t = get_template('mytemplate.html');
    html = t.render(new Context({'current_date': new Date()}))
    return new HttpResponse(html);
}
```

En esta vista se utiliza la API para cargar plantillas, a la cual se accede mediante la funcion *get_template*, antes de poder utilizar esta funcion es necesario indicarle al framework donde están las plantillas. El lugar para hacer esto es en el *archivo de configuración*.

Existen varios cargadores de plantillas que se pueden habilitar en el archivo de configuración, este se vera en profundidad en el Apéndice E, por ahora se vera el cargador realacionado con la variable de configuración `TEMPLATE_URL`. Esta variable le indica al mecanismo de carga de plantillas dónde buscar las plantillas. Por omisión, ésta es una cadena vacia. El valor para esta varible es la url del servidor en donde se sirven los templates que se renderizan localmente, por defecto si no se especifica la variable los archivos se buscan en la base del soporte off-line /templates/.

Nota: Terminar esto. con el tema de la base del soporte off-line Introducir el concepto al inicio de doff Hacer la referencia correspondiente al anexo de plantillas

6.4 Formularios

———————————— TIENE PONIES ————————————-

Autor invitado: Simon Willison

Si has estado siguiendo el capítulo anterior, ya deberías tener un sitio completamente funcional, aunque un poco simple. En este capítulo trataremos con la próxima pieza del juego: cómo construir vistas que obtienen entradas desde los usuarios.

Comenzaremos haciendo un simple formulario de búsqueda “a mano”, viendo cómo manejar los datos suministrados al navegador. Y a partir de ahí, pasaremos al uso del *framework* de formularios

que trae Django.

6.5 Búsquedas

En la Web todo se trata de búsquedas. Dos de los casos de éxito más grandes, Google y Yahoo, han construido sus empresas multimillonarias alrededor de las búsquedas. Casi todos los sitios observan un gran porcentaje de tráfico viniendo desde y hacia sus páginas de búsqueda. A menudo, la diferencia entre el éxito y el fracaso de un sitio, lo determina la calidad de su búsqueda. Así que sería mejor que agreguemos un poco de búsqueda a nuestro pequeño sitio de libros, ¿no?

Comenzaremos agregando la vista para la búsqueda a nuestro URLconf (`mysite.urls`). Recuerda que esto se hace agregando algo como `(r'^search/$', 'mysite.books.views.search')` al conjunto de URL patterns (patrones).

A continuación, escribiremos la vista `search` en nuestro módulo de vistas (`mysite.books.views`):

```
from django.db.models import Q
from django.shortcuts import render_to_response
from models import Book

def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })
```

Aquí han surgido algunas cosas que todavía no vimos. La primera, ese `request.GET`. Así es cómo accedes a los datos del GET desde Django; Los datos del POST se acceden de manera similar, a través de un objeto llamado `request.POST`. Estos objetos se comportan exactamente como los diccionarios estándar de Python, y tienen además otras capacidades, que se cubren en el apéndice H.

Así que la línea:

```
query = request.GET.get('q', '')
```

busca un parámetro del GET llamado `q` y retorna una cadena de texto vacía si este parámetro no fue suministrado. Observa que estamos usando el método `get()` de `request.GET`, algo potencialmente confuso. Este método `get()` es el mismo que posee cualquier diccionario de Python. Lo estamos usando aquí para ser precavidos: *no* es seguro asumir que `request.GET` tiene una clave `'q'`, así que usamos `get('q', '')` para proporcionar un valor por omisión, que es `''` (el string vacío). Si hubiéramos intentado acceder a la variable simplemente usando `request.GET['q']`, y `q` no hubiese estado disponible en los datos del GET, se habría lanzado un `KeyError`.

Segundo, ¿qué es ese `Q`? Los objetos `Q` se utilizan para ir construyendo consultas complejas – en este caso, estamos buscando los libros que coincidan en el título o en el nombre con la consulta. Técnicamente, estos objetos `Q` consisten de un QuerySet, y puede leer más sobre esto en el apéndice C.

En estas consultas, `icontains` es una búsqueda en la que no se distinguen mayúsculas de minúsculas (*case-insensitive*), y que internamente usa el operador `LIKE` de SQL en la base de datos.

Dado que estamos buscando en campos de muchos-a-muchos, es posible que un libro se obtenga más de una vez (por ej: un libro que tiene dos autores, y los nombres de ambos concuerdan con la consulta). Al agregar `.distinct()` en el filtrado, se eliminan los resultados duplicados.

Todavía no hay una plantilla para esta vista. Esto lo solucionará:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Search{% if query%} Results{% endif%}</title>
</head>
<body>
  <h1>Search</h1>
  <form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>

  {% if query%}
    <h2>Results for "{{ query|escape }}":</h2>

    {% if results%}
      <ul>
      {% for book in results%}
        <li>{{ book|escape }}</l1>
      {% endfor%}
      </ul>
    {% else%}
```

```
      <p>No books found</p>
    {% endif %}
  {% endif %}
</body>
</html>
```

A esta altura, lo que esto hace debería ser obvio. Sin embargo, hay unas pocas sutilezas que vale la pena resaltar:

- *action* es `.` en el formulario, esto significa "la URL actual". Esta es una buena práctica estándar: no utilices vistas distintas para la página que contiene el formulario y para la página con los resultados; usa una página única para las dos cosas.
- Volvemos a insertar el texto de la consulta en el `<input>`. Esto permite a los usuarios refinar fácilmente sus búsquedas sin tener que volver a teclear todo nuevamente.
- En todo lugar que aparece `query` y `book`, lo pasamos por el filtro `escape` para asegurarnos de que cualquier búsqueda potencialmente maliciosa sea descartada antes de que se inserte en la página

  ¡Es *vital* hacer esto con todo el contenido suministrado por el usuario! De otra forma el sitio se abre a ataques de cross-site scripting (XSS). El **'Capítulo 19'_** discute XSS y la seguridad con más detalle.
- En cambio, no necesitamos preocuparnos por el contenido malicioso en las búsquedas de la base de datos – podemos pasar directamente la consulta a la base de datos. Esto es posible gracias a que la capa de base de datos de Django se encarga de manejar este aspecto de la seguridad por ti.

Ahora ya tenemos la búsqueda funcionando. Se podría mejorar más el sitio colocando el formulario de búsqueda en cada página (esto es, en la plantilla base). Dejaremos esto de tarea para el hogar.

A continuación veremos un ejemplo más complejo. Pero antes de hacerlo, discutamos un tópico más abstracto: el "formulario perfecto".

6.6 El "formulario perfecto"

Los formularios pueden ser a menudo una causa importante de frustración para los usuarios de tu sitio. Consideremos el comportamiento de un hipotético formulario perfecto:

- Debería pedirle al usuario cierta información, obviamente. La accesibilidad y la usabilidad importan aquí. Así que es importante el uso inteligente del elemento `<label>` de HTML, y también lo es proporcionar ayuda contextual útil.
- Los datos suministrados deberían ser sometidos a una validación extensiva. La regla de oro para la seguridad de una aplicación web es "nunca confíes en la información que ingresa". Así que la validación es esencial.

- Si el usuario ha cometido algún error, el formulario debería volver a mostrarse, junto a mensajes de error detallados e informativos. Los campos deberían rellenarse con los datos previamente suministrados, para evitarle al usuario tener que volver a tipear todo nuevamente.
- El formulario debería volver a mostrarse una y otra vez, hasta que todos los campos se hayan rellenado correctamente.

¡Construir el formulario perfecto pareciera llevar mucho trabajo! Por suerte, el *framework* de formularios de Django está diseñado para hacer la mayor parte del trabajo por ti. Se le proporciona una descripción de los campos del formulario, reglas de validación, y una simple plantilla, y Django hace el resto. El resultado es un "formulario perfecto" que requiere de muy poco esfuerzo.

6.7 Creación de un formulario para comentarios

La mejor forma de construir un sitio que la gente ame es atendiendo a sus comentarios. Muchos sitios parecen olvidar esto; ocultan los detalles de su contacto en *FAQs*, y parecen dificultar lo más posible el encuentro con las personas.

Cuando tu sitio tiene millones de usuarios, esto puede ser una estrategia razonable. En cambio, cuando intentas formarte una audiencia, deberías pedir comentarios cada vez que se presente la oportunidad. Escribamos entonces un simple formulario para comentarios, y usémoslo para ilustrar al *framework* de Django en plena acción.

Comenzaremos agregando `(r'^contact/$', 'mysite.books.views.contact')` al URLconf, y luego definamos nuestro formulario. Los formularios en Django se crean de una manera similar a los modelos: declarativamente, empleando una clase de Python. He aquí la clase para nuestro simple formulario. Por convención, lo insertaremos en un nuevo archivo `forms.py` dentro del directorio de nuestra aplicación:

```
from django import newforms as forms

TOPIC_CHOICES = (
    ('general', 'General enquiry'),
    ('bug', 'Bug report'),
    ('suggestion', 'Suggestion'),
)

class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField()
    sender = forms.EmailField(required=False)
```

Un formulario de Django es una subclase de `django.newforms.Form`, tal como un modelo de Django es una subclase de `django.db.models.Model`. El módulo `django.newforms` también contiene cierta cantidad de clases `Field` para los campos. Una

lista completa de éstas últimas se encuentra disponible en la documentación de Django, en http://www.djangoproject.com/documentation/0.96/newforms/.

Nuestro `ContactForm` consiste de tres campos: un tópico, que se puede elegir entre tres opciones; un mensaje, que es un campo de caracteres; y un emisor, que es un campo de correo electrónico y es opcional (porque incluso los comentarios anónimos pueden ser útiles). Hay una cantidad de otros tipos de campos disponibles, y puedes escribir nuevos tipos si ninguno cubre tus necesidades.

El objeto formulario sabe cómo hacer una cantidad de cosas útiles por sí mismo. Puede validar una colección de datos, puede generar sus propios *"widgets"* de HTML, puede construir un conjunto de mensajes de error útiles. Y si estás en perezoso, puede incluso dibujar el formulario completo por ti. Incluyamos esto en una vista y veámoslo en acción. En `views.py`:

y en `contact.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    <form action="." method="POST">
        <table>
            {{ form.as_table }}
        </table>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

La línea más interesante aquí es `{{ form.as_table }}`. `form` es nuestra instancia de ContactForm, que fue pasada al `render_to_response`. `as_table` es un método de ese objeto que reproduce el formulario como una secuencia de renglones de una tabla (también pueden usarse `as_ul` y `as_p`). El HTML generado se ve así:

```
<tr>
    <th><label for="id_topic">Topic:</label></th>
    <td>
        <select name="topic" id="id_topic">
            <option value="general">General enquiry</option>
            <option value="bug">Bug report</option>
            <option value="suggestion">Suggestion</option>
        </select>
    </td>
</tr>
<tr>
    <th><label for="id_message">Message:</label></th>
```

```
    <td><input type="text" name="message" id="id_message" /></td>
</tr>
<tr>
    <th><label for="id_sender">Sender:</label></th>
    <td><input type="text" name="sender" id="id_sender" /></td>
</tr>
```

Observa que las etiquetas `<table>` y `<form>` no se han incluido; debes definirlas por tu cuenta en la plantilla. Esto te da control sobre el comportamiento del formulario al ser suministrado. Los elementos *label* sí se incluyen, y proveen a los formularios de accesibilidad "desde fábrica".

Nuestro formulario actualmente utiliza un *widget* `<input type="text">` para el campo del mensaje. Pero no queremos restringir a nuestros usuarios a una sola línea de texto, así que la cambiaremos por un *widget* `<textarea>`:

El *framework* de formularios divide la lógica de presentación para cada campo, en un conjunto de *widgets*. Cada tipo de campo tiene un *widget* por defecto, pero puedes sobreescribirlo fácilmente, o proporcionar uno nuevo de tu creación.

Por el momento, si se suministra el formulario, no sucede nada. Agreguemos nuestras reglas de validación:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

Una instancia de formulario puede estar en uno de dos estados: *bound* (vinculado) o *unbound* (no vinculado). Una instancia *bound* se construye con un diccionario (o un objeto que funcione como un diccionario) y sabe cómo validar y volver a representar sus datos. Un formulario *unbound* no tiene datos asociados y simplemente sabe cómo representarse a sí mismo.

Intenta hacer clic en *Submit* en el formulario vacío. La página se volverá a cargar, mostrando un error de validación que informa que nuestro campo de mensaje es obligatorio.

Intenta también ingresar una dirección de correo electrónico inválida. El `EmailField` sabe cómo validar estas direcciones, por lo menos a un nivel razonable.

6.8 Procesamiento de los datos suministrados

Una vez que el usuario ha llenado el formulario al punto de que pasa nuestras reglas de validación, necesitamos hacer algo útil con los datos. En este caso, deseamos construir un correo electrónico que contenga los comentarios del usuario, y enviarlo. Para esto, usaremos el paquete de correo electrónico de Django.

Pero antes, necesitamos saber si los datos son en verdad válidos, y si lo son, necesitamos una forma de accederlos. El *framework* de formularios hace más que validar los datos, también los convierte a tipos de datos de Python. Nuestro formulario para comentarios sólo trata con texto, pero si estamos usando campos como `IntegerField` o `DateTimeField`, el *framework* de formularios se encarga de que se devuelvan como un valor entero de Python, o como un objeto `datetime`, respectivamente.

Para saber si un formulario está vinculado (*bound*) a datos válidos, llamamos al método `is_valid()`:

```
form = ContactForm(request.POST)
if form.is_valid():
    # Process form data
```

Ahora necesitamos acceder a los datos. Podríamos sacarlos directamente del `request.POST`, pero si lo hiciéramos, no nos estaríamos beneficiando de la conversión de tipos que realiza el *framework* de formularios. En cambio, usamos `form.clean_data`:

```
if form.is_valid():
    topic = form.clean_data['topic']
    message = form.clean_data['message']
    sender = form.clean_data.get('sender', 'noreply@example.com')
    # ...
```

Observa que dado que `sender` no es obligatorio, proveemos un valor por defecto por si no fue proporcionado. Finalmente, necesitamos registrar los comentarios del usuario. La manera más fácil de hacerlo es enviando un correo electrónico al administrador del sitio. Podemos hacerlo empleando la función:

```
from django.core.mail import send_mail

# ...

send_mail(
    'Feedback from your site, topic: %s' % topic,
    message, sender,
    ['administrator@example.com']
)
```

La función `send_mail` tiene cuatro argumentos obligatorios: el asunto y el cuerpo del mensaje, la dirección del emisor, y una lista de direcciones destino. `send_mail` es un código conveniente que envuelve a la clase `EmailMessage` de Django. Esta clase provee características avanzadas como adjuntos, mensajes multiparte, y un control completo sobre los encabezados del mensaje.

Una vez enviado el mensaje con los comentarios, redirigiremos a nuestro usuario a una página estática de confirmación. La función de la vista finalizada se ve así:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.core.mail import send_mail
from forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            topic = form.clean_data['topic']
            message = form.clean_data['message']
            sender = form.clean_data.get('sender', 'noreply@example.com')
            send_mail(
                'Feedback from your site, topic: %s' % topic,
                message, sender,
                ['administrator@example.com']
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

6.9 Nuestras propias reglas de validación

Imagina que hemos lanzado al público a nuestro formulario de comentarios, y los correos electrónicos han empezado a llegar. Nos encontramos con un problema: algunos mensajes vienen con sólo una o dos palabras, es poco probable que tengan algo interesante. Decidimos adoptar una nueva póliza de validación: cuatro palabras o más, por favor.

Hay varias formas de insertar nuestras propias validaciones en un formulario de Django. Si vamos a usar nuestra regla una y otra vez, podemos crear un nuevo tipo de campo. Sin embargo, la mayoría de las validaciones que agreguemos serán de un solo uso, y pueden agregarse directamente a la clase del formulario.

En este caso, necesitamos validación adicional sobre el campo `message`, así que debemos agregar un método `clean_message` a nuestro formulario:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)

    def clean_message(self):
        message = self.clean_data.get('message', '')
```

```
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Este nuevo método será llamado después del validador que tiene el campo por defecto (en este caso, el validador de un `CharField` obligatorio). Dado que los datos del campo ya han sido procesados parcialmente, necesitamos obtenerlos desde el diccionario `clean_data` del formulario.

Usamos una combinación de `len()` y `split()` para contar la cantidad de palabras. Si el usuario ha ingresado muy pocas palabras, lanzamos un error `ValidationError`. El texto que lleva esta excepción se mostrará al usuario como un elemento de la lista de errores.

Es importante que retornemos explícitamente el valor del campo al final del método. Esto nos permite modificar el valor (o convertirlo a otro tipo de Python) dentro de nuestro método de validación. Si nos olvidamos de retornarlo, se retornará `None` y el valor original será perdido.

6.10 Una presentación personalizada

La forma más rápida de personalizar la presentación de un formulario es mediante CSS. En particular, la lista de errores puede dotarse de mejoras visuales, y el elemento `<ul>` tiene asignada la clase `errorlist` para ese propósito. El CSS a continuación hace que nuestros errores salten a la vista:

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

Si bien es conveniente que el HTML del formulario sea generado por nosotros, en muchos casos la disposición por defecto no quedaría bien en nuestra aplicación. `{{ form.as_table }}` y similares son atajos útiles que podemos usar mientras desarrollamos nuestra aplicación, pero todo lo que concierne a la forma en que nuestro formulario es representado puede ser sobreescrito, casi siempre desde la plantilla misma.

Cada *widget* de un campo (<input type="text">, <select>, <textarea>, o similares) puede generarse individualmente accediendo a {{ form.fieldname }}. Cualquier error asociado con un campo está disponible como {{ form.fieldname.errors }}. Podemos usar estas variables para construir nuestra propia plantilla para el formulario:

```
<form action="." method="POST">
    <div class="fieldWrapper">
        {{ form.topic.errors }}
        <label for="id_topic">Kind of feedback:</label>
        {{ form.topic }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
        {{ form.message }}
    </div>
    <div class="fieldWrapper">
        {{ form.sender.errors }}
        <label for="id_sender">Your email (optional):</label>
        {{ form.sender }}
    </div>
    <p><input type="submit" value="Submit"></p>
</form>
```

{{ form.message.errors }} se muestra como un <ul class="errorlist"> si se presentan errores y como una cadena de caracteres en blanco si el campo es válido (o si el formulario no está vinculado). También podemos tratar a la variable form.message.errors como a un booleano o incluso iterar sobre la misma como en una lista, por ejemplo:

```
<div class="fieldWrapper{% if form.message.errors%} errors{% endif%}">
    {% if form.message.errors%}
        <ol>
        {% for error in form.message.errors%}
            <li><strong>{{ error|escape }}</strong></li>
        {% endfor%}
        </ol>
    {% endif%}
    {{ form.message }}
</div>
```

En caso de que hubieran errores de validación, se agrega la clase “errors” al contenedor y se muestran los errores en una lista ordenada.

6.11 Creando formularios a partir de Modelos

Construyamos algo un poquito más interesante: un formulario que suministre los datos de un nuevo publicista a nuestra aplicación de libros del **'Capítulo 5'_**.

Una regla de oro que es importante en el desarrollo de software, a la que Django intenta adherirse, es: no te repitas (del inglés *Don't Repeat Yourself*, abreviado DRY). Andy Hunt y Dave Thomas la definen como sigue, en *The Pragmatic Programmer*:

> Cada pieza de conocimiento debe tener una representación única, no ambigua, y de autoridad, dentro de un sistema.

Nuestro modelo de la clase `Publisher` dice que un publicista tiene un nombre, un domicilio, una ciudad, un estado o provincia, un país, y un sitio web. Si duplicamos esta información en la definición del formulario, estaríamos quebrando la regla anterior. En cambio, podemos usar este útil atajo: `form_for_model()`:

```
from models import Publisher
from django.newforms import form_for_model

PublisherForm = form_for_model(Publisher)
```

`PublisherForm` es una subclase de `Form`, tal como la clase `ContactForm` que creamos manualmente con anterioridad. Podemos usarla de la misma forma:

```
from forms import PublisherForm

def add_publisher(request):
    if request.method == 'POST':
        form = PublisherForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/add_publisher/thanks/')
    else:
        form = PublisherForm()
    return render_to_response('books/add_publisher.html', {'form': form})
```

El archivo `add_publisher.html` es casi idéntico a nuestra plantilla `contact.html` original, así que la omitimos. Recuerda además agregar un nuevo patrón al URLconf: `(r'^add_publisher/$', 'mysite.books.views.add_publisher')`.

Ahí se muestra un atajo más. Dado que los formularios derivados de modelos se emplean a menudo para guardar nuevas instancias del modelo en la base de datos, la clase del formulario creada por `form_for_model` incluye un conveniente método `save()`. Este método trata con el uso común; pero puedes ignorarlo si deseas hacer algo más que tenga que ver con los datos suministrados.

`form_for_instance()` es un método que está relacionado con el anterior, y puede crear formularios preinicializados a partir de la instancia de un modelo. Esto es útil al crear formularios "editar".

CAPÍTULO 7

La aplicación

Una vez lograda la implementación de Django en el clinete sobre el framework de aplicaciones Doff, aparece la necesidad de conectar el proyecto online con el browser. Surge la necesidad de crear un equivalente al proyecto para ser instalado en el clinete.

El desarrollo de los doff.models sigue el esquma de aplicaciones de Django.

7.1 Definición de un proyecto en el cliente

Nota: hacer la referencia al capítulo de django

Doff fue desarrollado con el objetivo de realizar la menor cantidad de reescritura posible de las aplicaciones Django para ser ejecutadas de manera desconectada. La transferencia del proyecto al clinete conlleva la transferencia de la configuración, las aplicaciones y la base de datos.

Una aplicación está conformada por al menos un módulo de vistas, uno de urls y un módulo de modelos.

En el caso de las vistas, estas deben ser reescritas debido a que si bien Porotpy está diseñado para que los dearrolladores encunetren en Javascript 1.7 una sintaxis y semántica cercana a la de Python, pueden existir módulos de la API de Python que Protopy no soprte. Así como también pueden hacer uso de módulos de terceros, el caso de ReportLab para la genreación de PDF, MatplotLib para la generación de gráficas, o alguna otra librería.

Por lo tanto vemos que no podemos realizar una reescritura uno a uno de las vistas. Además las vists que se rendericen sobre templates que accedan a datos o servicios de otros dominios tampoco pueden ser transformadas de modo directo, sinedo un caso dificilmente salvable.

En cuanto a la base de datos por razones de seguirad y eficiencia, es posible que no se desee realizar una transferencia total de los datos al cliente.

Por ejemplo, tablas que almacenan datos con información privada o confidencial, también es posible que un usuario o grupo tenga diferentes niveles de accesso sobre los datos que otros.

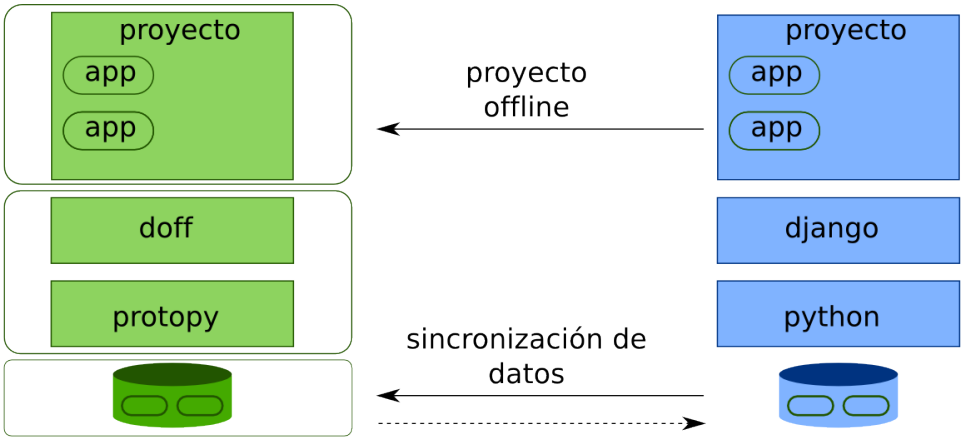


Figura 7.1: Análisis inicial de elementos a transferir para lograr una versión offline de un proyecto

Se arriba a la conclusión de que una aplicación offline puede diferir tanto en funcionalidad como en datos de una versión en linea. Por lo tanto se hace necesario definir un proyecto offline, analizando como se realiza la equivalencia de cada elemento de la versión offline.

7.2 Módulo de vistas y urls en una aplicación offline

Como hemos se analizó en el apartado anterior, las vistas deben ser reescritas para la versión offline.

7.3 Bootstrap

7.4 Transferencia de los modelos

Debido a que la API de base de datos posee diferencias mínimas, la transferencia de la definición de los modelos al cliente se ideó de tal forma que sea posible mediante introspección.

Para que el clinete conozca la defínción de un modelo, realiza una

7.5 Sitio Remoto

Se creó la entidad RemoteSite para definir un proyecto desconectado. Un proyecto puede tener uno o más RemoteSites. Cada sitio remoto está publicado en una URL.

Un sitio remoto consiste en una instancia de una clase que posee un nombre. Este nombre coincide con el nombre del directorio que almacena las vistas.

El sitio remoto tiene la responsabilidad de servir el código del framework doff y el proyecto offline (vistas, modelos, urls, templates).

CAPÍTULO 8

Sincronizacion

Se ha analizado hasta aquí la transferencia de un proyecto Django a un equivalente en un browser con soprte para Javascript 1.7 y GoogleGears. Tras la instalación de la aplicación en el cliente, se requiere la transferencia de los datos.

Muchas aplicaciones requieren cierto contendio de datos iniciales o de trabajo para poder funcionar. Estos datos residen en la base de datos del servidor y para que la aplicación offline pueda trabajar, es necesario proveer un al menos un mecanismo de transferencia de datos desde el servidor hacia el cliente.

Debido a que el desarrollo se realizó apegandose al las estructuras de Django, tanto para la definición de proyecto, como de las aplicaciones y sus componentes, se decide realizar una sincronización a nivel ORM y no a bajo nivel, es decir, se utiliza la API de acceso a datos de Django, en vez de plantear sincronización mediante SQL.

En el framework Django, el acceso a la base de datos se realiza mediante el mapeador objeto relacional, en particular para la las consultas, mediante las instancias de Managers definidas en cada modelo. En Doff se birnda la misma API, de menara que el acceso a datos en cualquiera de las aplicaciones es transparente.

8.1 Sincronización simple de servidor a cliente

Cada entidad del modelo definido en la aplicación del servidor posee al menos el manager *objects*, que equivale a un consulta por todas las filas de la tabla a la cual está asociada la entidad.

Django provee un mecanismo de serialización de QuerySets, que son los objetos que encapsulan las consultas, en varios formatos.

Figura 8.1: Esquema de sincronización simple

8.2 Identificación de instancias en el servidor

Nota: No se donde está la aclaración de pk e id para hacer la aclaración

Django provee un sub frameowork llamado *Content Types Framework* que permite generar relaciones genéricas en los instancias arbitrarias del modelo. Cada modelo posee un identificador único entero en Conent Types y en conjunción con la clave del objeto (id o pk) se obtiene una calve única para cualquier instancia, sin importar su tipo.

8.2.1 Modelos

Los modelos que son sincronizables, en el cliente deben extender al tipo *SyncModel*.

A estos modelos se los provee con un Proxy que interactua con el servidor para service de los datos

CAPÍTULO 9

Conclusiones y lineas futuras

9.1 Conclusiones

El desarrollo consistió en el siguiente diagrama:

9.2 Lineas futuras

9.2.1 Sitio de administración

Django se caracteriza por brindar una aplicación `django.contrib.admin` de admnistración que permite realizar CRUD (Create, Retrieve, Update, Delete) sobre los modelos de las aplicaciones de usuario, interactuando con la aplicación `django.contrib.auth` que provee usuarios, grupos y permisos.

9.2.2 Historial de navegación

9.2.3 Workers con soporte para Javascript 1.7

Google Gears provee un mecanismo de ejecución de código en el cliente de manera concurrente llamado Worker Pool. De esta manera tareas que demandan tiempo de CPU pueden ser envaidas a segundo plano, de manera de no entorpecer el refresco de la GUI. Una caractersística de los worker pools, es que se ejecutan en un ámbito de nombres diferente al del ‘‘hilo principal’‘. Es decir, existe encapsulamiento de su estado.

CAPÍTULO 10

Glossary

.net Plataforma de desarrollo creada por Microsoft.

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.

BSD ve ese de

CGI Common Gateway Interfase

deployment Etapa en el dearrollo de sistemas en la cual el producto es puesto en producción. El deployment involucra todas las actividades necesarias para poner el sistema en funcionamiento para el usuario final.

Deplpyment Etapa del desarrollo que consiste en la puesta en producción de una aplicación

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.

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.

HTML Lenguaje de hipertexto basado en etiquetas de marcado

HTTP Hyper Text Transfer Protocol

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.

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

Killer App Aplicación que populariza un lenguaje

MIME Estandar de especificación de tipo de contenido para correo electrónico utilizado también en encabezados HTTP.

model Models store your application's data.

MTV hola

MVC Model-view-controller; a software pattern.

PHP Lenguaje de programación diseñado para ser incrustado en documentos HTML.

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.

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.

Script Porgrama escrito en un lenguaje interpretado

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.

URL Localizador universal de recursos, especificada en la RFC 2396

view A function responsible for rending a page.

XHTML Lenguaje de hipertexto basado en etiquetas de marcado que no viola la esepcificación HTML

Bibliografía

[WikiListFramework2009] *Lista de Web Application Frameworks* http://en.wikipedia.org/wiki/List_of_web_application_frameworks

[ApacheSlingEv2009] *Evolución de Frameworks*, Alexander Klimetschek y Lars Trieloff, http://cwiki.apache.org/SLING/evolution-of-web-application-frameworks.html

[SOverAdvWebFwk2009] *Advantages of web applications over desktop applications*, StackOverflow, Dimitri C., último acceso en Septiembre de 2009, http://stackoverflow.com/questions/1072904/advantages-of-web-applications-over-desktop-applications

[CanalARRIA2005] *¿Qué son las Rich Internet Applications?*, Carlos Nascimbene, http://www.canal-ar.com.ar/noticias/noticiamuestra.asp?Id=2639

[PerezAJAX2009] *Introducción a AJAX*, Javier Eguíluz Pérez, http://www.librosweb.es/ajax/

[OSI2009] *Opne Source Intiative* http://www.opensource.org/

[NetCraft2009] *Estadisticas de utilización de servidores de NetCraft* http://news.netcraft.com/archives/web_server_survey.html

[ApacheMod2009] *Módulos del servidor Apache 2.2*, último acceso, Septiembre de 2009, http://httpd.apache.org/docs/2.2/mod/

[MicrosoftIIS2009] *Módulos en Microsoft IIS*, último acceso Septiembre 2009, http://msdn.microsoft.com/en-us/library/bb757040.aspx

[RailsQuotes2009] Citas de Ruby On Rails, http://rubyonrails.org/quotes

[ApacheTomcat2009] http://tomcat.apache.org/index.html

[SunServlet2009] http://java.sun.com/products/servlet/

[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

[DwightErwin1996] *Limitaciones de CGI*, *Using CGI*, http://www.bjnet.edu.cn/tech/book/seucgi/ch3.htm#CGILi

[DavidPollak2006] *Ruby Sig:How To Design A Domain Specific Language*, Google Tech Talk, 2:44, http://video.google.com/videoplay?docid=-8103284744220333344

[WikiPlataforma2009] *Multiplataforma*, Wikipedia, 2009, último acceso, Agosto de 2009, http://es.wikipedia.org/wiki/Multiplataforma

[TolksdorfJVM2009] *Programming languages for the Java Virtual Machine JVM*, Robert Tolksdorf, último acceso Septiembre de 2009, http://www.is-research.de/info/vmlanguages/

[PHPNetPopularity2009] *Utilización de PHP según php.net*, último acceso Septiembre de 2009, <http://www.php.net/usage.php>

[SnakesAndRubies2005] Video de la charla *Snakes and Rubies*, Universidad DePaul, Chicago http://www.djangoproject.com/snakesandrubies/

[BlogHardz2008] http://hardz.wordpress.com/2008/02/07/php-hipertexto-pre-procesado/

[YARV2009] Yet Another Ruby VM, escrita por Sacasada Kiochi http://www.atdot.net/yarv/

[JRuby2009] JRuby es una máquina virtual de Ruby escrita sobre la máquina virtual de **Java**, http://jruby.codehaus.org/

[Rubinius2009] Rubinius es una máquina vritual de Ruby escrita en **C++** http://rubini.us/

[IronRuby2009] IronRuby es una implementación de Ruby sobre la plataforma **.Net** http://www.ironruby.net/

[MacRuby2009] MacRuby es una implementación de Ruby sobre **Objective-C** para el sistema Mac OS X, http://www.macruby.org/

[EricRaymon2000] *Why Python*, Linux Journal, publicado el 1° de Mayo de 2000, http://www.linuxjournal.com/article/3882

[PythonPyPi2009] En el repositorio de proyectos del lenguaje se encuentran más 1100 resultados para paquetes relacionados con el término "web". http://pypi.python.org/pypi?%3Aaction=search&term=web&submit=search

[DhananjayNene2009] *Performance Comparison – C++ / Java / Python / Ruby/ Jython / JRuby / Groovy*, blog de Dhananjay Nene, último acceso, septiembre de 2009, http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/

[PythonOrgZen2009] *El zen de Python*, último acceso Septiembre de 2009, http://www.python.org/dev/peps/pep-0020/

[PEP333] PEP *Python Enhancement Proposals* son documentos en los que se proponen mejoras para el lenguaje

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

[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

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

[DjangoDoc2009] *Sirviendo archivos estáticos con Django*, Django Wiki, http://docs.djangoproject.com/en/dev/howto/static-files/#the-big-fat-disclaimer

[DjangoDocsConentType2009] *Framework de ConentType*, Documentación de Django 1.0, último acceso Septiembre de 2009, http://docs.djangoproject.com/en/dev/ref/models/instances/#the-pk-property

[DjangoDocsModelsKey2009] *Definición de claves foráneas*, Documentación de Django 1.0, último acceso

[WikiSSL2009] *Transport Layer Security*, Wikipedia, 2009, último acceso,

[StephenChapmanJS2009] *Javascript and XML*, Stephen Chapman, About.com,

[W3cCSS2009] *Guia breve de CSS*, W3C, español, último acceso Agosto 2009,

[IronPythonFAQ2009] *Diferencias entre IronPython y CPython* http://ironpython.codeplex.com/Wiki/View.aspx?title=IPy2.0.xCPyDifferences&referringTitle=Home

[AtulVarma2009] *Python For Javascript Programmers*, Atul Varma http://hg.toolness.com/python-for-js-programmers/raw-file/tip/PythonForJsProgrammers.html

Índice

X