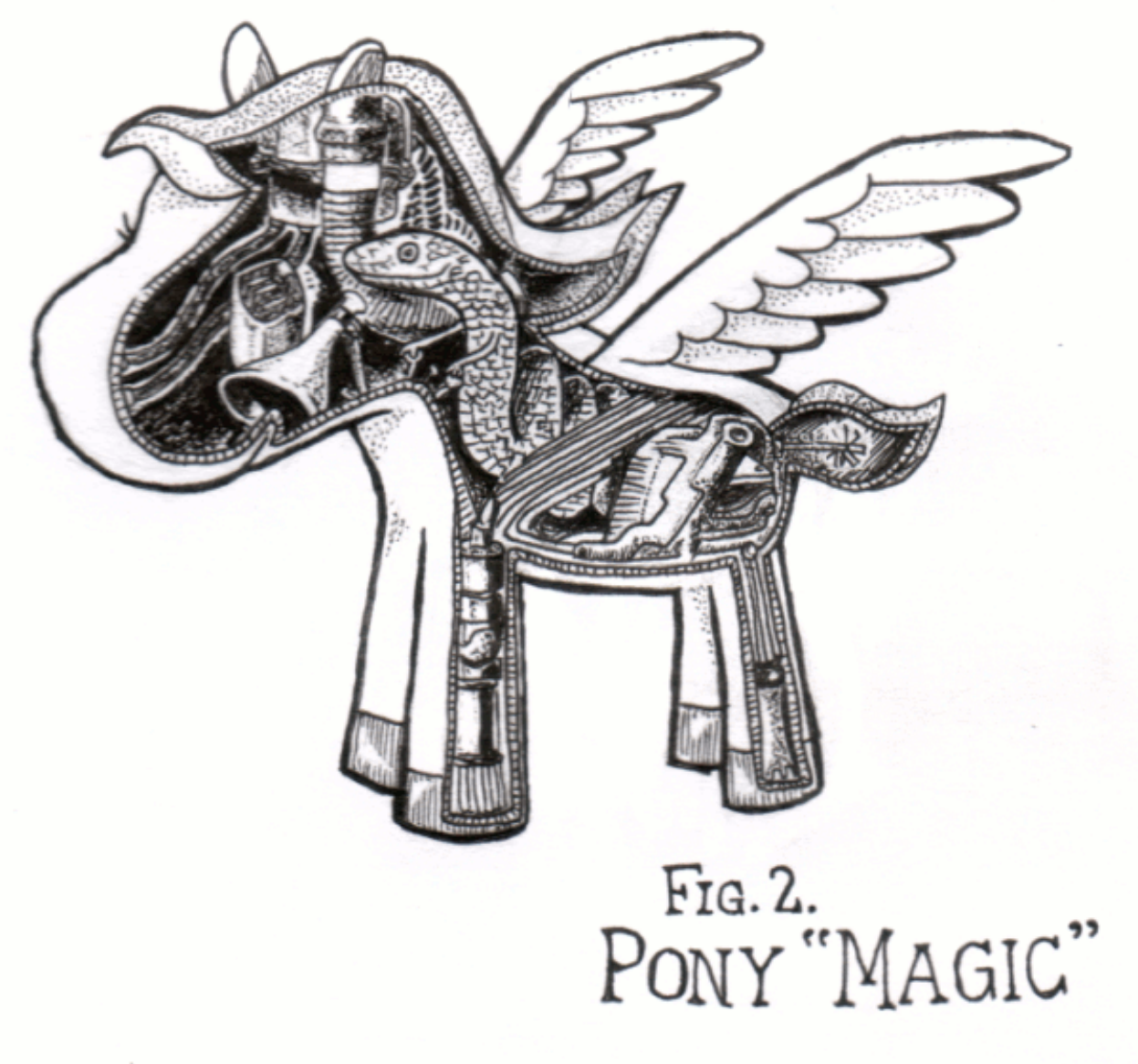

Aplicaciones Web Desconectadas

Release 1

Defossé, Nahuel; van Haaster, Diego Marcos

November 05, 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.).

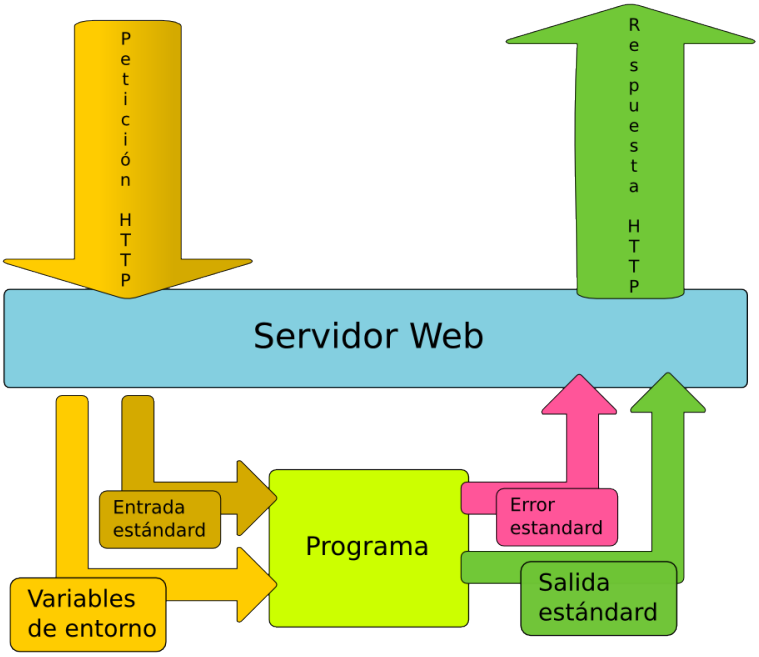

Figura 2.1: Proceso de una solicitud con CGI.

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 [3], 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.

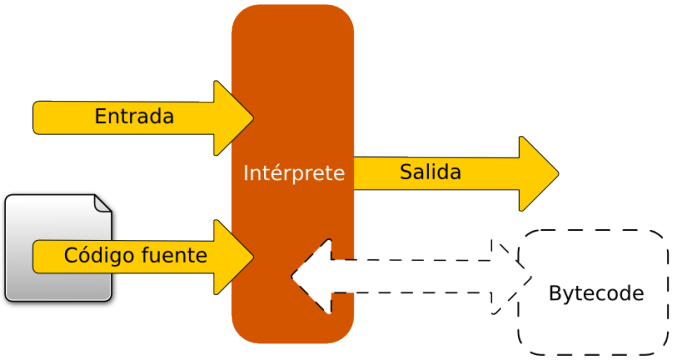

Figura 2.2: Lenguaje interpretado con máquina virtual.

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 [4].

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

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

A diferencia de los lenguajes compilados, en los cuales se requiere disponer de un compilador para la plataforma [5] y compilar el código, la mayoría de los lenguajes interpretados permiten ser ejecutados en varias plataformas (multiplataforma), ya que sólo es necesario disponer de un intérprete compilado.

Generalmente 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, esto no resulta sencillo.

En base a la clasificación antes mencionada, a continuacion 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), de los lenguajes AWK, sed y Lisp, y, en menor medida, de muchos otros lenguajes de programación. Se usa principalmente para 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 está 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 que se le hace a este lenguaje es la ambigüedad y complejidad de su sintaxis, ya que una tarea puede ser realizada de varias maneras diferentes, dando lugar a confusión en los grupos de trabajo.

Java

Es un lenguaje de programación orientado a objetos, multiplataforma y de propósito general, basado en máquina virtual, de compilación explícita. Java ha definido algunos elementos importantes en cuanto a la web dinámica, como los applets [6] y la especificación J2EE. Dos desventajas que presenta este lenguaje son la sintaxis verborrágica y el tipado estático, heredados de su predecesor C++ [SeanKellyRecFromAdict2009].

Nota: Poner citas, ver el tema en cuanto a lenguajes dinamicos sobre la jvm

[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

Como contrapartida resulta una alternativa veloz para ciertas tareas, lo que motivó el desarrollo de lenguajes dinámicos [TolksdorfJVM2009] que hacen uso de la máquina virtual de Java [7]. Entre ellos, se destaca Scala, que hoy cuenta con frameworks con una concepción posterior a 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 un lenguaje de propósito general. Toma elementos de Perl, shellscript, C y recientemente de Java.

Tradicionalmente su 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, 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 los frameworks Django y Ruby On Rails provienen del lenguaje PHP [SnakesAndRubies2005].

Ruby

Es un lenguaje orientado fuertemente a objetos, multiplataforma, creado en 1995 por Yukihiro "Matz" Matsumoto, en Japón. A menudo es comparado con *Smalltak* y se suele decir que Ruby es un lenguaje de objetos puro, ya que todo en él 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; el oficial escrito en C. ademas se conocen: YARV [YARV2009], JRuby [JRuby2009], Rubinius [Rubinius2009], IronRuby [IronRuby2009], y MacRuby [MacRuby2009].

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

La aplicación que popularizo al lenguaje es el framework "Ruby on Rails". Su versión estable oficial fue liberada en el año 2005 y represento un cambio radical al enfoque complejo de *J2EE* [RailsQuotes2009].

Ruby aún posee baja aceptación debido, quizás, a que la documentación oficial solía estar en el idioma japonés (aunque la situación se ha venido revirtiendo últimamente). Otra desventaja importante es que la velocidad del intérprete oficial es bastante baja y varía según las plataformas (cuando se la compara con otros lenguajes dinámicos similares como Python).

Python

Es un lenguaje de programación interpretado multiparadigma, de propósito general. Fue creado por Guido van Rossum en el año 1991. Se trata de un lenguaje dinámico que toma elementos de varios lenguajes, como C, Java y Scheme, entre otros.

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

La comunidad UNIX considera a Python como una evolución de Perl, con sintaxis limpia y potente. Eric Raymond, en el artículo "Why Python?", explica su conversión de Perl a Python [EricRaymon2000].

Muchos de las sistemas web basados en CGI, están escritos en Perl, por lo cual no es sorprendente encontrar una buena cantidad de proyectos Python orientados a la web [PythonPyPi2009]. Se han desarrollado varios módulos para la interconexión del intérprete de Python con un servidor web. Phillip J. Eby formuló un método, denominado WSGI (Web Server Gateway Interface), que sigue la filosofía del lenguaje [PEP333]. Informalmente se puede decir que WSGI es una traducción de CGI al lenguaje Python. Su objetivo principal es estandarizar, sobre el lenguaje, el mecanismo de comunicación entre el servidor y una aplicación.

Para satisfacer una solicitud bajo WSGI, se invoca la función de entrada con dos 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 se mustra una aplicación mínima en WSGI. La función `app` es el punto de entrada y devuelve la cadena Hello World1. Utiliza la función que recibe como segundo argumento, `start_response`, para que el cliente determine cómo tratar la respuesta (en el ejemplo como texto plano).

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

En la figura 2.3 se detallan las capas intervinientes en WSGI: el cliente genera una solicitud, el servidor discrimina qué tipo de recurso está siendo requerido. Si se trata de un recurso dinámico, la respuesta se genera a través del módulo WSGI. En este caso interviene el intérprete de Python

con sus liberías (Site Packages) y la aplicación WSGI. Los motivos por los cuales se seleccionó [8]

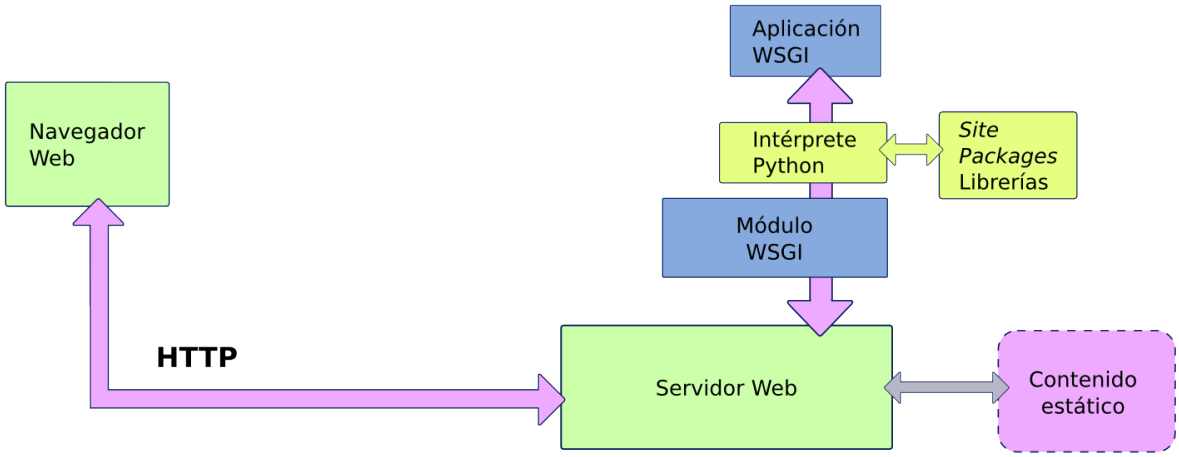

Figura 2.3: Esquema WSGI.

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.

2.2 Frameworks Web

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.

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

Sin embargo, existen ciertas características que diferencian a un framework de una librería o de una aplicación de usuario:

- Inversión de control

  Tradicionalmente las aplicaciones se escriben las haciendo llamadas a las bibliotecas de manera explícita y 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

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

- Extensibilidad

  El comportamiento predefinido de cada componente del framework generalmente es modificado por el programador con algún fin específico. Los mecanismos utilizados en los frameworks desarrollados en lenguajes orientados a objetos suelen ser de redefinición o de especialización.

- No modificabilidad del código propio del framework

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

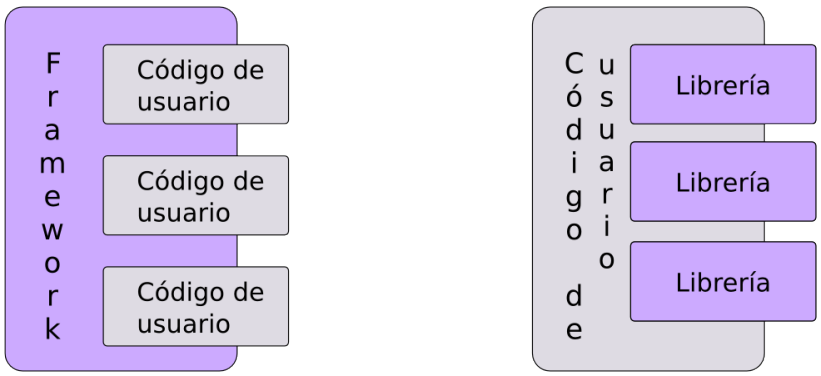

Figura 2.4: Framework vs Librería.

Al utilizar un framework para simplificar el desarrollo de un proyecto, los programadores no deben preocuparse en resolver detalles comunes de bajo nivel.

Por ejemplo, en un equipo en 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, existen una serie de argumentos comunes en contra de la utilización de frameworks:

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

2.2.1 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 tres 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 los laboratorios de investigación de Xerox. La implementación original se detalla en "Programación de Aplicaciones en Smalltalk-80(TM): Como utilizar Modelo Vista Controlador" [SmallMVC].

Las capas del patrón son:

- Modelo

  Define los datos, sus relaciones y la manera de acceder a éstos. Asegura la integridad de los datos y permite definir nuevas abstracciones de datos.

- Vista

  Es la presentación del modelo, seleccionando qué mostrar y cómo mostrarlo, usualmente es la interfaz de usuario.

- Controlador

  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 en el desarrollo de aplicaciones web. Gran cantidad de frameworks que surgieron desde entonces han aplicado en mayor o menor 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 del modelo entidad-relacion utilizado para la definición de bases de datos. Un mapeador objeto-relacional 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 [9] y consultas sobre éstos.

2.2.2 Mapeador Objeto-Relacional (ORM)

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 la 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 orientados a objetos (OO).

Como, generalmente, se utiliza 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.
- **Creación, modificación y eliminación de entidades a partir de instancias** de clases del 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 persistencia de objetos.

[9] Create, Retrive, Update, Delete

- 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 XSS [10], etc.

2.2.3 Django

Django es un framework web escrito en Python que se apoya en el patrón MVC. Surge para la administración de páginas de noticias, lo que se pone de manifiesto en su diseño proporcionando una serie de características que facilitan el desarrollo rápido de páginas orientadas a contenidos (CMS). Posteriormente los desarrolladores encontraron que su CMS es lo suficientemente genérico como para cubrir un ámbito más amplio de aplicaciones.

Su código fuente fue liberado bajo la licencia BSD en julio de 2005 (Django Web Framework). El slogan del framework es "Django, El framework para perfeccionistas con fechas límites" [11].

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

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, las cuales se clasifican en:

- Aplicaciones de usuario

  Son aquellas que resuelven algún problema específico (ej: ventas, alquileres, blog, noticias, etc.).

- 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*

    Es un sistema de comentarios genérico que permite comentar diversas entidades del modelo.

  - *django.contrib.syndication*

    Consiste en herramientas para sindicar contenido vía RSS y/o Atom.

  - *django.contrib.gis*

    Es un sistema de información geográfico.

  - *django.contrib.localflavour*

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

Provee localización (l10n) e internacionalización (i18n).

La concepción de MVC de Django difiere ligeramente del enfoque tradicional. Se denomina vista al controlador y a la vista, template; resultando en el acrónimo MTV. El controlador de MVC se presenta en Django como conjunto de funciones y asociaciones de URLs a ellas. 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 su creciente popularidad y a su sencillez, se seleccionó a Django como plataforma para la implementación del desarrollo de esta tesina, puesto que se buscan tales características en el trabajo a realizar.

Nota: Correcciones de Marta hasta 06/10/2009

Estructura de un Proyecto

Durante la instalación del framework en el sistema del desarrollador, se añade a la variable de sistema `PATH` [12] un comando con el nombre `django-admin.py`. Mediante este comando se crean proyectos y se los administra.

Un proyecto es un paquete Python que contiene 3 módulos:

- settings.py

  Configuración de la base de datos, directorios de plantillas, etc.

- manage.py

  Interfase de consola para la ejecución de comandos.

- urls.py

  Mapeo de URLs en vistas (funciones).

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

- Base de datos
- Templates
- Clases de middleware

El siguiente ejemplo muestra la creación de un proyecto:

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

En este ejemplo, un listado jerárquico del sistema de archivos mostraría la siguiente estructura:

[12] Imágenes, sonido, flash, etc.

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

Se analizan a continuación la función de cada uno de los 3 módulos de un proyecto.

Módulo settings

El modulo settings es código fuente y 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 en donde serán publicados [13] (`MEDIA_URL`, `MEDIA_ROOT`). Otras configuraciones son: idioma, zona horaria, clases de middleware [14], ruta a los templates, etc.

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, o `runserver` que lanza el servidor de desarrollo o `syncdb` que, mediante el ORM crea el esquema en la base de datos a partir del 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 de proyecto. Dentro de éste, se puede derivar el tratado de ciertas URLs en módulos similares de aplicaciones instaladas en el proyecto.

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

[13] En Django una clase middleware conecta los componentes de MTV. El usuario puede definir un comportamiento personalizado.

[14] Definidas bajo `TEMPLATE_LOADERS` en el módulo settings.

La URL en las asociaciones se define 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`. Por ejemplo, derivar el tratado de todo lo que comience con la cadena personas 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),
)
```

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 de 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 templates [15] que realizan la búsqueda de la plantilla 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.

Los componentes que puede contener una plantilla son:

- 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 lógica o define algún bloque. Algunas etiquetas son `if`, `for`, `extends`, `load`, entre otros.
- Es posible definir etiquetas de plantilla de usuario (o template tags), que actúan como una función en la plantilla.

[15] Mediante el comando `syncdb` del módulo `manage` del proyecto.

- Django posee un sistema jerárquico de plantillas que propone una estructura de herencia. Una plantilla base puede definir una serie de bloques, diseño general de una página, etc. Una plantilla particular puede utilizar la etiqueta de plantilla `extends` y redefinir uno o más bloques, conservando intacto todo lo que se encuentre fuera de los bloques redefinidos.
- Cualquier otro texto pasa literalmente a la salida.

Si bien las plantillas suelen estar orientadas a la generación de HTML, es posible obtener otros formatos.

Normalmente los archivos de plantilla 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 el módulo `settings` realizan la búsqueda cuando la vista lo requiera.

A continuación se presenta una simple plantilla de ejemplo:

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

<body>

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


</body>
</html>
```

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

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

```
def mi_vista(request):
        return render_to_response('base.html', {'nombre': 'pepe', 'titulo': 'Mi pág
```

La salida sería:

```
<html>
        <head><title>Mi página bonita</title></head>

<body>

    <p>Hola pepe,</p>


</body>
</html>
```

Estructura de una Aplicación

Una aplicación es un paquete Python que consta de dos módulos: `models` y `views`.

Para crear una aplicación se utiliza el comando `startapp` del módulo `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 `urls` de la aplicación.

Se analizan a continuación la función de cada uno de los 2 módulos de una aplicación.

Módulo models

Al crear una aplicación se genera un módulo `models` en cual se definen las clases que extienden de `django.db.models.Model`. Estas definiciones son transformadas por el ORM en tablas relacionales [16].

Por ejemplo:

```
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, también reciben parámetros obtenidos de los grupos nombrados en la URL.

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

[16] **Data Definition Language** son las sentencias encargadas de la definición de la estructura, la componen las sentencias CREATE, ALTER y DROP.

```
# 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)
```

Contenido Dinámico y Estático

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). Delega esta tarea al servidor web [DjangoDoc2009], pero para el desarrollo se provee de 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 optó por el estudio de WSGI ya que es un estándar totalmente definido en Python). En ese momento se reemplazará el uso de las vistas estáticas para ser delegadas al servidor web, tal como antes se mencionó.

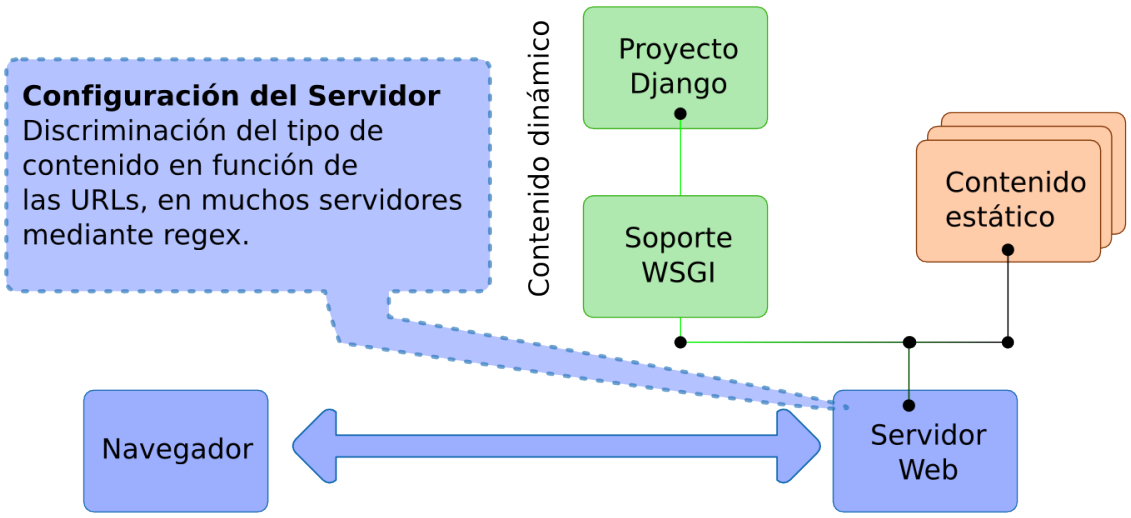

Figura 2.5: Puesta en producción de un proyecto en Django

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., que reciben el nombre de clases de middleware. Según qué métodos se implementen, pueden anteponerse o anexarse a una o más etapas del proceso del `request`.

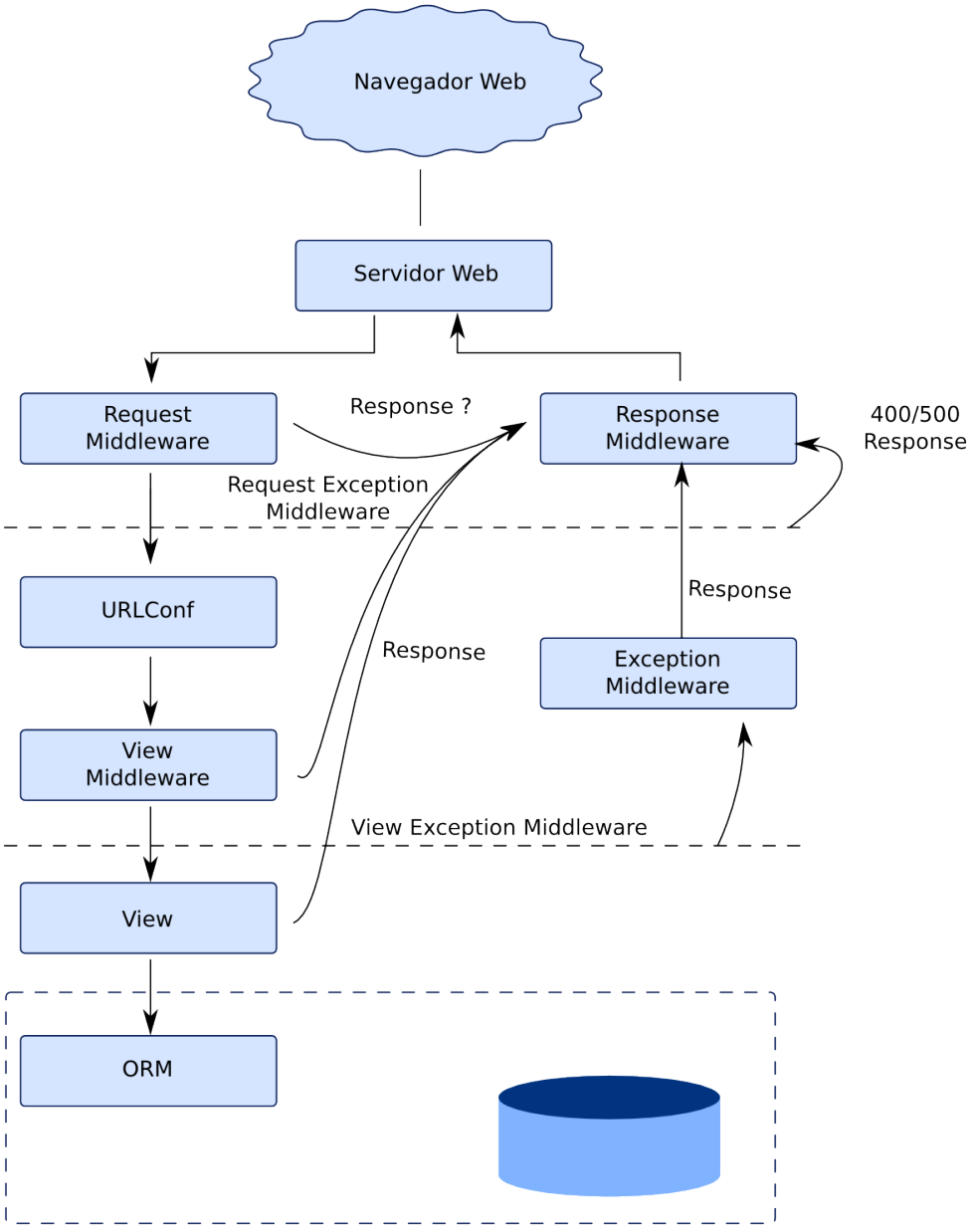

Figura 2.6: Procesamiento de un request con la interacción de los middlewares.

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 encarga 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 relacionar 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 dónde comienza dicha búsqueda. Una vez obtenida la vista, los middlewares de vistas pueden posicionar su ejecución antes o después de ésta.

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

3. Salida mediante Middleware de Response

   Finalmente, cuando la vista ha generado una respuesta, ésta es procesada por los middlewares de respuesta (`Response Middlewares`). Algunas tareas que se pueden realizar con este tipo de middlewares son: compresión de la salida, inclusión de librerías de JavaScript muy modulares, como YUI, etc.

4. Middlewares de Excepciones

   Si se lanza 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 una traza del error).

Las clases de middleware son provistas por Django y se incluyen 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 siguiente.

API del Modelo

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` de cada aplicación, en vez de realizarse en SQL.

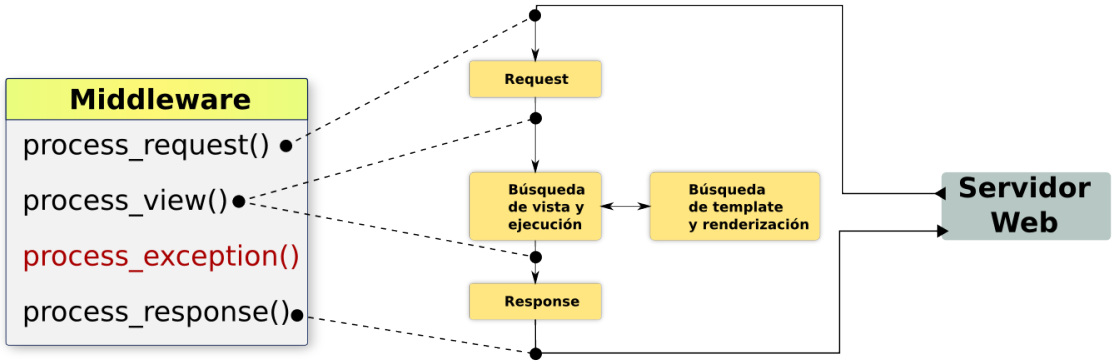

Figura 2.7: Interacción de los métodos de una clase de middleware en el proceso del `request`

Si se cuenta 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`.

Aunque 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 (Por ejemplo: `"personal.Persona"`, `"auth.User"`, `"ventas.Producto"`, etc.).

Las tareas del ORM son:

- Creación de Tablas Relacionales

  Esta operación se activa mediante el comando `syncdb` del modulo `manage` del proyecto y se realiza mediante el DDL [17] 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, éstas serán creadas.

  Por ejemplo, para la siguiente clase:

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

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

[17] **Data Manipulation Language** son las sentencias encargadas de la creación, recuperación, modificación. La compoenen las sentencias INSERT, UPDATE, DELETE.

```
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 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 ésta 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 los 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 `ContentType` [DjangoDocsContentType2009]), 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].

El atributo `pk` es un alias del campo `id` o del campo que se definió 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`, por ejemplo:

```
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

- Para crear un objeto, se importa la clase del modelo apropiada y se crea una instancia asignando valores para cada campo.
- Para guardar el objeto en la base de datos, se usa el método `save()`.
- Para recuperar objetos de la base de datos, se usa `Persona.objects`, 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 perezosa, es decir, la consulta se hace efectiva cuando es necesario acceder a los datos [18].

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:

- El atributo `objects` de cada entidad del modelo.

  El programador puede definir nuevos managers para consultas frecuentes.

  Por ejemplo:

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

- Cada relación, ya sea 1 a N o N a N.

  Por ejemplo:

  ```
  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 resuelta por Django
  ```

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.

[18] Generalmente esto ocurre en la iteración en el template.

A continuación se listan los métodos que retornan `QuerySet`:

- `filter( **argumentos )` [19]

  Sólo 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()`

  Sólo admite una instancia de cada elemento.

- `values(*campos)`

  Retorna sólo los campos `campos` como un diccionario.

- `dates(campo, tipo, order='ASC')`

  Retorna fechas.

- `none()`

  Evalúa a QuerySet vacío.

- `select_related()`

  Preselecciona en la consulta los datos relacionados para evitar el sobrecarga 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 de métodos que no devuleven `QuerySet` es:

- `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.

[19] El doble asterisco (`**`) representa argumentos del tipo `clave = valor`, mientras que el asterisco simple (`*`), define que son argumentos variables.

- `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 instancia más reciente comparando por el campo `campo`. Debe ser del tipo DateField.

Nota: Poner ejemplo de comportamiento recursivo

Formularios

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

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 coherencia global, tras la cual se obtiene el resultado de la validación.

Mediante los atributos `widget` de los campos que componen el formulario, éste puede de renderizarse como HTML.

El ciclo de trabajo normal para la entrada de datos 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 del navegador,

el cual se encarga de codificar los datos y enviarlos a la URL indicada en el atributo `action` del tag de definición del formulario.

El servidor

Recibe los datos y los valida. Si los datos son correctos, generalmente envía una respuesta `HTTP 300` indicando que la carga ha sido exitosa. Si la validación es incorrecta devuelve al usuario el formulario con los datos que hayan sido válidos, y se muestran los mensajes pertinentes de los errores que hayan ocurrido, volviendo al paso 2 del usuario.

La validación de los formularios puede ser una tarea compleja. Django provee un mecanismo para generar formularios en el módulo `django.forms` alivia la tarea.

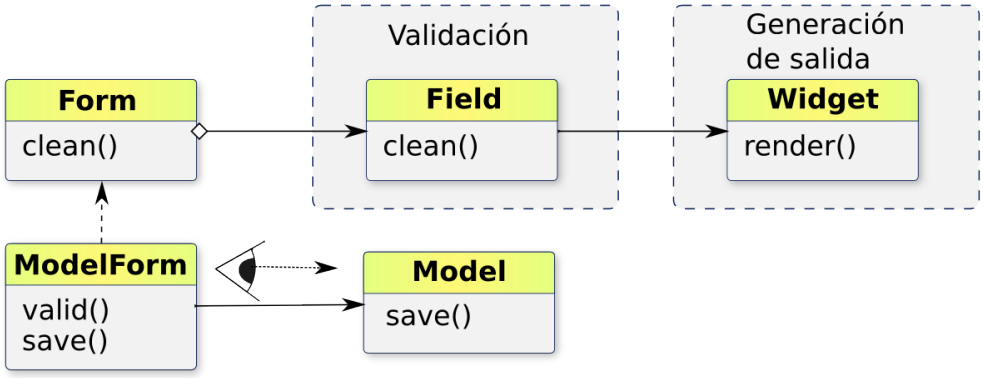

Figura 2.8: Diagrama de clases de un formulario en Django

CAPÍTULO 3

Tecnologías del Cliente

3.1 Web Dinámica Desde la Perspectiva del Cliente

Desde la perspectiva del usuario, la Web consiste en una gran cantidad de documentos interconectados a nivel mundial llamados *páginas web* a los cuales se accede mediante un navegador.

Cada uno de estos documentos está escrito en un lenguaje de marcado (o etiquetas), típicamente HTML, que utiliza a su vez dos lenguajes:

- CSS [1] para definir el estilo de las páginas de manera consistente.
- Javascript para definir la interacción en la página.

Nota: Poner como llegamos a decir "dinámica" en el cliente, sino simplemente puede ser web desde la perspectiva del cliente el titulo.

JavaScript [WikiJavascript09] ha tenido durante mucho tiempo la reputación de ser un lenguaje inadecuado para el desarrollo serio. Una de las razones fue que, a pesar de los estándares impuestos por la European Computer Manufacturers Association (ECMA) [2] [ECMAScript09], los desarrolladores de navegadores realizaron sus propias variantes del lenguaje, como Microsoft en el implementación de JScript [MSFTJScript09] para su navegador Internet Explorer (IE). Estos problemas relegaron a JavaScript a tareas prescindibles, como validación de formularios en el cliente y efectos visuales simples.

Un problema similar ocurrió con Document Object Model (DOM), el mecanismo por el cual se interactúa con el documento y el navegador, estandarizado por la World Wide Web Consortium (W3C) [3]. Sus versiones fueron nombradas como nivel DOM, existiendo nivel 0, 1, 2 y 3. No todos los navegadores implementaron por completo los niveles, dando lugar a confusión e incompatibilidades.

[1] Cascading Style Sheet
[2] Organización fundada en 1961 para estandarizar los sistemas computarizados en Europa.
[3] Consorcio internacional que produce recomendaciones para la World Wide Web.

A partir de la aparición de AJAX [4], una técnica de desarrollo web para crear aplicaciones interactivas, los desarrolladores diseñaron librerías de JavaScript que presentaron una API uniforme que salvaba las incompatibilidades existentes.

Con la popularización de estas librerías, también lo hicieron las herramientas de depuración sofisticadas como Firebug [Firebug09], lo que permitió realizar aplicaciones más complejas y compatibles con la gran mayoría de los navegadores del mercado.

Google lanzó, en 2007, un plugin para los navegadores populares llamado Google Gears que agrega tres componentes:

- Web Server

  Un servidor de archivos que se ejecuta en el cliente.

- DataBase

  Una base de datos transaccional.

- Worker Pool

  Un mecanismo para la ejecución de JavaScript como procesos.

3.2 Estructura de un Navegador

3.2.1 Navegador Web

Es un software que presenta documentos de hipertexto al usuario. Los lenguajes de codificación de hipertexto más populares son HTML y XHTML.

Un navegador no sólo interpreta los documentos de hipertexto, sino que también puede mostrar otros tipos de contenidos, como imágenes (JPEG, GIF, PNG, etc.), sonido (WAV, MP3, OGG), vídeo (MPEG, H264, RM, MOV), así como elementos interactivos (como 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 MIME.

Un navegador web acepta como entrada del usuario una URL. Una vez validada, éste descarga el recurso apuntado mediante el protocolo HTTP.

Una URL tiene el siguiente esquema, donde se pueden diferenciar varios componentes:

Los componentes de una URL son:

- Esquema

[4] *Asynchronous JavaScript And XML*, una técnica de desarrollo web para crear aplicaciones interactivas.

Figura 3.1: Formato de una URL.

Especifica el mecanismo de comunicación. Generalmente HTTP y HTTPS [5].

- Anfitrión

  Especifica el nombre de dominio del servidor en Internet, por ejemplo: *google.com*, *nasa.gov*, *wikipedia.com*, etc. Se popularizó la utilización del subdominio "www" para identificar al 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

  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:

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

- Enlace

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

Un navegador generalmente accede a documentos ubicados en la web mediante el protocolo HTTP. Sin embargo, es posible acceder a documentos locales, donde el esquema suele ser `file` y el anfitrión se encuentra ausente, por ejemplo:

file:///home/nacho/paginas/pepe.html

[5] *Hypertext Transfer Protocol Secure* es la versión segura de HTTP.

3.2.2 HTTP

HTTP es el protocolo de transferencia de hipertexto.

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

```
GET /archivos/curso_javascript.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 se conoce como una consulta HTTP o **HTTP Request**.

En la primer línea se especifica el método HTTP y el nombre del recurso, junto con la versión del protocolo que soporta el navegador (o cliente):

```
GET /archivos/curso_javascript.html HTTP/1.1
```

La segunda línea especifica el host al cual se accede. Un mismo servidor web puede estar publicado en varios dominios, mediante esta línea se puede discriminar a cuál se intenta acceder al recurso:

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

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

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

Una vez que el servidor web ha localizado y accedido al recurso, procede a enviar la respuesta, que generalmente es una página codificada en HTML.

3.2.3 HTML

HTML es un lenguaje de marcado que tiene como objetivo describir un documento de hipertexto. Un documento HTML se conforma por una serie de **tags** o etiquetas, las que tienen el siguiente formato.

Un documento HTML está delimitado por las etiquetas `html` y contiene una cabecera delimitada por `head` y un cuerpo, delimitado por `body`. Por ejemplo:

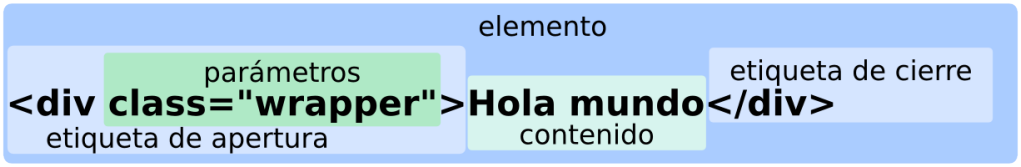

Figura 3.2: Formato de una etiqueta HTML.

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

También suele contener enlaces a recursos entendibles para el navegador, como a las hojas de estilos o código JavaScript.

La inclusión de una hoja de estilo se realiza mediante la etiqueta `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 CSS, como en:

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

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

Al incrustarse el estilo en una página en particular, éste sólo tiene validez para ese recurso.

Mediante la etiqueta `script` se incluye código JavaScript, aunque es posible utilizar otros lenguajes, como VBScript en IE. Por ejemplo:

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

De manera similar a lo que ocurre con el estilo, el código JavaScript se puede embeber en el código HTML de varias formas [StephenChapmanJS2009], entre ellas:

```
<script type="text/javascript">
    var x = 2;
    var y = 4;
</script>
```

3.2.4 CSS

CSS es un lenguaje utilizado para definir el estilo de las páginas. Una hoja de estilo en cascada o **StyleSheet** determina cómo se va a mostrar un documento en pantalla, cómo se va a imprimir o, inclusive, cómo se realiza la pronunciación a través de un dispositivo de lectura [W3cCSS2009].

El objetivo de CSS es separar el contenido de la presentación de un documento HTML o XML. Una hoja de estilos puede ser enlazada desde varias páginas, permitiendo mantener coherencia y consistencia en todo el sitio.

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. Surgió a principios de 1996, como un lenguaje de scripting para la web y enfocado en la interacción directa con el usuario.

Tiene una sintaxis semejante a la de Java y C, pero JavaScript dista mucho de ser Java y debe su nombre más a cuestiones de marketing que a principios de diseño. De hecho, fue influenciado por lenguajes como Self, Scheme, Perl, e incluso en versiones modernas, por Python.

Nota: 13/10/2009 Marta's corrections.

Si bien JavaScript es un lenguaje orientado a objetos, carece de clases y ocultamiento de información. Existen varias técnicas para lograr encapsulamiento, abstracción, herencia y polimorfismo. Entre las más utilizadas se encuentran el arreglo asociativo, el uso de prototipos y las clausuras.

Arreglo Asociativo (`Object`)

Este tipo de dato representa una lista de asociaciones clave-valor, donde la clave y el valor pueden ser de cualquier tipo arbitrario. Además el operador de indexación en el arreglo `[]` (corchetes) responde de la misma manera que el operador `.` (punto). Por ejemplo:

```
// Defincion de un
>>> var x = {nombre: "Eduardo", apellido: "Expósito", edad: 47, factor: 2.5}
>>> x['nombre']
"Eduardo"
>>> x.nombre
"Eduardo"
```

El arreglo asociativo tiene la particularidad de que en la invocación de las funciones miembro (es decir, contenidas como *valor* de alguna asociación), la palabra `this` apunta a la instancia de arreglo. De esta manera se obtiene un comportamiento similar a el tipo `struct` de C++, como se ve en el siguiente ejemplo:

```
>>> var obj = {
            metodo: function () {
               print(this.x);
            }
  x: 3
    }
>>> obj.metodo();
3
```

Esta técnica se utiliza en combinación con las clausuras en las librerías de JavaScript como Prototype [ProtypeOrgSrc09] para lograr encapsulamiento.

Prototipos

Un prototipo es el equivalente a una clase en los lenguajes como Java o C++. Consiste en la definición de la función que es llamada anteponiendo la palabra reservada `new`. Durante la llamada, `this` apunta a la instancia que está siendo creada. El código de la función se comporta como un constructor de instancias. Por ejemplo:

```
>>> var Clase = function () {
                this.metodo = function () {
                        console.log( this.valor );
                }
                this.valor = 3;
        }
>>> var instancia = new Clase();
>>> instancia.metodo();
-> 3
```

Sin embargo, la definición de métodos utilizando la sintaxis `this.metodo = ...` realiza una nueva copia del método por cada instancia.

Para acceder a la estructura de la clase, se utiliza el atributo `prototype`. Este atributo almacena la estructura de la clase y consiste en un arreglo asociativo. Como se ve en siguiente ejemplo:

```
>>> var Clase = function () {}
>>> Clase.prototype.metodo = function () {
                console.log( this.valor );
        }

>>> Clase.prototype.valor = 3;
```

```
>>> var instancia = new Clase();
>>> instancia.metodo();
-> 3
```

Esta técnica permite implementar herencia.

Clausuras

Una clausura es la utilización de funciones internas con referencias locales que permanecen enlazadas aún cuando el contexto contenedor ha desaparecido [StuartLangridgeClosures09]. Mediante esta técnica se suele lograr encapsulamiento y ocultamiento de información. Por ejemplo:

```
>>> function mostrar_saludo(nombre) {
            var saludo = "Hola";
            function saludar(){
                    print(saludo);
            }
            return saludar;
}
```

Son muy utilizadas para funciones relacionadas con temporizadores, manejadores de eventos y respuestas asincrónicas.

3.2.6 DOM

Document Object Model [6] es una API para documentos XML y HTML. Provee una representación en objetos de la estructura del documento, que permite modificar tanto el contenido como la representación visual. Su función esencial es conectar al navegador con un lenguaje de programación [MDCDOM09].

Cada fabricante de navegador web [WIKIDOM09] realizó en principio su propia implementación de DOM, razón por la cual la W3C emitió una especificación, en octubre de 1998, denominada **DOM.Nivel 1** en la cual se consideraron las características y manipulación de todos los elementos existentes en los archivos HTML y XML [W3CDomLevels09].

En noviembre de 2000 se emitió la especificación del **DOM.Nivel 2**. En ésta se incluyó la manipulación de eventos en el navegador, la capacidad de interacción con CSS, y la manipulación de partes del texto en las páginas web.

DOM.Nivel 3 se emitió en abril de 2004; utiliza DTD (Definición del Tipo de Documento) y validación de documentos.

[6] A veces traducido como Modelo en Objetos para la representación de Documentos o también Modelo de Objetos del Documento.

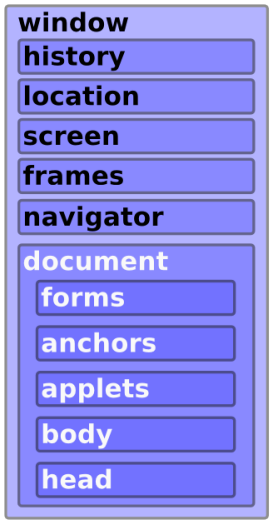

Figura 3.3: Modelo de objetos de DOM

DOM define una estructura jerárquica de objetos, donde `window` representa la ventana o pestaña del navegador. `window.history` hace referencia al historial de navegación. Por ejemplo, el siguiente código, retrocede una página en el historial:

```
window.history.back();

// Pero como window es el ámbito global, se puede abreviar a

history.back();
```

El elemento `document` referencia al documento (X)HTML. Posee los atributos `body` y `head` que representan los bloques homónimos en HTML, una serie de métodos para recuperación de elementos (`document.getElementByID()`, `document.getElementsByTagName()`), atajos a los formularios y otros elementos (`document.forms`, `document.anchors`, `document.applets`, etc.). También tiene la capacidad de crear elementos.

Toda la jerarquía que deriva de `document` es instancia del tipo `HTMLElement`, el cual le confiere una API que permite gestionar el árbol jerárquico. Algunos elementos de esta API son:

```
children // Lista de sólo lectura de nodos hijos
appendChild(el) // Agrega un elemento
parentNode  // Apunta al elemento padre
```

Mediante esta API se pueden realizar tareas como atender eventos y modificación de DOM, como se ve en el siguiente ejemplo:

```
<html>
    <head>
```

```
        <title>Pruebas de eventos</title>
        <script>
            window.onload = function () {
                var link = window.document.getElementById('un_link'),
                    div = document.getElementById('cosa');
                link.onclick = function (event) {
                    // Crear un elemento párrafo
                    var un_p = document.createElement('p'),
                        txt = prompt('Ingrese un texto');
                    un_p.innerHTML = txt;
                    div.appendChild(un_p);
                }
            }
        </script>
    </head>

    <body>
        <a id="un_link">Pulsar aquí</a>
        <div id="cosa">

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

El código JavaScript de un documento se evalúa en el contexto del elemento `window`, como espacio de nombres global. Es decir, que cualquier variable global pertenece a `window`. Por ejemplo:

```
>>> window.x = 3;
>>> x
3
```

3.2.7 AJAX

AJAX (Asynchronous JavaScript And Xml) es una tecnología que surge tras la necesidad de agilizar las interfaces de usuario basadas en la web. Es utilizada para lograr RIAs basadas en en las capacidades nativas del navegador (sin plugins de terceras partes [7]).

Consiste en la capacidad del navegador de originar peticiones HTTP que no inicien una carga del documento (segundo plano). Permite realizar interfaces web más interactivas, debido a que las transferencias asincrónicas sólo recuperan del servidor los elementos que requieran ser actualizados.

AJAX es una tecnología asincrónica 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

[7] Flash, Sliverlight, Applets Java, JavaFx, etc.

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 el objeto `XMLHttpRequest`. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.

Estas peticiones se programan en JavaScript y, si bien originalmente se pensó en XML como lenguaje de intercambio de datos, es posible transferir cualquier tipo de archivo como código HTML, código JavaScript, imágenes, hojas de estilos, JSON, etc.

Esta tecnología utiliza 4 elementos [WIKIAJAX09]:

- XHTML (o HTML) y hojas de estilos en cascada (CSS) para el diseño que acompaña a la información.
- DOM como método de control de la representación y el navegador por parte de JavaScript.
- 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 pre-formateado, texto plano, JSON y hasta EBML.

Funcionamiento

1. Se crea y configura un objeto `XMLHttpRequest`:

```
var req = new XmlHTTPRequest();
req.open('GET', 'http://host.com/uri');
// Configuración del callback del evento
req.onreadystatechange = function () {
    // Al igual que como se vio en el comportamiento
    // de un arreglo asociativo, this se refiere a la
    // petición.
    if (this.readyState == 4) {
        if (this.status == 200) {
            alert("Se a completado la descarga asincrónica");
        }
    }
}
```

2. El objeto `XMLHttpRequest` realiza una llamada al servidor:

```
req.send();
```

3. La petición se procesa en el servidor.

4. El servidor retorna un documento XML (o algún otro tipo) que contiene el resultado.
5. El objeto `XMLHttpRequest` llama a la función `onreadystatechange` y procesa el resultado.
6. Se actualiza la página mediante DOM.

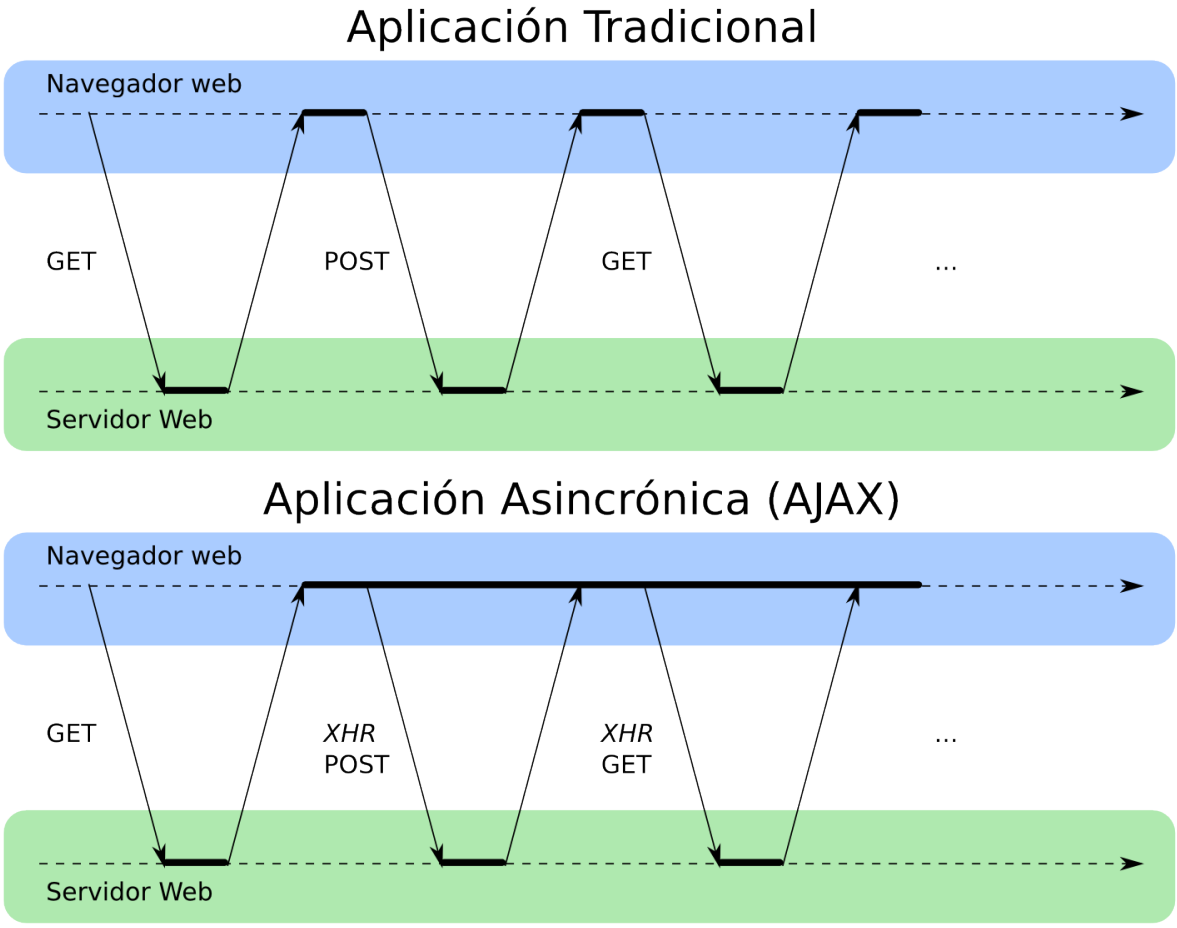

Figura 3.4: Compraración entre una aplicación tradicional y una asincrónica. La línea gruesa indica el ciclo de vida de la página en el cliente y las peticiones en el servidor.

AJAX presenta ciertas ventajas para realizar aplicaciones interactivas, como mayor interactividad, reducción de latencia de las aplicaciones y uniformidad entre las plataformas. Sin embargo presenta algunos detalles a tener en cuenta debido a que, al no realizarse recargas del documento, la URL permanece estática. Entre ellas se pueden mencionar: complicaciones en la manipulación del botón *Volver hacia Atrás* del navegador (que requiere a veces `iframes`), problemas para agregar favoritos y dificultades para la impresión.

3.2.8 JSON

JSON [JSONOrg2009] (JavaScript Object Notation) es un estándar de codificación de datos, inspirado en la sintaxis de objetos de JavaScript. Sus objetivos son:

- Ser legible para los programadores.
- Ser fácil de interpretar para las computadoras (en principio debido a la cercanía con JavaScript).

Surge como alternativa a XML para el intercambio de datos en aplicaciones web.

Los desarrolladores descubrieron que formateando los datos como una cadena literal para ser luego interpretada por la sentencia `eval()` [JSONOrgJS09] podían visualizar los datos que eran enviados al cliente en un formato legible de gran ayuda a la hora de realizar depuración.

La sentencia `eval()` es considerada peligrosa [SimonWillson24Ways09], debido a que abre la posibilidad de inyectar código de meanera directa sobre el navegador, razón por la cual las librerías de JavaScript comenzaron a brindar mecanismos seguros para JSON.

Actualmente algunos navegadores incorporan un intérprete nativo de JSON [MozillaMDCJSON-Nativo09] [IEBlogNativeJSON09], que ofrece incorpora chequeos de seguridad y un tiempo de respuesta corto. Algunas librerías de JavaScript sacan provecho de este intérprete nativo [DaveWardEncosia2009].

JSON es ampliamente utilizado como formato de intercambio de datos en AJAX. Un ejemplo de esta ulilización se ve en el siguiente código:

```
var http_request = new XMLHttpRequest();

// Esta URL debería devolver datos JSON
var url = "http://example.net/jsondata.php";

// Descarga los datos JSON del servidor.
http_request.onreadystatechange = handle_json;
http_request.open("GET", url, true);
http_request.send(null);

function handle_json() {
  if (http_request.readyState == 4) {
    if (http_request.status == 200) {
      var json_data = http_request.responseText;
      var the_object = eval("(" + json_data + ")");
    } else {
      alert("Ha habido un problema con la URL.");
    }
    http_request = null;
  }
}
```

Donde el servidor envía una respuesta de la siguiente manera:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

El equivalente en XML es:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

3.3 Google Gears

Google Gears [GGGears09] es un plugin de código abierto distribuido por Google que añade tres componentes al navegador.

Una vez instalado como una extensión en el navegador, el producto agrega una API que permite programar en JavaScript interacciones con los componentes que contiene. Esta API se añade al DOM.

Los tres componentes principales que incorpora Gears son:

- Local Server

  Permite almacenar localmente datos correspondientes a las páginas web. Tanto HTML, JavaScript, imágenes, etc., pueden ser almacenados localmente por el cliente e interponerse entre el requerimiento del navegador al servidor en consultas posteriores, evitando así la solicitud HTTP y optimizando el tiempo de respuesta de la aplicación.

  Pese a que su funcionamiento es muy similar al de la caché del navegador, la diferencia fundamental está en que la actualización de los recursos que almacena es realizada y mantenida por el desarrollador.

- DataBase

  Permite almacenar localmente datos que no correspondan a una página web pero son parte de la lógica de la aplicación y requieren de un almacenamiento persistente.

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

  Luego de que el usuario de la aplicación web otorgue el permiso explícito de creación de la base, el desarrollador puede disponer de un almacenamiento del tipo relacional en la máquina huésped.

- Worker Pool

  De manera similar a los hilos del sistema operativo, éste 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 éste se detenga o no se ejecute directamente.

Básicamente Gears y sus principales componentes están enfocados en permitir al programador ejecutar sus aplicaciones cuando el navegador no está conectado al servidor. Bret Taylor, el líder del grupo de desarrollo, dijo que buscaba ser capaz de acceder al Google Reader mientras usaba la conexión de la compañía, la cual frecuentemente tenía un acceso defectuoso a Internet [BretTaylor09].

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

A partir de la versión 0.4 se añadieron nuevas características como:

- API para GIS, para el acceso a la posición geográfica del usuario.
- API Blob, para la gestión bloques de datos binarios.
- Acceso a archivos en el equipo cliente a través de la API de Desktop.
- Envío y recepción Blobs con la API `HttpRequest`.
- Traducción de los cuadros de diálogo de Gears al idioma local (i18n).
- API para Canvas, para la manipulación de imágenes desde JavaScript.

Gears permite desarrollar aplicaciones en JavaScript que funcionen de manera desconectada. Para esto se necesita instalar el plugin y la aplicación.

CAPÍTULO 4

Introducción al Desarrollo

En este capítulo se realiza una breve descripción del análisis de las tecnologías evaluadas para llevar a cabo la desconexión de una aplicación web.

Tras la elección de Django como framework, se adoptó como estrategia inicial, para desarrollo de la aplicación de esta tesina, intentar ejecutar un intérprete de Python en el navegador web para una posterior ejecución de Django. Esto precisa que la versión del intérprete posea al menos los paquetes de la librería estándar `re`, `sys`, `time`, `urllib`, `datetime`, `mimetypes`, entre otros, que son utilizados por el framework.

Además de la posibilidad de ejecución del intérprete en el navegador, se necesitó un sistema de almacenamiento persistente en el cliente tanto para el código de la aplicación (framework + aplicación) como para los datos (típicamente un RDBMS).

Otro aspecto importante que se tuvo en cuenta, además del intérprete, el almacenamiento local y la base de datos, es que los frameworks web están diseñados para ser ejecutados en un entorno cliente-servidor. La interacción con una aplicación web se realiza generalmente mediante links, formularios y AJAX, y todas estas técnicas se traducen en alguna primitiva HTTP. En ausencia del servidor se debió realizar una adaptación de su funcionamiento.

También se analizaron otro tipo de consideraciones, como la seguridad. Transferir los datos de una aplicación en linea a una que se transporta en un navegador puede tener implicancias en la integridad de la información, ya que no es posible lograr un grado de aseguramiento al de un servidor web para una máquina potencialmente desconocida.

Además fue importante tener en cuenta que el acceso a los datos en una aplicación web está restringido por la propia aplicación. El usuario no tiene acceso a la base de datos, sino a la visión que la aplicación le da sobre ésta. Se puede decir que cada usuario o grupo tiene asociada una *perspectiva* de los datos.

Si se plantea que la transferencia de una aplicación web del servidor al cliente implica la copia de su base de datos, un usuario con suficientes conocimientos podría tener acceso a información que de otra manera no tendría (cuentas de usuario, registros de actividad, información económica

o financiera, etc.). Por esto fue importante analizar *qué datos puede ver cada usuario, grupo o rol en el sistema*. La aplicación desconectada debería poseer una técnica de encartamiento sobre los datos (lo que podría repercutir en el desempeño) o trabajar con una base de datos reducida.

De lo anterior se puede infirió que no todas las aplicaciones desconectadas son idénticas, sino que en función del usuario tendrán más o menos funcionalidad y datos asociados. Además, en una aplicación desconectada no se requiere autenticación, o al menos, no de la misma manera que en la aplicación en linea, donde la autenticación suele encontrarse asegurada mediante SSL.

Otro aspecto considerado fue la posibilidad de sincronizar las instancias de aplicaciones desconectadas con la aplicación web original.

4.1 Python en el Navegador

Sobre la plataforma Windows, existen dos formas de ejecutar Python en el navegador. La primera consiste en la ejecución del intérprete embebido en un control ActiveX. 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 elementos incrustables 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 de éstos. Si bien esta técnica es 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 *Plug-in For Hosting ActiveX Controls* [1] pero no es multiplataforma.

La segunda alternativa es utilizar la tecnología Silverlight de Microsoft, que permite generar aplicaciones para navegadores, mediante la plataforma .NET. Silverlight es un plugin similar al popular Adobe Flash, pero las aplicaciones pueden ser creadas en cualquier lenguaje de la plataforma .NET, incluyendo Python [PythonMailistMay07] y Ruby [IronRubyNet09]. En .NET todos los lenguajes compilan a un bytecode llamado Common Language Runtime, para el cual existe un solo intérprete, la propia plataforma .NET.

IronPython [MichaelFroodIP09] es una implementación de Python sobre .NET que en un principio no contaba con la API estándar [PythonDocAPI09], sino que permitía utilizar sólo la propia de .NET, por lo que Django no podía ser ejecutado. En la versión 2.0 de IronPython se implementó la API estándar lográndose ejecutar Django sobre IronPython [InforQDjangoIP09].

Gracias a la posibilidad de acceso a DOM por medio de una aplicación construida con Silverlight [MSDNSilverlightDOM09] [SwOnCodeSlvlgth09] y al almacenamiento local en el cliente introducido en Silverlight 2.0 [DinoEspositoSlvlght09], esta tecnología brinda las herramientas para ejecutar Django en el cliente sin conexión [AshishShettySlvlght09].

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

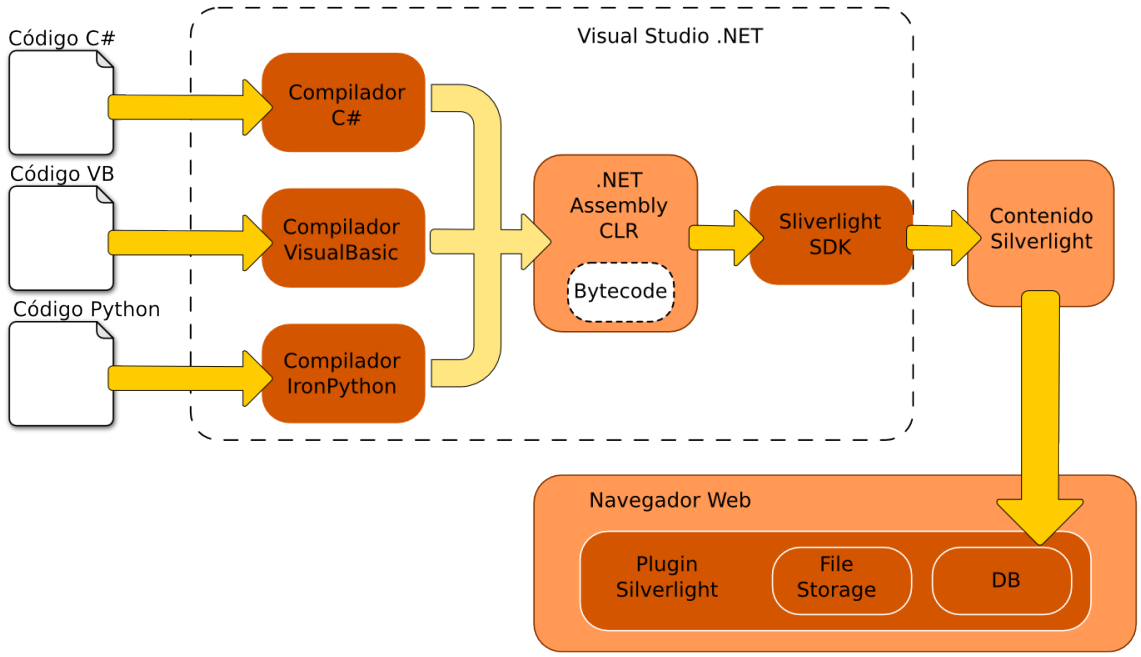

Figura 4.1: Esquema de desarrollo de Silverlight

Sin embargo, la arquitectura de software necesaria para desplegar este tipo de aplicaciones es considerablemente compleja lo que va en contraposición a los ideales de Python y Django. Además tiene varias limitaciones:

- Necesidad de plugin propietario

  Es necesario un plugin en el navegador que no se encuentra disponible para todas las plataformas.

- Herramientas de desarrollo no multiplataforma

  Las herramientas de desarrollo sólo están en un estado más maduro sobre la plataforma Windows. Si bien existen compiladores gratuitos, las IDEs que permiten un desarrollo más eficiente son propietarias.

- No existe soporte para IronPython en la IDE VisualStudio
- La implementación de Python no es la estándar, y carece de soporte [IronPythonFAQ2009] .

Debido a estas limitaciones se descartó para el desarrollo de esta tesina a Silverlight como tecnología de soporte.

En la plataforma Mozilla, la integración con Python se puede realizar mediante PyXPCOM [2] , PyShell [3] y también existe una extensión para XUL [4], pero al igual que con Silverlight, es una solución engorrosa.

Luego de evaluar las alternativas actuales de ejecución local de Python, se decidió analizar la posibilidad de realizar la aplicación del cliente utilizando las tecnologías propias del navegador.

4.2 Lenguaje de Aplicación en el Cliente

Como ya se introdujo en los apartados teóricos, JavaScript es el lenguaje de programación presente en todas las implementaciones de los navegadores web.

Javascript y Python parecen lenguajes bastante diferentes en su sintaxis, sin embargo comparten ciertas características como ser orientados a objetos y permitir la definición de clausuras [AtulVarma2009]. A partir de la versión 1.7 y 1.8, JavaScript incluye semántica funcional en los arreglos, generadores e iteradores, getters y setters, características que los acercan aún más [SteveLeeJs17Py09], tal como lo expresa Guyon Morée en la publicación titulada "Javascript Pythonico, es Python con llaves [5]" [GuyonMoreePythonBraces09].

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

[3] *PyShell*, consola interactiva.

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

[5] En Python existe un *huevo de pascua* relacionado con las características nuevas que se incluyen en el lenguaje, que se activa mediante la sentencia `from __future__ import braces` y produce la excepción `SyntaxError: not a chance` (*Not a chance* se traduce coloquialmente como "*imposible*, *olvídalo!*").

JavaScript no posee mecanismos de almacenamiento local. Si bien un navegador almacena muchos recursos de este tipo en su caché, lo hace con el objetivo de mejorar la performance y su permanencia en el equipo del cliente no está garantizada. Para lograr que una aplicación escrita en JavaScript pueda ejecutarse desconectada es necesario almacenar los recursos que componen la aplicación en el cliente mediante alguna técnica que no sea la caché.

Uno de los objetivos de Google Gears es el almacenamiento local y lo implementa mediante el módulo `Local Server`. El programador a través de su API genera repositorios de almacenamiento de recursos en línea (URLs) para proveerlos a través de un servidor web interno cuando el navegador no cuente con conexión.

Además de almacenamiento local, Gears provee una base de datos, satisfaciendo las necesidades para la creación de aplicaciones desconectado que se plantearon anteriormente. Gears al contrario que Silverlight es de código abierto y en la especificación HTML 5 (actualmente en desarrollo) [W3CHTML5OffWebApp09] se incluyen varios de los componentes que éste provee [ScottLoganbillHTML5Gears09]. Es decir, varios de los componentes de Gears serán incluidos nativamente en los navegadores que adopten este estándar.

Por lo tanto, la combinación de JavaScript y Gears constituye la alternativa más promisoria para la implementación de aplicaciones desconectadas escritas en Django por lo que se utilizaron para llevar a cabo el desarrollo de la presente tesina.

Tras el análisis del proyecto *Gears On Rails* [GearsOnRails09] que persigue objetivos similares a los de esta tesina, pero basado en el framework Ruby On Rails, se descubrió un proyecto denominado Django Offline [DjangoOffline09], que divide a Django en componentes y analiza cuáles son necesarios implementar en el modo desconectado:

- API de modelos: no

  Los modelos se definen únicamente en el servidor se utiliza la definición para el cliente. Los modelos, en su estructura, deben estar sincronizados.

- Soporte para base de datos: sí

  Sólo se requiere soporte para SQLite.

- Managers de modelos: sí
- Despacho de URLs: sí
- Middlewares: no
- Formularios: no

Basados en los componentes anteriores, también realiza un análisis de las partes transportables de manera automatizada al cliente:

- Modelos: sí
- URLs: sí
- Vistas: no

El enfoque del proyecto Django Offline no contempla los recaudos de seguridad que se deben tener en cuenta al momento de transferir la base de datos al cliente, ni especifica cómo tratar el manejo de elementos activos (links, formularios, AJAX). Sin embargo, pone de manifiesto que no se necesita reimplementar la totalidad del framework en JavaScript ni la totalidad del proyecto, reduciendo el tiempo de desarrollo y logrando mayor cohesión entre las partes.

4.3 Análisis de Migración de Componentes

El primer componente del framework que se analizó fue el ORM, el cual debe ser migrado a JavaScript para conservar la semántica de acceso a datos de las aplicaciones escritas en Django. Se analizaron los ORM existentes en JavaScript que trabajen sobre Gears.

Uriel Katz implementó Gears ORM, que luego reimplementó bajo el nombre de JStORM [Uriel-KatzJStORM09], que permite definir la estructura de las tablas y realizar consultas. Por ejemplo para definir una tabla:

```
var Person = new JStORM.Model({
  name:"Person",
  fields:
  {
     firstName:new JStORM.Field({type:"String",maxLength:25}),
     lastName:new JStORM.Field({type:"String",maxLength:25}),
  },
  connection:"default"
});
```

Presenta la particularidad de evaluación perezosa de las consultas. Cada consulta devuelve una instancia de `Query`, similar a los `QuerySet` persentes en Django. Desafortunadamente la definición del criterio de selección es de bajo nivel, como se observa en el siguiente ejemplo:

```
var katzFamily = Person.filter("lastName = ?","Katz");
katzFamily.each(function(person)
{
  console.log(person.firstName);
});
```

El autor se centró en algunas características como conexiones con múltiples bases de datos, que aún no están presentes en Django, pero abandonó el proyecto sin implementar elementos esenciales como las claves foráneas y las relaciones muchos a muchos.

JazzRecord es otro ORM analizado, que implementa la API ActiveRecord (la misma de ORM de Ruby On Rails). Se encuentra en un estado mucho más maduro que JStORM, incluyendo características como soporte para varias bases de datos, como Adobe AIR o Titanium PR1, validación de datos a nivel modelo, etc. Un ejemplo de utilización para definir una tabla es:

```
var Programmer = new JazzRecord.Model({
  table: "programmer",
  columns: {
    name: "text",
    income: "float"
  },
  validate: {
    atUpdate: function() {
      this.validatesIsString("name", "You must have a name!");
      this.validatesIsFloat("income" "We will glady pay you a null salery!");
    },
    atSave: function() {
      this.validatesIsString("name", "You must have a name!");
      this.validatesIsFloat("income" "We will glady pay you a null salery!");
    }
  }
});
```

Las consultas se relizan mediante `finders` [NickCarterJazzModels09], que equivalen a los `QuerySet`. JazzRecord fue descartado como candidato a ORM del cliente en la presente tesina, por la diferencia de filosofía con Django (convesión (Rails) vs. definición (Django)).

Luego del análisis de los ORM existentes se decidió implementar el ORM de Django (API de base de datos) en JavaScript 1.7 como primera tarea para la desconexión de una aplicación. Con una implementación en JavaScript de la misma API, y gracias a que Python soporta introspección, es posible generar las defincciones en JavaScript de modelos para el ORM a partir de los módulos `models` del proyecto Django.

Otro elemento importante para lograr una aplicación desconectada, es el manejo de los elementos activos (links, formularios y AJAX). DOM permite capturar los eventos `click`, que se genera cuando se activa un enlace, y `submit`, cuando se envía un formulario. Para AJAX se puede realizar un enmascaramiento del elemento `XMLHttpRequest`.

La selección de elementos en DOM es verborrágica (cuando los elementos carecen de `id`) y el manejo de eventos es primitivo (por ejemplo, no implementa el patrón Listener de manera consistente), por lo que se decidió utilizar la librería Prototype, que además de simplificar el manejo de eventos y presentar técnicas avanzadas de selección (de elementos), implementa un sistema de clases.

Al comenzar a implementar el ORM de Django sobre el sistema de clases de Prototype, se observó que había ciertas diferencias de sintaxis que no eran producto de las diferencias entre Python y JavaScript. La construcción de clases en Python se realiza en dos fases (creación de instancia `__new__` e inicialización `__init__`), mecanismo que en Prototype no existe (Prototype toma la orientación a objetos de Ruby). Esta inicialización en dos fases se utiliza en el ORM y formularios de Django, por lo que se decidió modificar Protoype para que admitiera la orientación a objetos de Python.

Otro elemento ausente en Prototype [6] es un sistema de módulos y paquetes, necesario para implementar muchos elementos de Django. Por ello se decidió implementar esta característica y además se agregaron la mayoría de las funciones integradas del intérprete de Python [7], como `isinstance`, `issubclass`, `map`, `int`, `bool`, etc.

Esta modificación de librería se bautizó como Protopy y su objetivo es facilitar la migración de código Python a JavaScript (en particular en esta tesina, la migración del framework y proyectos Django).

En la siguiente figura se muestra la estructura de Protopy y su interacción con JavaScript, DOM y el proyecto Django desconectado:

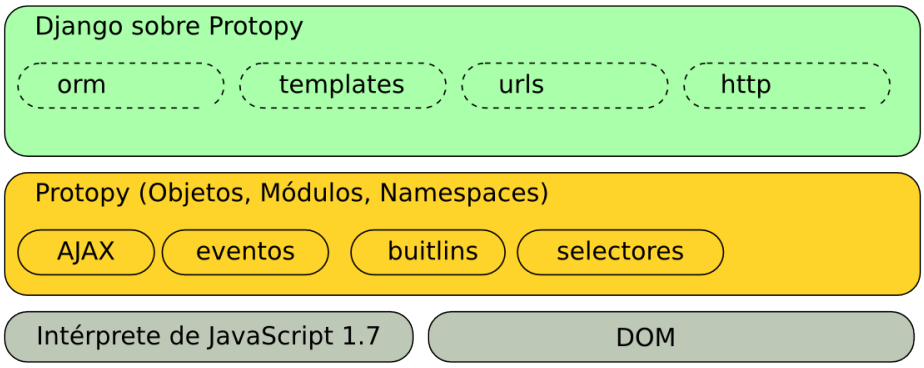

Figura 4.2: **Protopy**, Librería desarrollada en JavaScript para soporte de Python sobre JavaScript 1.7

En los siguientes capítulos se detalla el desarrollo de esta librería y a la migración de Django sobre Protopy.

[6] También ausente en JavaScript.
[7] Conocidos como *buitins*.

CAPÍTULO 5

Protopy

El presente capítulo profundiza sobre el desarrollo de la librería de JavaScript sobre la cual se implementó una versión desconectada de Django. Como se mencionó en el capítulo anterior, si bien JavaScript en sus versiones 1.7 y 1.8 incorpora muchos elementos que acercan su sintaxis a la de Python, fue necesario implementar esta capa intermedia entre JavaScript y Django "desconectado". Uno de los objetivos más relevante es emular la API estándar de Python.

La biblioteca se basó en el código fuente de Prototype, sobre la cual se fue agregando y sustituyendo código. El resultado de esta modificación se denomino Protopy en honor a sus dos "padres", *Prototype JavaScript Library* y el lenguaje de programación *Python*:

> **proto** type + **py** thon = **protopy**

5.1 Características de la Librería

Algunas de las características más relevantes de Protopy son:

- Modularización y ámbito de nombres.

  Un framework con funciones mínimas, como una API de base de datos y un motor de plantillas, requiere una cantidad considerable de código. Django por ejemplo consta de alrededor de 43000 líneas, sin contar las aplicaciones contribuidas (administración, GIS, sesiones, autenticación, bitácora) [1]. En particular, para migrar un framework implementado sobre Python, el sistema de paquetes es muy importante. En Python los módulos definen ámbitos de nombres de los cuales se pueden importar selectivamente sólo los símbolos usados, sin contaminar el ámbito local.

  Protopy implementa un sistema de ámbitos de nombres similar al de Python, integrado con el sistema de paquetes. Cada módulo posee un ámbito de nombres

[1] Métrica obtenida del comando `wc -l $(find django -iname "*.py" | grep -v contrib | grep -v backends)`

aislado. La publicación de símbolos de un paquete (funciones, constantes, clases) es explícito a diferencia de Python.

- Orientación a Objetos "Pythonica"

  En JavaScript, cada prototipo almacena la estructura estática de la "clase" y en el constructor se inicializa la instancia. En Python, la creación de la clase la realiza el método `__new__` y la inicialización, el método `__init__`. Python permite herencia múltiple y la definición dinámica de tipos a través de metaclases o mediante el builtin `type` (que sirve como factory). Es necesario para adaptar el código Python a JavaScript, definir los tipos base `object` y `type` debido a que piezas claves de Django como el ORM y el sistema de formularios basan su funcionamiento en estos builtins. Por ello en Protopy se implementaron un sistema de tipos o clases similar al de Python.

  En Python también se pueden definir métodos especiales en las clases, que permiten sobrecarga de operadores (`+` mediante `__add__`, `==` mediante `__eq__`, `&` y `^` mediante `__and__` y `__or__`, los paréntesis de invocación `()` mediante `__call__`, etc). Algunos de éstos se pueden emular en JavaScript y Protopy brinda facilidades para esto.

- Tipos de datos y `builtins`

  Existen ciertos tipos que no existen en JavaScript con la misma funcionalidad que en Python. Un caso puntual son los `Object` o arreglos asociativos comparados con el tipo de datos `dict`. En Protopy se implementaron los tipos de datos más comunes provistos por Python (por ejemplo, `Dict` y `Set`).

  En Python el conjunto de funciones, tipos disponibles en el ámbito de nombres global se conoce como `builtins`. Forman parte de este conjunto, `int`, `bool`, `str`, `list`, `tuple`, `map`, `filter`, `abs`, `all`, etc. La mayoría de estos símbolos fueron portados a Protopy y también publicados en el ámbito global.

- Selección de elementos mediante CSS

  DOM permite realizar búsquedas de elementos dentro de un documento de tres maneras: mediante un identificador, mediante un nombre de etiqueta, o

  > mediante el acceso jerárquico tipo árbol (estas técnicas se pueden combinar).

  Los selectores CSS simplifican la tarea de seleccionar un conjunto de elementos que cumplen con cierta condición. Utilizan la sintaxis de las hojas de estilo en cascada para determinar los elementos que están siendo seleccionados.

  Por ejemplo con la API de DOM la selección de los elementos del tipo `link`, que posean un atributo `href` se realiza de la siguiente manera:

```
var links = document.getElementsByTagName('a').
    filter( function (e) { return e.getAttribute('href'); });
```

Mientras que con selección CSS, esto se reduce a:

```
var links = $$('a[href]');
```

Las librerías YUI, Prototype, jQuery, Dojo y Ext JS entre otras poseen este tipo de selector. Su utilización libera al desarrollador de incompatibilidades o implementaciones pobres de DOM, a la hora de interactuar con los elementos del documento.

Peppy es una librería de JavaScript que implementa un selector CSS que fue incorporado dentro del núcleo de Protopy como selector de elementos del DOM. Las funciones se implentaron en `$("id")` para recuperar elementos por identificador y `$$("selector_css")` para recuperar un conjunto basado en un criterio CSS.

- Compatibilidad de Protopy

  JavaScript 1.7 sólo se encuentra disponible sobre la plataforma Mozilla (Firefox 3.0+), de modo que se eliminaron gran parte de los arreglos de compatibilidad presentes originalmente en Prototype y se portó a la nueva sintaxis muchas partes del código original.

  Por esta razón Protopy es únicamente compatible con Firefox actualmente.

En la siguiente figura se ilustran los diferentes componentes de Protopy:

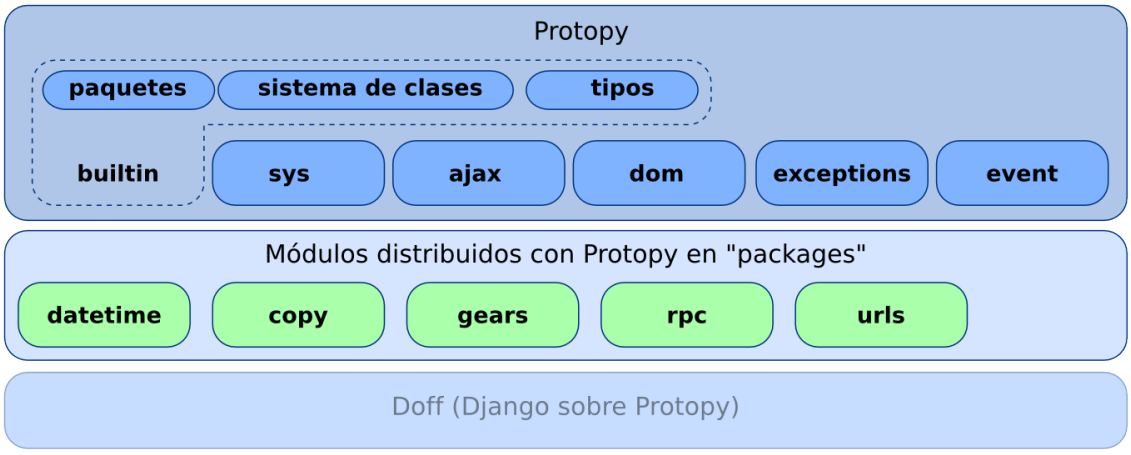

Figura 5.1: Composición de Protopy

5.2 Utilización de la Biblioteca

La librería Protopy consiste en un único recurso JavaScript llamado `protopy.js`. Su inclusión debe realizarse en la cabecera del documento, de la siguiente manera:

```
<script type="text/javascript;version=1.7" src="/ruta/a/protopy/protopy.js">
</script>
```

Dentro del atributo `src` se define lo que se denomina *ruta base* y consiste en la jerarquía de directorios *(/ruta/a/protopy/* en el ejemplo expuesto anterior).

Tambien mediante el atributo `src` de la etiqueta `script` de inclusión de Protopy, se pueden pasar argumentos en la URL. Por ejemplo:

```
src="/ruta/a/protopy/protopy.js?argumento=valor&argumento=valor"
```

Existen algunos argumentos utilizados por el framework desconectado que se analizarán en el capítulo siguiente.

5.3 Recursos y Módulos

Como se mencionó en el apartado teórico dedicado a JavaScript, la inclusión de código en un documento HTML se realiza mediante la etiqueta `script` definiendo en el atributo `src` la URL del recurso. Cuando el navegador encuentra estas etiquetas en el análisis del documento (o *parsing*), descarga el recurso y lo evalúa en el contexto de elemento `window` (es decir, si en el código se define una variable `a` fuera del bloque de una función, ésta pasa a ser miembro del objeto `window`).

La carga de JavaScript mediante etiquetas de inclusión es práctico para proyectos pequeños (donde se utiliza JavaScript para validación, enriquecimiento de formularios, accesibilidad) pero resulta limitado y poco mantenible en proyectos grandes (en los cuales la cantidad de JavaScript crece).

En Protopy se buscó la forma de solucionar este problema analizando la forma en la que se resuelve en el lenguaje Python, en donde la modularidad y la creación de ámbitos de nombres están íntimamente relacionados.

En vez de cargar los recursos mediante la inclusión de etiquetas, se tomó la idea de la carga asincrónica de código de Kris Kowal [KrisKowal09], también presente en YUI [YahooYUILoader09], donde se utiliza una función de inclusión que genera peticiones `XmlHTTPRequest` a los recursos y los evalúa cuando se completan. Se utilizó la implementación de Dojo quedando de manifiesto en los comandos de inclusión, por ejemplo, `require` en lugar de `import`.

Por ello, se agregó al ámbito global la función `require( "nombre_modulo" )` que recibe como parámetro una cadena con el nombre del módulo que se desea cargar y lo descarga asincrónicamente para luego evaluarlo en un contexto aislado.

En Protopy, un módulo es un recurso JavaScript que publica explícitamente una interface. De esta manera la funcionalidad se encuentra encapsulada. Para publicar un elemento del módulo se utiliza la función `publish({nombre: objeto, nombre: objeto})`.

Además de cargar el módulo, la función `require()` se encarga publicación en el ámbito local a la invocación de los símbolos que sean publicados mediante `publish()`.

..note:: EH?? Me parece que la oración anterior no está bien escrita!

Durante la evaluación se encuentra disponible la variable `__name__` que tiene como valor el nombre del módulo que se está evaluando.

Los módulos se pueden organizar jerarquicamente en directorios, los cuales se denomina paquetes. El comando `require("a.b")` carga el elemento `b` del paquete `a`, siendo `a` un directorio con un archivo `b.js` en el servidor.

Cuando se utilizan paquetes, a diferencia de Python, éstos no incluyen funcionalidad per se [2] . La única función de ellos es la de establecer una estructura.

5.3.1 Carga de Módulos

Cuando se utiliza la función `require` la búsqueda de los módulos se realiza en la *ruta base*. Esta se estableció como el subdirectorio `packages` a partir de la ruta desde la cual se cargó la librería Protopy. Por ejemplo, si `protopy.js` se encuentra en `http://dominio.com/media/js/protopy.js` la invocación `require("dom")` cargará el archivo `dom.js` desde `http://dominio.com/media/js/packages/dom.js`.

La búsqueda de módulos se puede ampliar más allá de la ruta base. En el módulo `sys` existe la variable `paths`, que consiste en una lista de rutas en las cuales se realiza la búsqueda cuando el módulo no se encuentra en la ruta base. El programador puede añadir entradas a `sys.path`.

La función `require` se puede usar para:

- Obtener un módulo como un objeto,

```
var mod = require('events');

require('events');  // Events queda en el espacio global
```

- Obtener un símbolo de un módulo

```
require('events', 'Event');
```

- Importar todas las definiciones del módulo en el espacio de nombres del llamador.

[2] En Python, un paquete es un directorio que tiene un módulo llamado `__init__.py`, donde se puede definir funcionalidad que es evaluada si se realiza la importación del paquete tal si fuese un módulo (Ej: `import paquete`)

```
    require('events', '*');
```

Una descripción detallada se encuentra en el apéndice dedicado a Protopy.

5.3.2 Publicación de Módulos

En Python no es necesario declarar de manera explícita que símbolos se exponen en un módulo, ya que todas las definiciones son públicas. Por ejemplo, si se define un módulo `utils.py` con el siguiente código:

```
def promedio(lista_de_enteros):
    return sum(lita_de_enteros) / float(len(lista_de_enteros))

def cantidad_palabras(linea):
    return len(linea.split())
```

el programador puede usar las sentencias `from utils import promedio` que incorpora la función `promedio` al ámbito de nombres local, `from utils import *` que incorpora todas las funciones de `utils` al ámbito de nombres o simplemente `import utils`, que incorpora el módulo, el cual tiene como miembros en este caso a `promedio` y a `cantidad_palabras`.

En cambio los módulos en Protopy se evalúan dentro de una clausura y los llamadores no podrán acceder a sus funciones si no son publicadas.

La función `publish` es la encargada de realizar la tarea de publicar el contenido del módulo.

A continuación se presenta un fragmento de código que ejemplifica la definición y publicación de las dos funciones mostradas en el ejemplo anterior:

```
1  function promedio(lista_de_enteros) {
2      var suma = 0;
3      for (int i = 0; i < lista_de_enteros.length; i++){
4          suma += lista_de_enteros[i]
5      }
6      return suma / lista_de_enteros.length;
7  }
8
9  function cantidad_palabras(linea) {
10     return linea.split(' ').length
11 }
12
13 publish({
14     promedio: promedio,
15     cantidad_palabras: cantidad_palabras
16 });
```

5.4 Módulos Nativos

Como mencionó en el capítulo anterior, es necesario que la implementación de Python que se utilice en el navegador como soporte para Django, posea algunos módulos de la API estándar de Python. Por esta razón se incorporaron los siguientes módulos a Protopy:

Nota: ¿Y todo esto cómo se relaciona con Protopy? Deberían decir que entonces se desarrollaron los siguientes módulos, etc.

Está demasiado escueto. Deberían describir un poquito más cada módulo.

- `builtin`

  Este módulo cuenta con los tipos y funciones disponibles en el ámbito global ni bien se inicia Protopy. Provee los siguientes símbolos:

  - `publish()`

    Publica el contenido de un módulo, recibe como argumento una arreglo asociativo. Como ya se mencionó, en Protopy el contenido de los módulos es privado por defecto.

    Por ejemplo, para publicar un elemento en un módulo se utiliza el siguiente código:

    ```
    var contador = 0; // Variable a nivel módulo

    function saludar() {
        // El operador subs sobre String opera de la misma
        // manera que el operador% en Python. Similar a printf de C.

        return "Hola mundo, invocado%d veces".subs(contador);
    }

    function cant_llamadas() {
        // Contador de llamadas sobre el código de
        var tmp = contador;
        contador++;
        return tmp;
    }

    // Publicación explícita
    publish({
        saludar: saludar
    });
    ```

  - `require(modulo, [simbolo])`

Carga mediante el sistema de paquetes un módulo, incroporandolo al ambito de nombres.

Los nombres de las funcinoes `publish`/`require` son tomadas de Dojo.

Por ejemplo, para cargar la función del módulo expuesto en el ejemplo de la función `publish()`:

```
>>> require('mi_modulo');
>>> mi_modulo.saludar()
"Hola mundo, invocado 0 veces"
>>> mi_modulo.saludar()
"Hola mundo, invocado 1 veces"
>>> mi_modulo.saludar()
"Hola mundo, invocado 2 veces"
```

- `$("id")` y ‘‘$$(“selector_css”)

  Son alias a los elementos del módulo `dom` descrito más abajo.

- `object`

  En Python el tipo base para todos los objetos es `object`. Sirve como base para el sistema de tipos Pythonico implementado en Protopy.

- `type(instancia)` ó `type(nombre, bases, [estatico], dinamico)`

  Este método posee dobe signatura. Cuando se lo invoca con una instancia como argumento, devuelve el tipo/clase de la instancia.

  La otra signautra sirve para crear tipos y debe ser invocada con la palabra reservada `new` de JavaScript. Recibe como argumentos el nombre del tipo, las calses base, opcionalmente un arreglo asociativo de atributos/métodos de clase y un arreglo asociativo con los atributos/métodos de instancia.

  Ambos casos responden al comportamiento del `builtin` homónimo de Python.

- `extend`

- `dom`

  En este módulo solo se publica el selector CSS Peppy de James Donaghue. Fue elegido por la velocidad y simplicidad del módulo. De la API [JamesDonaghuePeppyDocs] solo se publica `query()` y `cache()`.

  Por ejemplo sobre el siguiente código HTML:

```
<html>
    <head>
        <title>Uno</title>
    </head>
    <body>
        <div>
            <a href="/admin" class="lnk2">Admin</a>
            <a href="/ventas">Ventas</a>
            <a href="/otros" class="lnk2">Opciones</a>
        </div>
    </body>
</html>


// Carga del módulo
>>> require('dom');
// Utilizando el selector
>>> dom.query('.lkn2');
[a /admin/, a /otros/]
```

Se publican los símbolos `$("id")` y `$$("selector")` que referencian a esta función, de manera similar a la API de Prototype.

- `sys`

  Equivalente al módulo Python del mismo nombre. Maneja las rutas de cargas de módulos.

- `event`

  Manejo de eventos, implementación de Listeners y Publisher/Subscriber.

- `ajax`

  Envoltura de `XMLHttpRequest`, facilidades de interpretación de tipos de respuesta.

- `exceptions`

  Conjunto de excepciones.

- `timer`

  Envoltura de `window.setTimer()` y `window.setInterval()`.

5.5 Orientación a Objetos Basada en Clases

En el capítulo dedicado a las tecnologías del cliente se describió en el enfoque OO que posee JavaScript. Para lograr migrar muchos componentes de Django a JavaScript, se requiere un sistema

de objetos basado en clases. Muchos autores cuestionan el intento de imponer un sistema de clases sobre un lenguaje que ya posee su técnica para crear objetos, argumentando que no tiene sentido emular un paradigma dentro de otro.

Habiendo analizado la situación, se llegó a la conclusión que tanto para la presente tesina como para acercar a los programadores que utilizan Django al lenguaje JavaScript/Protopy era necesario proveer en Protopy un sistema de tipos similar al de Python. Prototype hace lo propio con el lenguaje Ruby.

La forma de crear nuevos "tipos de objetos" en Protopy es a través de la función `type`. Esta función no fue parte de la biblioteca hasta que no se observó la necesidad de otorgar mayor poder al constructor de clases que brindaba Prototype. La incorporación de type posibilitó nuevas formas de construir clases, similares a las que brinda la función homónima de Python, a la cual debe su nombre.

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

```
// Creación de un diccionario, que hereda del tipo object
var Dict = type('Dict', object, {
    ...
});
// Creación de una clase que hereda de Dict, observar que es una lista ya
// se permite herencia múltiple.
var SortedDict = type('SortedDict', [ Dict ], {
    __init__: function(object) {
        this.keyOrder = (object && isinstance(object, SortedDict))? \
            copy(object.keyOrder) : [];
        super(Dict, this).__init__(object);
    },
    // Iterador, retorna pares clave, valor
    __iter__: function() {
        for each (var key in this.keyOrder) {
            var value = this.get(key);
            var pair = [key, value];
            pair.key = key;
            pair.value = value;
            yield pair;
        }
    },
    // Método utilizado para la copia profunda
    __deepcopy__: function() {
        var obj = new SortedDict();
        for (var hash in this._key) {
            obj._key[hash] = deepcopy(this._key[hash]);
            obj._value[hash] = deepcopy(this._value[hash]);
        }
        obj.keyOrder = deepcopy(this.keyOrder);
```

```
31          return obj;
32      },
33      // Alias del método toString, sirve para representar en una cadena
34      // a la instancia
35      __str__: function() {
36          var n = len(this.keyOrder);
37          return "%s".times(n, ', ').subs(this.keyOrder);
38      },
39      // Setter
40      set: function(key, value) {
41          this.keyOrder.push(key);
42          return super(Dict, this).set(key, value);
43      },
44      unset: function(key) {
45          without(this.keyOrder, key);
46          return super(Dict, this).unset(key);
47      }
48  });
```

Este ejemplo presenta la definición del tipo `SortedDict` o diccionario ordenado, 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, seguida de un arreglo con los tipos base y de un arreglo asociativo con los atributos y métodos para los objetos de ese tipo.

Con los constructores así definidos es posible tener instancias que se comporten en función de sus respectivas definiciones:

Nota: Comentar el código

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

Como se observa en este ejemplo, 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` hace referencia al objeto instanciado con `new`.

Internamente `type` utiliza al objeto `prototype` para la construcción, con lo cual el operador `instanceof` presenta un comportamiento coherente. Pese a esto se recomienda usar la función `isinstance` disponible como `builtin` en Protopy, ya que ésta permite navegar por la cadena de herencia, como se muestra en el siguente ejemplo:

```
>>> d instanceof SortedDict
true
```

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

5.5.1 Inicialización

En el ejemplo de creación de una clase en Protopy se muestra la inicialización del tipo `SorteDict` usando una funcion `__init__`. Este método es llamado por el operador `new` inmediatamente después de crear una instancia. Actúa de una forma similar a un constructor del lenguaje Java, pero Python y Protopy permiten también la personalización de la instanciación implementando el método `__new__`. El constructor es el conjunto `__new__` e `__init__`.

Los métodos `__init__` deben llamar explícitamente al método `__init__` de la(s) clase(s) padre(s) de ser necesario.

5.5.2 Métodos Especiales

Protopy soporta la definición de algunos métodos especiales para los objetos, que serán invocados por la biblioteca en circunstancias particulares o cuando se use una sintaxis específica.

Nota: Está un tanto confuso el párrafo anterior. Los métodos son invocados por la misma librería ante determinadas circunstancias o se invocan directamente por el usuario?? Tampoco es clara la descripción de los métodos (finalidad o uso, quién o cuándo se invocan)

Estos métodos son:

- `__str__`

  Este método se utiliza cuando es necesario proveer de una representación en texto del objeto. Si bien JavaScript provee para esta tarea el método `toString`, por compatibilidad con Python y consistencia con la filosofía Python, se recomienda utilizar `__str__`. Ejemplo:

  ```
  >>> var Persona = new type('Persona', {
          // Constructor
          __init__: function (nombre) {
              this.nombre = nombre || "Sin nombre";
          },
          __str__: function (){
              return "Soy la persona de nombre: " +this.nombre;
          }
      });

  // Se crea una instancia con el nombre
  ```

```
>>> p = new Persona("diego");
// Se invoca la representación en cadena por cohersión
>>> "" + p
"Soy la persona de nombre: diego"
```

- `__iter__`

  Este método se utiliza cuando se requiere un iterador de la instancia sobre la cual es invocado.

  Protopy se vale de versiones modernas de JavaScript para brindar este método, que debe retornar un objeto que implemente el método `next()`. Ejemplo:

```
>>> var Iterable = new type('Iterable', [ object ], {

    __init__: function (cadena) { this.cadena = cadena; },
    __iter__: function () {

        return {
            next: function() {
                // Completar
            }
        };


    }

});

for (var elem in objeto_iterable) {
    print (elem)
}
```

- `__cmp__`

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

  Ejemplo

- `__len__`

  Este método es usado por la función `len`. La función incorporada `len` devuelve la longitud de un objeto. Funciona sobre cualquier objeto posea longitud (listas, diccionarios, etc.). Por ejemplo:

```
>>> len([1, 2, 4])
3
```

- `__copy__` y `__deepcopy__`

Se utilizan para copiar un objeto de manera superficial o profunda respectivamente.

- `__json__` y `__html__`

  Son métodos para serialización (o *marshalling*) del objeto en HTML o JSON respectivamente.

Protopy tiene otros métodos especiales, que generalmente están orientados a emular algún comportamiento de Python en JavaScript. Una descripción detallada se encuentra en el apéndice dedicado a Protopy.

5.5.3 Herencia

Como ya se mencionó, el código de un framework no debe ser modificado. El mecanismo de extensión en lenguajes OO es la especialización de componentes mediante herencia. Por esta razón se consideró fundamental dotar el constructor de tipos `type` con la capacidad de herencia similar a la de Python, ya lo que se desea migrar es un framework OO escrito en Python.

Cuando el desarrollador implemente funcionalidad basada en las características del framework desconectado, lo hará extendiendo alguna clase. Es por ello que Protopy utiliza la herencia del tipo **Prototype chaining**, aunque lo hace de una forma un poco más compleja con el objeto de soportar herencia múltiple.

Para implementar la herencia y en particular la herencia múltiple, el constructor de tipos recibe una lista de los tipos base e internamente crea un objeto que agrega de derecha a izquierda todos los métodos 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 función para determinar la correspondencia entre instancias y tipos llamada `isinstance`.

Otra función importada de Python es `issubclass`. Bajo algunas condiciones resulta útil determinar si un tipo es una sub-clase de otro y para ello esta función inspecciona la cadena de herencia.

Para acceder a las funciones de tipo base cuando se está redefiniendo un método Protopy provee la función `super`. De manera similar a la de Python, esta función determina el tipo de la instancia, y mediante éste accede al método buscado. Este comportamiento es el equivalente en JavaScript al de llamar al método del tipo base con la función `Function.apply` o `Function.call`.

Se debe destacar que de no especificar por lo menos un tipo base, Protopy establece por defecto a `object`, encargado de proveer los principales métodos (`__init__`, `__str__`).

5.5.4 Métodos y Atributos de Clase

Protopy contempla la definición de métodos y atributos de clase. Esta tarea se realiza agregando el conjunto de atributos y métodos de clase como un arreglo asociativo opcional que se antepone al que define la estructura de los de instancia.

El siguiente ejemplo define la clase "Planeta":

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

Esta clase tiene un atributo contador (`count`) que las instancias utilizan para numerarse en la construcción y un método `reset` para reiniciar el contador. Dentro de los métodos de clase la palabra reservada `this`, hace referencia a la clase.

A continuación se muestra cómo usar dicha clase.

```
// Se crea una instancia
>>> p = new Planeta('Tierra')
window.Planeta name=Tierra count=0 __name__=Planeta
// Se pone a cero al conteador mediante el método de clase reset()
>>> Planeta.reset()
// Se crea la estrella Sol
>>> sol = new Planeta('Sol')
window.Planeta name=Sol count=0 __name__=Planeta __module__=window
// Se crea el planeta Mercurio
>>> mercurio = new Planeta('Mercurio')
window.Planeta name=Mercurio count=1 __name__=Planeta
// Se crea el planeta Venus
>>> venus = new Planeta('Venus')
window.Planeta name=Venus count=2 __name__=Planeta
// Se crea el planeta Tierra
>>> tierra = new Planeta('Tierra')
window.Planeta name=Tierra count=3 __name__=Planeta
```

5.6 Objetos Nativos

Los objetos nativos de Protopy tienen como objetivo emular tipos de datos de Python o realizar una adaptación de sintaxis. Ellos son:

- `Dict`

  Si bien un arreglo asociativo nativo de JavaScript se comporta de manera similar a un diccionario de Python (`dict`), los diccionarios de Protopy permiten mejores formas de trabajar con la dupla clave-valor, posibilitando además el uso de objetos como claves en lugar de solo cadenas.

- `Set`

  Los Sets son listas de objetos en las cuales existe una única instancia de cada elemento como máximo. Permiten realizar operaciones de conjuntos como unión, intersección, diferencia, etc. Constituyen una adaptación del tipo `set` de Python.

- `Arguments`

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

  Este objeto tiene como objetivo brindar emapquetado de argumentos en JavaScript "Pythonico", una caractersística muy utilizada en el framework Django.

  Utilizando `Arguments` se puede convertir la siguiente sentencia:

  ```
  def funcion(arg1, arg2=None, arg3=None):
      print arg1, arg2, arg3
  ```

  al siguiente código JavaScript:

5.7 Manejo de DOM

Nota: Está parte está bastante confusa y escueta. Cuenten más y pongan métodos relevantes de ejemplo.

Va a ir en dos secciones distintas. Así que ya las rearmé, pero por favor hay que describirlas más y mejor.

Dentro del módulo `dom` de Protopy se provee un selector CSS que permite seleccionar elementos basados en selectores de nivel 3 de CSS [W3CCSelCSS309].

Los eventos están uniformados bajo un módulo de manejo de eventos (`events`), que permite conectar eventos DOM con funciones en JavaScript, así como también crear eventos de usuario.

Nota: Discutir

5.8 Agregados a JavaScript

Con la idea de extender el atributo `prototype` de los `HTMLElement` de la biblioteca de JavaScript Prototype, se agregaron funciones para modificar e incorporar elementos al documento, serialización (o *marshalling*) de formularios y obtención de valores, etc.

También se extendieron los tipos de datos propios de JavaScript, como el caso de las cadenas (`str` en Python) agregando métodos de sustitución de patrones, conversión de números, manejo de fecha, etc.

5.9 Envoltura de Google Gears

Como se mencionó en el capítulo anterior, para la migración del framework a un framework desconectado, son necesarios los componentes `DataBase` y `Local Server`.

Se decidió utilizar la API de Gears que provee estos componentes en el desarrollo de Protopy. Pero, a fin de mantener la filosofía "Pythonica" de Protopy se realizó un recubrimiento de esta API, de manera de brindar mayor uniformidad y consistencia en las APIs presentadas al programador de aplicaciones desconectables.

Cabe destacar que Protopy no depende de Gears para su funcionamiento. Durante la inicialización de la librería se detecta si el navegador tiene disponible la API de Gears. Si el desarrollador requiere almacenamiento persistente deberá asegurarse de que Gears esté instalado.

Dentro del módulo `sys`, se publica el objeto `gears` que almacena información sobre la disponibilidad de Gears, qué permisos le ha otorgado el usuario, qué versión del plugin está disponible, el factory de Gears, etc. Además provee un mecanismo para facilitar la instalación de Gears si no se encuentra instalado.

El método `create` del objeto `sys.gears` es el encargado de crear y retornar las instancias de los diversos módulos de Gears. Es un recubrimiento del factory original de Gears. Este método, tras la creación de la instancia del tipo solicitado, analiza si se encuentra alguna funcionalidad extra de recubrimiento y la aplica de estar disponible, devolviendo una instancia con funcionalidad aumentada.

Si bien no es necesario que el código de aplicación realizado por el programador acceda a Gears a través de `sys.gears`, se recomienda su utilización ya que simplifica y uniforma la interfase entre Gears y la aplicación.

Por otra parte, el desarrollo del framework implicó extender los siguientes componentes de Gears:

- `desktop`

  El componente `desktop` permite interactuar con el escritorio del cliente. Se extendió la creación de accesos directos para simplificar la generación de los mismos y agregar la posibilidad de manejar recursos del tipo `Icon` e `IconTheme`.

- `database`

  Sobre el objeto `DataBase` se agregó funcionalidad para uniformar el acceso a la base de datos por los módulos de Protopy y encapsular los `ResulSet` en cursores a los que se incorporó iteradores, registro de funciones para tipos de datos, etc.

5.10 Auditado de Código

Un obstáculo muy común con el cual se encuentran los desarrolladores a la hora de la codificación de JavaScript es la dificultad de depuración en el navegador.

Se suele recurrir a la función `alert()`, que genera una ventana emergente y modal, que detiene la ejecución de JavaScript hasta que se pulse el botón de aceptación. Se utiliza como punto de ruptura (*breakpoint*) en el código, pero resulta engorroso debido a que el programador debe interactuar activamente en la depuración, cerrando las ventanas tras la llegada a una sentencia `alert` y editando el código con cada ciclo de depuración, quitando caracteres de comentarios en el camino por el cual desea realizar la depuración.

Existe un plugin, llamado Firebug, que integra una consola de JavaScript, visualizador y analizador de DOM, CSS, peticiones de red y un depurador de JavaScript avanzado (breakpoints, visualización sobre variables, ruptura ante condición, etc).

Cuando Firebug se encuentra instalado y la consola activada, el desarrollador puede utilizar las funciones `console.log`, `consoel.info` y `console.warn` para depurar la aplicación imprimiendo cadenas y valores de variables, sin necesidad de utilizar `alert`.

Protopy agrega un sistema de *logging* sobre las funciones antes mencionadas de firebug, que permite la configuración de las salidas, el formateo y la prioridad, de manera similar a *log4j* (popular sistema de depuración de Java). Usando este sistema el desarrollador no necesita suprimir las sentencias de depuración del producto final, simplemente anula la salida en la configuración.

Una vez configurado el sistema de logging, los módulos que requieran auditar el código sólo deben requerir un *logger* (objeto encargado de enrutar mensajes de depuración) en su espacio de nombre e invocar a sus funciones.

Los loggers se agrupan jerárquicamente y toman los nombres de los módulos en los cuales se ejecutan. La configuración respeta esta jerarquía. Un logger de un submódulo adopta la configuración de su padre a menos que se defina algo particular para él.

En este ejemplo se requiere el módulo `logging` y posteriormente un logger para el módulo con el nombre `__name__`.

```
// Requerir el modulo 'logging.base'
var logging = require('logging.base');
// Crear un logger con el nombre del módulo actual, que se almacena
// en __name__
var logger = logging.get_logger(__name__);
```

```
var query = ... // carga con información de depuración
var params = ... // carga con información de depuración
// Envío de un mensaje al sistema de logging
logger.debug('La query: %s\n Los parámetros: %s', query, params, {});
```

Suponiendo que el módulo del ejemplo se llama `doff.db.models.sql` el siguiente archivo de configuración prepara el logger en modo `DEBUG` para auditar el código en la consola de Firebug y en una URL con distintos formatos.

```
{
    'loggers': {
        // Logger básico
        'root': {
            'level': 'DEBUG',   // Los mensajes con prioridad DEBUG
                                // o mayor se imprimirán
            'handlers': 'firebug' // La salida será por el manejador
                                // firebug, definido más abajo
        },
        'doff.db.models.sql': {
            'level': 'DEBUG',   // El módulo tiene prioridad DEBUG
            'handlers': [ 'firebug', 'remote'], // La salida se realizará
                                                // tanto por firebug como
                                                // de manera remota
            'propagate': true // Los mensajes se propagan al padre
        },
    },
    'handlers': {
        'firebug': {
            'class': 'FirebugHandler', // Salida por el plugin FireBug
            'level': 'DEBUG',   // Nivel (configuración de firebug)
            // Formato de la salida
            'formatter': '%(time)s %(name)s(%(levelname)s):\n%(message)s',
            // Argumentos extras
            'args': []
        },
        'remote': {
            // Otro handler, para auditoría remota
            'class': 'RemoteHandler',
            // Nivel
            'level': 'DEBUG',
            // Formato
            'formatter': '%(levelname)s:\n%(message)s',
            // Argumento extra, URL donde se envían los mensajes
            'args': ['/loggers/audit']
        },
        'alert': {
```

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

5.11 Interactuando con el Servidor

Protopy provee una interface de recubrimiento sobre las peticiones asincrónicas (AJAX) con el fin de simplificar la utilización de `XMLHttpRequest` y de trabajar con varias codificaciones de datos de manera segura (JSON).

Esta funcionalidad se encuentra en el módulo `ajax`. El objeto de transporte para AJAX es, por defecto, `XMLHttpRequest`. La forma de realizar una petición es creando una instancia del objeto `ajax.Request`.

```
// Carga del módulo
require('ajax');
// Request
new ajax.Request('una/url', {method: 'GET'});
```

Nota: RESCRIBIR

Nota: Hay que redactar un poco mejor lo del arreglo asociativo con sus parámetros opcionales. Está un poco confuso.

El primer argumento de `Request` es la URL a la cual se realizará la solicitud. El segundo parámetro es un arreglo asociativo con parámetros opcionales, como en este caso el método HTTP ("GET", "POST", "PUT", etc). En caso de no especificarse, el método es `POST`.

Por defecto `Request` se comporta de manera asincrónica, y se debe agregar al arreglo asociativo de opciones la función que se invocará cuando la petición se haya completado, como se muestra en el siguiente ejemplo:

```
new ajax.Request('una/url', {
    method:'get',
    // Función que se invoca en el caso de que la petición sea exitosa
    onSuccess: function(transport){
        var response = transport.responseText || "sin texto";
        alert("Success! \n\n" + response); },
    // Función que se invoca en el caso de que la petición no sea exitosa
    onFailure: function(){
```

```
        alert('Algo esta mal...'); }
});
```

A cada función manejadora se la invoca con un objeto que representa la respuesta obtenida y que está en relación con el evento capturado, como argumento.

Otras funciones manejadoras de evento que se pueden definir son:

- `onUninitialized`

  Se invoca inmediatamente después de la instanciación.

- `onLoading`

  Se invoca periodicamente mientras la petición se encuentra en curso.

- `onLoaded`

  Se invoca cuando el contenido de la respuesta ha sido cargado en su totalidad.

- `onInteractive`

  Se invoca periodicamente mientras la petición se encuentra en curso, pero no se invoca si se trata del final de la respuesta.

- `onComplete`

  Se invoca cuando se ha procesado la respuesta, trasformandola a un tipo de dato adecuado, en el caso de existir un `content-type` como JSON.

- `onException`

  Se invoca cuando se produce algún tipo de error.

Todos estos dependen del estado del objeto `XMLHttpRequest`.

Nota: Y el párrafo siguiente está más confuso. Me parece que deberían explicitar en forma bien clara cuáles son las opciones y para qué se usan

De igual manera que el resto de las opciones, es posible agregar parámetros a la petición, estos pueden ser pasados como un objeto arreglo asociativo o como una cadena clave-valor. Estos parámetros pasan a ser parte de la petición HTTP.

5.12 Soporte para JSON

JSON fue el formato elegido para la transmisión de datos entre el framework desconectado en el navegador y su contra parte en el servidor. Estos datos comprenden información sobre medios estáticos para el almacenamiento de recursos locales, datos de modelos y llamadas a procedimientos remotos.

El manejo de JSON en Protopy se realizó en el módulo `json`. Este módulo posee la capacidad de serializar los objetos nativos de JavaScript así como también las instancias de clases que implementen el método `__json__`.

No se implementó sobre Protopy soporte para serialización XML debido a su complejidad y a que no brindaba ventajas significativas ante JSON para los objetivos de la presente tesina. Sin embargo se puede implementar esta característica en futuras versiones.

5.13 RPC (Remote Procedure Call)

RPC consiste en la ejecución de un procedimiento (función) de manera remota. El cliente envía el nombre y los argumentos de la función que solicita. El servidor ejecuta la función requerida y devuelve los resultados al cliente. Tanto los parámetros como los resultados requieren de una codificación preestablecida para ambas partes.

RPC brinda un nivel de abstracción sobre mecanismos más primitivos de comunicación como sockets, en bajo nivel, o peticiones HTTP.

Existen diversos estándares de RPC como ONC RPC de Sun (RFC 1057), RPC de OSF denominado DCE/RPC y Modelo de Objetos de Componentes Distribuidos de Microsoft DCOM, cada uno de los cuales define una codificación y un protocolo específico. La mayoría de ellos utilizan un lenguaje de descripción de interfaz (IDL) que define los métodos o funciones que publica el servidor, así como también los tipos de datos que transporta.

XML-RPC es un estándar que utiliza XML como lenguaje de comunicación y HTTP como protocolo de transporte. Es considerado el más simple de los mecanismos para publicación de servicios web y se incorporó a la librería Protopy como mecanismo para comunicación entre el framework desconectado y la aplicación en línea.

CAPÍTULO 6

Doff

En capítulos anteriores se mencionó que la creación de un framework generalmente surge de la identificación de objetos reusables en el desarrollo de software. Posteriormente los objetos identificados constituyen componentes que forman parte de una arquitectura. A éstos se accede mediante una API específica y se añade o modifica sus funcionalidades mediante configuración o extensión.

En el desarrollo del framework desconectado de esta tesina, si bien se contaba con Django como base, se detectó que muchos aspectos no estaban claramente definidos y algunos componentes carecían de sentido en el contexto del navegador. Por esta razón se realizaron dos proyectos que sirvieron para identificar las piezas necesarias del framework y mejorar el enfoque de Protopy, además de permitir crear nuevos componentes reusables que no están presentes en Django [1].

El framework desconectado, de manera similar a Protopy, se denominó Doff en base a su padre y a la tarea que cumple, tomando la *d* de Django y *off* de offline (desconectado). Su nombre significa, por lo tanto, Django Desconectado:

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

En este capítulo se realiza un análisis de los elementos migrados y las modificaciones que se definieron para ejecutar un proyecto en un ambiente desconectado. En el capítulo siguiente se presenta el trabajo realizado para lograr la interacción con un proyecto existente.

6.1 Migración de Componentes Básicos

Doff se inició como un conjunto de módulos para Protopy. Estos módulos implementaban diferentes componentes de Django sobre el cliente. Debido a que los módulos de JavaScript son recursos estáticos, se publicaron mediante un servidor de archivos muy simple llamado Aspen, que no requería intervención de Django [AspenWebServer09].

[1] Inicialmente se creó una aplicación de *blog*, debido a la popularidad de este tipo de aplicaciones, y luego "salesman" (o agente de ventas viajante) para explotar las posibilidades de contar con una aplicación web desconectada.

Como se mencionó en el capítulo introductorio, los componentes que requerían una migración directa a JavaScript eran el ORM (`django.db.models`) y el sistema de plantillas (`django.templates`). Las vistas y el sistema de URLs deberían ser adaptadas para el contexto del navegador.

El primer componente de Django migrado fue `django.db.models`, el cual se realizó simplificando algunos aspectos, como el soporte para múltiples bases de datos ya que Gears sólo provee SQLite. La tarea de migración ayudó a perfeccionar el sistema de tipos (clases) y el de módulos. El paquete resultante fue `doff.db.models` y, gracias a implementar progresivamente la misma API que Django, fue posible realizar el mismo manejo de datos en el cliente y en el servidor.

Las pruebas y depuración del proyecto se realizaron sobre Firebug, cargando Protopy en un archivo estático y realizando los `require()` correspondientes en la consola. Por ejemplo:

```
// En la consola de Firebug
>>> require('doff.db.models.base');
```

El siguiente componente en ser analizado y migrado fue el sistema de plantillas (`django.templates`). Si bien la librería de JavaScript Dojo provee un sistema de manejo de plantillas basado en la sintaxis de Django [DojoLibDjangoTmpl09], se optó por portar directamente la implementación de Django ya que Dojo no está basado en el sistema de módulos de Protopy. Esta funcionalidad se situó en el módulo `doff.templates`.

6.2 Definición de Proyecto

Luego de haber alcanzado un nivel de funcionalidad básico en `doff.db.models` y `doff.templates` se definió la interacción básica con Django, mediante la cual un proyecto en línea sería capaz de transferirse al cliente. Para este fin se creó una aplicación Django llamada `offline` cuyo objetivo inicial fue publicar estáticamente el código JavaScript de Protopy y Doff.

El siguiente paso fue definir un proyecto Django desconectado, es decir, un proyecto Doff. Se buscó el mayor grado de similitud posible con Django, donde un proyecto es un paquete Python (con los módulos `settings.py`, `manage.py` y `urls.py`) y cada aplicación un paquete dentro de la proyecto o localizable en el `PYTHONPATH` (una aplicación cuenta con los módulos `models.py` y `views.py`).

En Doff, un proyecto se definió como un paquete Protopy con un módulo `settings.js` con la configuración del proyecto. El módulo de proyecto Django `manage.py` carece de sentido en el navegador, ya que no se cuenta con una interfase de comandos en JavaScript, por lo que se implementó la clase `doff.core.project.Project`. Esta clase define una serie de métodos para interactuar con el proyecto que cumplen algunas de las funciones de el módulo `manange.py` y realizar tareas necesarias en el ambiente desconectado. El módulo `urls.js` se reservó para análisis posterior.

La clase `Project` posee los siguientes métodos:

- `bootstrap()`

  Se encarga de iniciar el manejo de URLs por parte del proyecto, de mostrar la página inicial y ceder el control

  de la navegación al framework desconectado.

- `install()`

  Se encarga de persistir el framework y el proyecto en el almacenamiento local del cliente. También crea la base de datos, tarea que en la aplicación en línea se realiza mediante la invocación de la orden `manage.py syncdb`.

- `uninstall()`

  Remueve la base de datos, el proyecto y el código del framework del cliente.

Un proyecto se crea de la siguiente manera:

```
<script type="text/javascript;version=1.7">
    require('doff.core.project', 'new_project');

    var agentes = new_project('agentes', '/off/agentes');
    agentes.bootstrap();
</script>
```

El primer argumento es el nombre del proyecto y el segundo es la URL que se agrega al `sys.path` de Protopy, definiendo de esta manera un nuevo paquete en Protopy en el ámbito de nombres global.

Las aplicaciones de un proyecto Doff se definieron como paquetes de Protopy que se encuentran dentro del proyecto. Constan de los módulos `views.js` y `models.js`.

Los siguientes apartados tratan sobre las implementaciones de la API del ORM, plantillas, vistas y formularios. Además se hace un análisis de una tarea fundamental del `Project` que es emular HTTP.

6.3 Modelos

Doff implementa la misma API que Django de definición y consultas, realizando adaptaciones a la sintaxis de JavaScript y a la librería Protopy.

Cada modelo es representado por un clase de Protopy que extiende de `doff.db.models.base.Model`.

A continuación se ejemplifica la implementación en el cliente del módulo `models.js` de una aplicación que tiene las entidades libro, autor y editor:

```
// Cargar el módulo del ORM
var models = require('doff.db.models.base');

// La clase editor (Publisher) extiende de models.Model y por lo tanto
// se crea como tabla en la base de datos
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()
});

// La clase autor (Author) extiende de models.Model y por lo tanto
// se crea como tabla en la base de datos
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' })
});

// La clase libro (Book) extiende de models.Model y por lo tanto
// se crea como tabla en la base de datos
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 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, al tipo de columna. 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 de una clase por tabla es el caso de las relaciones muchos a muchos. Por ejemplo, si un libro tiene uno o más autores y un autor es autor de uno o más libros entonces, en la base de datos, existe una tabla adicional para manejar esta relación. En el modelo `Book` tiene un `ManyToManyField` llamado `authors`, que no existe como columna en la tabla de la base de datos, para representar este situación.

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 autoincremental llamado `id`.

Para activar los modelos en el proyecto, la aplicación que los contiene debe estar incluida en la lista de aplicaciones instaladas de Doff. En consonancia con Django, esta configuración se realiza en el módulo `settings.js`, en la variable de configuración `INSTALLED_APPS`. Suponiendo que este archivo se encuentre en `bookstore/models.js`, el nombre de la aplicación sería `bookstore`.

Cuando el usuario instala el proyecto para la operación desconectada 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.

Una vez que se crea el modelo, Doff provee automáticamente una API JavaScript de alto nivel, muy similar a la de Django, para trabajar con estos modelos.

Además de implementar la API de consultas, el desarrollador puede realizar consultas en SQL sobre la base de datos del cliente mediante las herramientas de desarrollador provistas por Doff.

6.4 Inserción y Modificación de Datos

En el siguiente ejemplo se muestra la forma de realizar una inserción de una fila en la base de datos:

```
>>> 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/'});
// Guardado de la instancia
>>> p1.save();
```

Primero se crea una instancia del modelo pasando argumentos nombrados y luego se llama al método `save()` de la instancia. Esto genera internamente la siguiente sentencia SQL:

```
INSERT INTO books_publisher
(name , address, city, state_province, country,
```

```
website)
VALUES
('Addison-Wesley', '75 Arlington Street', 'Boston', 'MA', 'U.S.A.',
'http://www.apress.com/');
```

En el caso de `Publisher` se usa una clave primaria auto incremental `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 la sentencia SQL `UPDATE` en lugar de un `INSERT`).

Por ejemplo

```
>>> p1.name = "Pearson Education"
>>> p1.save()
```

se traduce en la siguiente secuencia SQL, asumiendo que el `id` asignado sea 1:

```
UPDATE books_publisher
SET name = "Pearson Education"
WHERE id = 1;
```

6.5 Recuperación de Datos

La forma de seleccionar y filtrar los datos es la utilización de `Managers` o administradores de consultas.

Todos los modelos automáticamente obtienen un administrador `objects` que debe ser usado cada vez que se quiera consultar sobre una instancia del modelo. El método `all()` es un método del administrador `objects` que retorna todas las filas de la base de datos. Al igual que en Django, el valor retornado es un `QuerySet`. Por ejemplo:

```
// Recuperación mediante el Manager
>>> publisher_list = Publisher.objects.all();
// Mostrar el resultado de la consulta
>>> print(array(publisher_list));
[<Publisher: Publisher object>]
```

En el ejemplo la línea `Publisher.objects.all()` solicita 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;
```

En la última línea del ejemplo anterior, se ve que en el listado no aparece una representación adecuada de la instancia de Editor. Esto responde a que en la definición del modelo no se incluyó un método de pasaje a cadena, o `__str__`. Se puede implementar este método para mejorar la depuración.

En el caso de requerir una fila o conjunto en particular, se utiliza el método `filter`. Por ejemplo:

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

`filter()` toma argumentos del tipo arreglo asociativo que son traducidos en las cláusulas SQL `WHERE` apropiadas, concatenando con el operador `AND` en el caso de tener más de un parámetro. El ejemplo anterior se traduce traducido en la siguiente consulta:

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

Las claves de los parámetros del arreglo asociativo de `filter` pueden contener subargumentos separados por dobles guiones bajos, por ejemplo:

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

que busca los editores cuyo nombre contenga la palabra “press”; lo que se traduce en la sentencia SQL:

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

Existen otros tipos de subargumentos de búsqueda, incluyendo `icontains` (`LIKE` no sensible a diferencias de mayúsculas/minúsculas), `startswith` (busca al comienzo), y `endswith` (busca al final) y `range` (consultas SQL `BETWEEN`). En el apéndice sobre Doff se describen en detalle todos esos tipos de búsqueda.

Cuando se requiere obtener un único objeto dada una condición, los administradores de consultas brindan el método `get()`:

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

En lugar de un `QuerySet`, este método retorna un objeto individual. Debido a eso, una consulta cuyo resultado sean múltiples objetos o ningún objeto causará una excepción.

Generalmente se necesita que los resultados de los `QuerySet` tengan un orden determinado por algún campo. Para esto se dispone del método `order_by()`. Por ejemplo, para un listado de editores ordenados por el campo nombre se debe hacer:

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

El ejemplo anterior es similar a la invocación de `all()`, pero la consulta SQL incluye un parámetro extra:

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

El ordenamiento puede realizarse por cualquier campo ordenable (cadenas, enteros, fechas, etc.) y además por múltiples campos. Si se antepone al nombre del campo un guión, el ordenamiento es inverso, por ejemplo:

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

Cuando el ordenamiento es frecuente, puede incluirse por defecto en la definición del modelo mediante la clase interna `Meta`:

```
var Publisher = type('Publisher', [ models.Model ], {
    name: new models.CharField({ maxlength: 30 }),
    //...

    // Clase interna
    Meta: {'ordering': ['name']}
});
```

Las búsquedas y ordenamientos, por ser métodos que devuelven `QuerySet`, se pueden encadenar, como se muestra a continuación:

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

Cuando se requiere acceder a un registro en particular, se puede usar la siguiente sintaxis:

```
>>> Publisher.objects.all().get(0) // Primer elemento del QuerySet
<Publisher: Addison-Wesley>
```

Esto se traduce en la siguiente sentencia SQL:

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

También se puede utilizar la sintaxis de rebanadas de JavaScript `slice(inicio, [fin])`, por ejemplo:

```
// Retrna a partir del primero
>>> Publisher.objects.all().slice(1)
[<Publisher: Apress Publishing>, <Publisher: O'Reilly>]
// Retorna entre las posiciones 1 y 3
>>> Publisher.objects.all().slice(1,3)
[<Publisher: Apress Publishing>]
```

que genera el siguiente SQL respectivamente:

```
SELECT
    id, name, address, city, state_province, country, website
FROM book_publisher
OFFSET 1;

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

6.6 Eliminación de Datos

Para eliminar objetos, simplemente se debe llamar al método `delete()` sobre una instancia:

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

que se traduce en el siguiente SQL:

```
DELETE FROM books_publisher
WHERE name = 'Adison-Wesley';
```

Los QuerySet pueden recibir también el método `delete()` y esto se traduce en la eliminación de todas las filas afectadas por el `QuerySet`. Por ejemplo:

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

que se traduce en el siguiente SQL:

```
DELETE FROM books_publisher;
```

6.7 Plantillas

Doff brinda soporte al sistema de plantillas de Django. Las plantillas escritas para una aplicación en línea son reutilizables sin modificaciones en la aplicación desconectada.

Existen, sin embargo, algunos factores a tener en cuenta a la hora de realizar plantillas para aplicaciones desconectadas. Cuando una aplicación se ejecuta de manera desconectada, el navegador sólo realiza una carga de documento y la navegación se basa en manipulaciones de DOM. Cada vez que se carga una página, en realidad se suprime la anterior y se anexa la nueva en los elementos `fake-body` y `fake-header`. Este trabajo es realizado por Doff, con ayuda del paquete `dom` de Protopy.

A continuación se realiza un breve análisis de la API de plantillas y su utilización en Doff.

6.7.1 Creación

Doff implementa la clase `Template` para crear plantillas, que se encuentra en el módulo `doff.template.base`. El argumento para la instanciación del objeto es el texto en crudo de la plantilla. En el siguiente ejemplo `t` es un objeto `Template` listo para ser procesado:

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

En caso de encontrar errores en el análisis de la plantilla, la instanciación de `Template` falla. Los errores contemplados 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`.

6.7.2 Contexto

Para obtener la salida procesada de una instancia de `Template`, es necesario proveer un *contexto*. Un contexto es simplemente un conjunto de variables y sus valores asociados. Una objeto plantilla usa las variables para rellenar la plantilla evaluando las etiquetas.

El contexto está representado en el tipo `Context`, el cual se encuentra en el módulo `doff.template.base`. La construcción del objeto toma como argumento opcional un arreglo asociativo. La llamada al método `render()` de la instancia de `Template` con el contexto como argumento "rellena" la plantilla, por ejemplo:

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

El objeto `Template` puede ser procesado 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 éste muchas veces:

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

# Código eficiente
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 `Context` puede contener variables más complejas. La forma de inspeccionar estas es con el operador . (punto). Usando el punto se puede acceder a objetos, atributos, índices, o métodos de un objeto.

En estos casos cuando el sistema de plantillas encuentra un punto en una variable el orden de búsqueda es el siguiente:

- Arreglo asociativo (por ej. `foo["bar"]`)

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

Por ejemplo, ante la siguiente plantilla

```
{{ x.elemento }}
```

al momento de ser procesada, inicialmente se comprobará si `x` es un arreglo asociativo y se reemplazará por el contenido de `x["elemento"]`, si no existe la clave o no se trata de un arreglo asociativo, se intentará obtener el atributo, simplemente `x.elemento`. Si en la búqueda como atributo, se encuentra un método, se lo ejecutará y se utilizará su salida para el remplazo. En el caso de no poder realizar el reemplazo por ninguna de las anteriores, buscará `elemento` como índice de arreglos. Tiene sentido cuando lo que se encuentra tras el punto es un numero.

Si una variable no existe en el contexto, el sistema de plantillas la procesa como una cadena vacía. Es posible cambiar este comportamiento modificando el valor de la variable de configuración `TEMPLATE_STRING_IF_INVALID` en el módulo `settings.js`.

6.7.3 Cargador de Plantillas

Si bien en los ejemplos anteriores se mostró la API de plantillas con cadenas como argumentos, las plantillas puden estar almacenadas en archivos. Para Doff estos son recursos estáticos y existe un sistema para la carga similar al de Django. Para tal fin en el módulo `settings.js` se establece el valor `TEMPLATE_URL`.

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

```
// Importar las clases Template y Context
require('doff.template.base', 'Template', 'Context');
// Importar la clase HttpResponse
require('doff.utils.http', 'HttpResponse');
// Importar la clase get_template
require('doff.template.loader', 'get_template');

// Definición de una vista
function current_datetime(request) {
    // Cargar la vista
    var t = get_template('mytemplate.html');
    // Rednderización sobre la variable html con el contexto con una
    // fecha
    html = t.render(new Context({'current_date': new Date()}))
    // Retornar la respuesta en una respuesta http
    return new HttpResponse(html);
}
```

En esta vista se utiliza la API para cargar plantillas, a la cual se accede mediante la función *get_template()*.

Existen varios cargadores de plantillas que se pueden habilitar en el archivo de configuración. En el presente trabajo se utiliza sólo el de carga de plantillas a través de URLs. Por defecto, `TEMPLATE_URL` es una cadena vacía, en cuyo caso la búsqueda de la plantilla se realizará en `templates/` a partir de donde se instanció el proyecto.

6.8 Emulación de HTTP

En una aplicación en línea, con cada click sobre un enlace, envío de formulario o solicitud asincrónica (AJAX) se realiza un requerimiento al servidor, es decir se traduce en una solicitud HTTP.

Cuando la aplicación se encuentra desconectada, es decir, se ha instanciado el proyecto, Doff utiliza dos clases para emular HTTP y el comportamiento predefinido del navegador. Estas clases son: `DOMAdapter` y `LocalHandler`.

`DOMAdapter` se encarga de crear un documento falso en el DOM, un historial y administrar la interacción con el usuario, generando instancias de `Request` con los eventos de los enlaces y formularios, que son enviadas al `LocalHandler` como `Request` emulando peticiones HTTP.

Cuando se recibe una petición en Django, ésta es procesada por una instancia de la clase `Handler`. El equivalente en el proyecto desconectado es `LocalHandler`, que se encarga de procesar las peticiones que envía el `DOMAdapter`.

El `LocalHandler` se encarga de realizar el pasaje del `Request` por los diferentes Middlewares, como indica la figura 2.6 del apartado de tecnologías del servidor. En este proceso se llama a la vista.

La salida de la vista se envuelve en un objeto `Response`, equivalente a la respuesta HTTP, que es devuelto al `DOMAdapter`.

Cuando el `DOMAdapter` recibe la respuesta, realiza las tareas de actualización de la fachada del navegador: actualizar el documento falso, generar una entrada en el historial y conectarse a los eventos que puedan generar los links y formularios.

Con cada entrada en el historial se modifica el enlace (*hash*) de la URL. La instancia de `History` captura, durante la instanciación del proyecto desconectado, la URL raíz. La navegación en el proyecto desconectado se realiza en base a la URL raíz. Por ejemplo, si la URL en la que se instancia el proyecto es:

```
http://mi_dominio.com/base_offline/base
```

y se accede en la aplicación en línea a la URL:

```
http://mi_dominio.com/base_offline/base/ventas
```

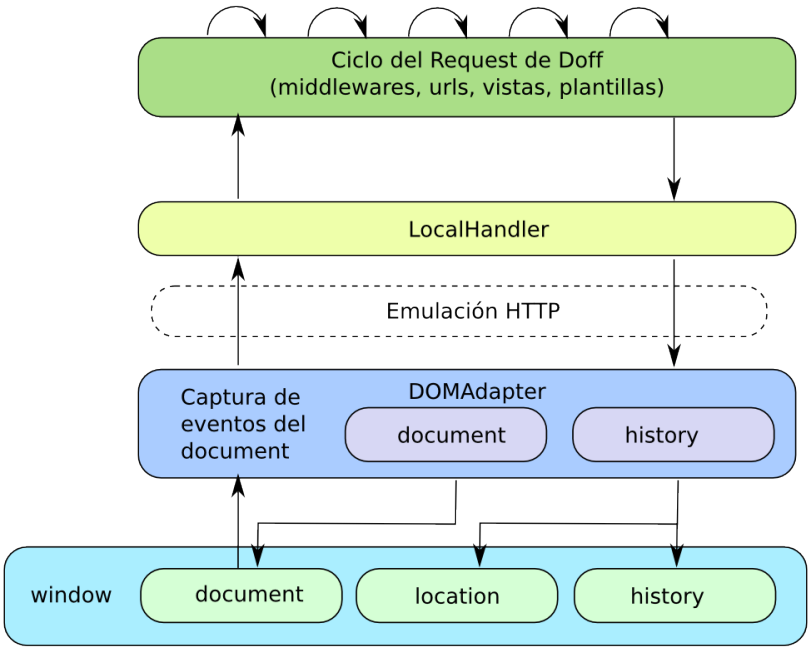

Figura 6.1: Diagrama de comunicación entre el documento y el sistema de emulación HTTP de `DOMAapter` y su comunicación con el `LocalHandler`.

el objeto `History` generará la URL:

> http://mi_dominio.com/base_offline/base/#ventas

Los módulos antes mencionados se encuentran conectados entre sí mediante el módulo de eventos `event` provisto por Protopy.

Debido a que la operación con el DOM se encuentra recubierta en la operación desconectada, el código de las plantillas debe ser adaptado para trabajar con la API de Doff/Protopy. Se debe:

- Utilizar `sys.window` en vez de elemento `window`. Éste apunta a `window` cuando el template se procesa con conexión, y a `DOMAdapter` cuando se encuentra sin conexión.
- Utilizar `sys.transport` en vez de `XMLHttpRequest` para que las llamadas asincrónicas sean correctamente enrutadas. Éste apunta a `XMLHttpRequest` cuando la aplicación se encuentra en línea y a una emulación que trabaja con el `LocalHandler` en modo desconectado.
- Utilizar el selector CSS `$$("selector")` para la selección de elementos que trabajan adecuadamente en el contexto desconectado y en línea. Como es un `builtin` está siempre disponible. Para el caso de la selección por ID se tiene `$("id")` (similar a Prototype).

```
var mi_boton = $('mi_boton'); // en vez de
               // sys.window.document.getElementById('mi_boton');
```

```
var mis_links = $$('a.mis_links');
```

- Utilizar `event.connect( nombre_evento, emisor, receptor, [método])` en vez de `addListener` sobre los `HTMLElements` u otros elementos del DOM. Por ejemplo:

```
require('event');
event.connect('click', $('boton'), function() {
    alert("Clickeaste mi boton");
});
```

Una operación muy común es ejecutar algo ante el evento de carga de página. Esto se realiza de la siguiente manera:

```
require('event');
event.connect('load', sys.window, miFuncionListener);
```

- Evaluar el valor del contexto `offline` si existiese parte del template que sólo debe mostrarse en un estado de la aplicación.

```
{% if offline %}
    <h1>Estoy Offline!</h1>
{% else %}
    <h1>Estoy en linea</h1>
{% endif %}
```

Como se ha descripto, Doff modifica el comportamiento de varios módulos de Protopy para lograr consistencia en el entorno desconectado.

6.9 Vistas

Las vistas en Doff son funciones que contienen la lógica de la aplicación. Están asociadas a URLs en el archivo `urls.js` (siempre que no se haya modificado el `ROOT_URLCONF`) del proyecto desconectado. Al igual que en Django, la variable que almacena estas asociaciones es `urlpatterns`. En el mapeo se pueden hacer inclusiones de módulos de URLs definidos en las aplicaciones.

Un módulo de URLs se conforma de la siguiente manera:

```
1 require('doff.conf.urls', '*');
2 // El proyecto se llama Blog y la aplicación Post, por lo que las vistas
3 // se encuentran en el submódulo blog.post.views
4 require('blog.post.views');
5
```

```
6  // Definición de los patrones
7  var urlpatterns = patterns('',
8      // Asociación por el nombre completo de la vista
9      ['^/$', 'blog.post.index'],
10     ['^add_tag/$', 'blog.post.views.add_tag'],
11     ['^remove_tag/([A-Za-z0-9-]+)/$', 'blog.post.views.remove_tag'],
12     // Asociación mediante la referencia de la función del módulo importado
13     ['^add_post/$', views.add_post],
14     ['^remove_post/([A-Za-z0-9-]+)/$', views.remove_post]
15 );
16
17 // Lo único que se publica es la variable urlpatterns
18 publish({
19     urlpatterns: urlpatterns
20 });
```

Es importante notar que las expresiones regulares nativas de JavaScript no poseen la capacidad de recuperación de grupos nombrados, sólo recuperación posicional como se puede apreciar en las líneas de 11 y 14 del ejemplo anterior.

Una vista se conforma de manera muy similar a Django:

```
require('doff.utils.shortcuts', 'render_to_response', 'redirect');

function remove_tag(request, slug){
    var tag = Tag.objects.get({'slug': slug});
    tag.delete();
    return redirect('/');
}

function index(request){
    return render_to_response('mi_template.html', {
        titulo: "Bienvenido a la aplicación desconectada"
    });
}

// ... más vistas

publish({
    remove_tag: remove_tag,
    index: index,
    // ... mas publicaciones
});
```

Las vistas reciben el objeto `HTTPRequest` como primer argumento, en consonancia con Django. En la aplicación sin conexión, la creación del ciclo del `Request` comienza con la captura del evento por parte del `DOMAdapter`, como se describe en la sección anterior.

Las vistas deben devolver un objeto `HTTPResponse`. Doff implementa algunos atajos de `django.shortcuts`, por ejemplo, `render_to_response(nombre_template, contexto)` para simplificar el prceso de construcción e interpretación de una plantilla y generación de la instancia del `HTTPResponse`.

6.10 Formularios

Los formularios son casi por excelencia el mecanismo de entrada para las operaciones CRUD en los sistemas de información basados en la web.

Doff posee la misma abstracción de objetos que brinda Django para facilitar la manipulación y validación de datos mediante instancias del tipo `Form`.

Un formulario en Doff es una clase que extiende de `Form` y tiene la responsabilidad de validar la entrada de datos y generar salida HTML.

Los campos del formulario se definen como atributos de la clase `Form`, y extienden de `doff.forms.fields.base.Field` o alguna de sus subclases.

Un campo tiene como atributo, de manera implícita, un `Widget` que se encarga de la representación en HTML del campo particular. Por ejemplo, un formulario para un autor (del ejemplo expuesto en la sección sobre modelos) se define de la siguiente manera:

```
var forms = require('doff.forms.base');

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

Las instancias de `Form` tienen los métodos `as_ul()` (genera salida en forma de lista) y `as_table()` (genera salida en forma de tabla) que generan la salida HTML del formulario y son utilizados en las plantillas. Por ejemplo:

```
<html>
    <head>
        <title>Template para formulario</title>
    </head>
    <body>
        <!-- Definición del formulario -->
        <form action="" method="POST">
            <ul>
                <!-- Si el formulario  fue enviado en el contexto del
```

```
                template con el nombre "form", se puede invocar el método
                que lo muestra como items de una lista -->
                {{ form.as_ul }}
            </ul>
            <!-- Botón de enviar -->
            <input type="submit" value="Enviar">
        </form>
    </body>
</html>
```

De esta manera se muestra el formulario en la página.

Por regla general, los formularios se encuentran en un módulo `forms.js`, aunque pueden definirse en tiempo de ejecución para necesidades puntuales.

Las vistas que manejan formularios suelen valerse del `request.method` para evaluar si el formulario está siendo requerido o enviado para su validación. Por ejemplo, la siguiente vista hace uso del atributo mencionado:

```
1  // Funciones de ayuda
2  require('doff.shortcuts', 'redirect', 'render_to_response');
3  // Importar el formulario
4  require('bookstore.core.forms', 'AuthorForm');
5  // Importar el modelo
6  require('bookstore.core.models', 'Author');
7
8  function crear_autor(request) {
9      // El formulario es para validación
10     if (request.method == "POST") {
11         var form = new AuthorForm({ data: request.POST });
12         // Los datos ingresados son válidos?
13         if (form.is_valid()) {
14             // Se crea el autor
15             var autor = new Author({data: form.data});
16             autor.save();
17             // Se redirecciona a una página sobre el autor
18             return redirect('/autores/%d'.subs(autor.id));
19         }
20
21         // Si el formulario no es válido, cuando se muestre
22         // en el template, se marcarán los errores de validación
23     } else {
24         // Se crea el formulario vacío
25         var form = new AuthorForm();
26     }
27
28     return render_to_response("mi_formulario", {
```

```
29          form: form
30      });
31
32  }
```

Una instancia de formulario puede estar en uno de dos estados: *bound* (vinculado) o *unbound* (no vinculado). Una instancia vinculada se construye con un arreglo asociativo (como en la línea 11 del ejemplo anterior) y posee la capacidad de validar y volver a representar los datos con los cuales fue construido. Un formulario desvinculado no tiene datos asociados y simplemente tiene la utilidad de representarse en HTML.

6.10.1 Validación de Datos

Un formulario vinculado, tiene la responsabilidad de validar los datos con los cuales ha sido instanciado.

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

```
form = new ContactForm({ data: request.POST })
if (form.is_valid()):
    # Hace algo con los datos del formulario
```

Para acceder a los datos se accede directamente al `request.POST`, pero de esta manera no se saca provecho de la conversión que realiza Doff, por ejemplo, para el caso de las fechas. En cambio se debería utilizar `form.clean_data`:

```
if (form.is_valid()) {
    var name = form.clean_data['first_name'],
        year = form.clean_data['email'];
        //...

}
```

La validación se lleva a cabo llamado al método `clean()` de cada campo del formulario. La salida de cada una de esas llamadas completa el arreglo asociativo `cleaned_data`. Finalmente se ejecuta la validación integral del formulario llamando al método `clean()` del formulario, si existe.

Se puede generar una validación personalizada creando una subclase del campo para el cual se requiera validación extra. Existe un mecanismo abreviado que consiste en implementar el método `clean_nombreCampo` directamente sobre el formulario. Por ejemplo:

```
var AuthorForm = type('Author', [ forms.Form ], {
    salutation: new forms.CharField({ maxlength: 10 }),
```

```
    first_name: new forms.CharField({ maxlength: 30 }),
    last_name: new forms.CharField({ maxlength: 40 }),
    email: new forms.EmailField(),
    headshot: new forms.ImageField({ upload_to: '/tmp' }),
    // Validación personalizada
    clean_salutation: function () {
        var salutation = this.data['salutation'];
        if (!salutation in ['mr', 'ms', 'mrs', 'miss']) {
            throw new ValidationError("%s no es un saludo válido".subs(
                salutation);
        }
        return salutation;
    }

});
```

El orden de validación es el siguiente:

1. `clean()` de cada campo del formulario (`CharField.clean()`, `EmailField.clean()`, etc).
   a) `clean_nombreCampo` sobre el formulario, si existiese.
2. `clean()` del formulario.

Si en este ciclo no se captura ninguna excepción los datos son válidos. En caso contrario. El formulario almacena el mensaje de error y lo muestra adecuadamente cuando se realiza su representación.

6.10.2 Cración de Formularios a Partir de Modelos

La mayoría de las veces los formularios son utilizados para el ingreso de datos a los modelos, por lo que la tarea de escribir un formulario para cada modelo es muy común. Doff implementa en el módulo `doff.forms.models` un tipo especial de formulario que inspecciona la definición del modelo y genera automáticamente los campos del formulario. De esta manera, cualquier cambio en los campos del modelo se refleja automáticamente en los campos del formulario. Además añade un método `save()` que genera de manera automatizada una instancia en la base de datos del modelo asociado.

Para utilizar este tipo de formularios se debe extender de `doff.forms.models.ModelForm` y suministrar en el arreglo asociativo `Meta` el modelo al cual se asocia. Por ejemplo:

```
var AuthorForm = new type('AuthorForm', [ ModelForm ], {
    Meta: {model: Author}
});
```

Adaptando la vista del ejemplo del apartado anterior, el codigo es el siguiente:

```
    function crear_autor(request) {
     // El formulario es para validación
     if (request.method == "POST") {
         var form = new AuthorForm({ data: request.POST });
         // Los datos ingresados son válidos?
         if (form.is_valid()) {
             // Se crea el autor a partir del formulario
             var autor = form.save();
             // Se redericciona a una página sobre el autor
             return redirect('/autores/%d'.subs(autor.id));
         }
         // Si el formulario no es válido, cuando se muestre
         // en la plantilla, se marcarán los errores de validación
     } else {
         // Se crea el form vacío
         var form = new AuthorForm();
     }

     return render_to_response("mi_formulario", {
         form: form
     });

}
```

ho********** La aplicación **********

Una vez lograda la implementación de Django en el cliente, el framework de aplicaciones Doff, se comenzó a trabajar en la integración con los proyectos en línea.

Doff fue desarrollado con el objetivo de realizar la menor cantidad de reescritura posible de las aplicaciones Django al momento de ser ejecutadas de manera desconectada. Si bien fue intecinalmente omitido en el capítulo sobre el framework desconectado, contar con la misma API de ORM hace posible la generación de modelos del cliente de manera autónoma a partir del análisis de las defincciones de los modelos de la aplicación en línea.

De la misma manera, las plantillas del proyecto conectado, no requieren adaptaciones (en el caso de que utilicen JavaScrpipt, deben adecuarlo a las normativas de Doff y las etiquetas de templates no estándares deben ser reescritas en Doff).

Las vistas y las URLs deben ser reescritas, ya que por no contarse con el mismo soporte para expresiones regulares, y debido a que las vistas en Django son funciones escritas en Python, estas no pueden ser traducidas a JavaScript sin la intervensión del programador.

En el capítulo sobre Doff se introdujo vagamente una aplicación Django llamada `offline` que tenía como objetivo servir estáticamente al cliente con el código de Doff y Protopy. El presente capítulo se centra sobre la utilidad de dicha aplicación.

Se introduce la entidad `RemoteSite` que tiene como objetivo vincular la aplicación desconectada

con la aplicación en línea y brindar mecanismos para seguridad y sincronización de datos.

El mecanismo de comunicación con Django es la lína de comandos (CLI) [2], y brinda una API para la generación de comandos presonalizados que se aprovechó para automatizar varios pasos de la migración de un proyecto.

[2] Tal como se mencionó en el capítulo teórico sobre las tecnologías del servidor.

CAPÍTULO 7

RemoteSite

La clase `RemoteSite` o sitio remoto define un proyecto desconectado. Tiene varias responsabilidades, entre ellas, servir el código JavaScript del framework Doff, el del proyecto, administrar la forma en que el cliente accede a los datos y permitir la sincornización. Esta clase se impleentó en `offline.sites.RemoteSite`.

Su interfase está basada en la adminstración que provee Django (`django.contrib.admin`), donde un sitio de administración se encarga de las operaciones de CRUD sobre los modelos que le son registrados. Pueden existir múltiples sitios de administración en un proyecto. Cada sitio se publica en una URL dentro del proyecto [DjangoNewFormsAdminBranch09].

Un `RemoteSite` representa *una* migración del proyecto. Es decir, para un mismo proyecto en línea, pueden existir una o más migraciones. Cada sitio remoto se encuentra publicado en alguna URL en el proyecto. Cada sitio remoto posee una serie de modelos registrados, sobre los cuales se puede definir cuales son de solo lectura y cuales de lectura-escritura (en la sincronización). Mediante el mecanismo de autenticación y autorización de Django (`django.contrib.auth`) se puede restringir que usuario o rol accede a cada sitio remoto.

La creación de un sitio remoto no se realiza manualmente. Cuando se instala la aplicación `offline` en el proyecto en línea, se añade un comando para la creación automatizada. Este comando se encarga de crear un directorio (que también es un módulo Python) donde se almacena los módulo que define los sitio remoto. Cada sitio remoto posee un nombre único. El nombre de este directorio se define en el módulo `settings.py` del proyecto como una constante de cadena con el nombre `OFFLINE_BASE`.

Por cada sitio remoto existe un subdirectorio donde se almacena el código JavaScript del proyecto y sus aplicaciones: `OFFLINE_BASE/<nombre_sitio_remoto>`. En su nivel base se cuenta con los módulos `settings.js`, `urls.js`. Cada aplicación es un subdirectorio y cuenta con módulos `views.js`, `mixins.js`, `tests.js` y `urls.js`.

Desde el punto de vista de Django, un sitio remoto publica sus métodos como vistas [1] y se comporta como un módulo de `urls`, pero a diferencia de una vista de usuario, la URL (patrón) donde

[1] Tomado de TurboGears y Pylons.

se publica está definida. El patrón que se debe utilizar es una propiedad de la instacia llamada `urlregex`, es decir que el programador no define donde se publica el sitio sino que está definido por el sito [2] (internamente en función del nombre del sitio remoto y del parámetro `OFFLINE_BASE`). Por ejemplo:

```
from soporte_offline.remote_mi_sitio import site as mi_sitio

urlpatterns = patterns('',
    # Las primeras dos asocaiciones utilizan una cadena
    (r'^$', index,)
    (r'^vendedores/(?P<id>\d)/$', vista_vendedores),

    # En el caso de los sitios remotos, la URL está indicada por el atributo
    # urlregex
    (r'^%s/(.*)' % mi_sitio.urlregex, mi_sitio.root ),
)
```

En el caso anterior la propiedad `urlregex` es `/soporte_offline/mi_sitio` y en ese lugar se encuentra el punto de entrada para la ejecución del proyecto desconectado. Al acceder a esta URL el navegador se encuentra con:

```
<html>
<head>
    <!-- El sitio remoto publica la librería Protopy, como JavaScript 1.7 -->
    <script type="text/javascript;version=1.7"
        src="/soporte_offline/mi_sitio/lib/protopy.js"></script>
    <!-- Creación de la instancia del proyecto desconectado -->
    <script type="text/javascript;version=1.7">
        <!-- Requerir del módulo de proyectos, la función new_project -->
        require('doff.core.project', 'new_project');
        <!-- Instanciación  del proyecto -->
        var mi_sitio = new_project('mi_sitio', '/soporte_offline/mi_sitio');
        <!-- Darle el control del navegador a la instancia del proyecto -->
        mi_sitio.bootstrap();
    </script>
</head>
<body>
</body>

</html>
```

Como se pudo observar en el ejemplo anterior, el código del Protopy se encuentra en `lib/protopy.js` (línea 4 y 5), y como se deinfinió en el capítulo anterior, el códgo de Doff se encuentra en `lib/packages/`. En la línea 10, se crea una entrada en `sys.path` de Protopy, de-

[2] Si bien no se recomienda modificar la URL en la cual se publica un sitio remoto, el desarrollador puede modificarla.

finiendo que el paquete `mi_sitio` se encuentra en la URL `/soporte_offline/mi_sitio`. En este caso `mi_sitio` es el paquete que define al proyecto. Para cargar el módulo `settings` mediante `require("mi_sitio.settings")` se está realizando una petición a `/soporte_offline/mi_sitio/js/settings.js`.

Además de las URLs descritas, el sitio remoto publica otras de dónde Doff descarga los modulos y recursos que componen a la aplicación, para almacenarlos localmente y poder ejecutar el proyecto cuando el cliente se encuentre sin conexión. Una de estas URLs es `manifest.json`, en la cual se encuentra una lista completa de los archivos que componen al proyecto. Durante la instalación del proyecto se utiliza esta lista para alamacenar los recursos que componen al proyecto.

A continuación se realiza una descripción del método de migración de proyectos.

CAPÍTULO 8

Migración de un Proyecto

La migración de un proyecto a través de los sitios remotos consiste de los siguientes pasos:

1. Instalación de la aplicación de soporte (`offline`), este paso es necesario solo una vez.
2. Creación de un sitio remoto
3. Migración de una aplicación
4. Registro de modelos en el RemoteSite
5. Publicación del sitio remoto
6. Creación del `manifest.json`, este paso se considera el paso a producción de la aplicación desconectada.

El primer paso para la migración consiste en la instalación de la aplicación de soporte `offline` al proyecto. Para esto se la descarga y añade al `PYTHONPATH`. Posteriormente se debe agregar a `INSTALLED_APPS` del proyecto en linea.

Una vez realizado este paso se habilitan varios comandos para la administración de proyectos desconectados al módulo `manage.py`. Estos comandos permiten realizar varios de los pasos antes mencionados. Los comandos implementados son los siguientes:

- `start_remotesite`

  Este comando recibe como argumento el nombre de sitio remoto que se desea crear. Si no exite el directorio `OFFLINE_BASE` en el proyecto en línea lo crea y dentro de el añade el módulo `remote_<nombre-site>.py` en el cual define la instancia de `RemoteSite` con el nombre dado. Utiliza también el nombre para crear un subdirectorio en `OFFLINE_BASE` donde crea el esqueleto del proyecto desconetado (crando `settings.js` y `urls.js` a partir de plantillas que son rellenadas con la configuración disponible de la aplicación en linea).

  Por ejemplo:

```
$ manage.py start_remotesite vendedor_viajante
```

Esto crea en el directorio de la aplicación la siguiente estuctura (suponiendo que `OFFLINE_BASE` sea la cadena `soporte_offline`):

```
soporte_offline/
    remote_vendedor_viajante.py
    vendedor_viajante/
        settings.js
        urls.js
        logging.js
```

- `list_remotesites`

  Este comando lista los `RemoteSites` de un proyecto en línea. Verifica además si estos se encuentran publicados. No recibe argumentos.

- `migrate_app`

  Migra una aplicación conectada a un `RemoteSite`. Recibe como argumentos el nombre del sitio remoto y el nombre de la aplicación. Dentro de `OFFLINE_BASE/<nombre-site>/<nombre-app>` crea la estructura de la aplicación (crea los módulos `views.js` y `models.js` a partir de plantillas).

  Por ejemplo:

  ```
  $ manage.py migrate_app vendedor_viajante catalogo
  ```

  Basados en el ejemplo del comando `start_remotesite`, produciría la siguiente estructura de archivos:

  ```
  soporte_offline/
      remote_vendedor_viajante.py
      vendedor_viajante/
          settings.js
          urls.js
          logging.js
          catalogo/
              views.js
              mixins.js
              tests.js
              urls.js
  ```

- `manifest_update`

  Acutaliza los recursos que componen al proyecto desconectado: vistas, plantillas, modelos y recursos estáticos.

La defnición de un remotesite dentro de `OFFLINE_BASE/remote_<nombre-site>.py`, resultado del comando `manage.py create_remotesite <nombre-site>` tiene la siguiente estructura:

```
from offline.sites import RemoteSite

site = LibrarianRemoteSite('librarian')
```

Cada aplicación migrada mediante el comando `migrate_app` es creada a partir de una esqueleto genérico, donde el programador deberá implementar las URLs y vistas. Si bien esta tarea podría parecer tediosa en primera intancia, el grado de similitud entre Protopy con Python y el hecho de que Doff implemente la misma API que Django facilita mucho la tarea.

Las plantillas y los modelos son provistos autmáticamente por el sitio remoto. A continuación se analiza la forma en la que se exponen los modelos y como se puede modificar.

CAPÍTULO 9

Publicación de Modelos

Como se mencionó en el capítulo introductorio, es deseable contar con un mecanismo de separación de los datos a los cuales accede al cliente por seguridad y eficiencia.

Para tal fin se creó la clase `RemoteModelProxy` o proxy de modelos que envuelve a los modelos de las aplicaciones. Por defecto ningún modelo es visible para el cliente. Se deben publicar explicitamente mediante un método llamado `register(model, proxy = None)` [1].

El método `register` debe ser llamado sobre la instancia del remote site (definida en el módulo `OFFLINE_BASE.remote_<nombre-modulo>` - en el sistema de archivos OFFLINE_BASE.remote_<nombre-modulo>.py). Este método recibe una instancia opcional de `RemoteModelProxy` que en el caso de no ser provista se genera internamente.

Por ejemplo, la defincón del sitio remoto quedaría como sigue para el ejemplo anterior:

```
from offline.sites import RemoteSite
from bookstore.core.models import Book, Author

site = LibrarianRemoteSite('librarian')
site.register(Book)
site.register(Author)
```

De esta manera las vistas del proyecto desconectado tendrán disponible los modelos `Book` y `Author`.

Para personalizar la definición y acceso a datos de un modelo, se debe crear una subclase de `RemoteModelProxy`. Dentro de esta subclase se pueden definir los campos a publicar mediante (`fields`, `include`, `exclude`) y el `Manager` del modelo a utilizar [2]. Por ejemplo:

[1] Esta idea también fue tomada de la aplicación genérica `django.contrib.admin` antes mencionada.

[2] Se pueden definir campos que no existen en el modelo que deberán ser provistos por el Manager a la hora de la sincronización.

```
from offline.sites import RemoteSite
from bookstore.core.models import Book, Author

site = LibrarianRemoteSite('librarian')

class BookRemote(RemoteModelProxy):
    class Meta:
        model = Book # Este campo es opcional
        exclude = ('author', )
        manager = Book.objects

site.register(Book, BookRemote) # Registro
```

El Manager permite filtrar las entidades que son accedidas por el cliente. Si el cliente es un usuario autenticado en el proyecto, mediante la implementación de un `Manager` que discrimine usuarios autenticados [3], se puede limitar la visión de las instancias de un modelo (o filas sobre la base de datos).

Cuando se migra una aplicación al cliente, mediante el comando `migrate_app`, no se crea un módulo `models.js` como podría esperarse. El sitio remoto lo genera automáticamente en función de los modelos que hallan sido registrados y los publica en cada aplicación del sitio remoto (es decir, los modelos son globales al proyecto desconectado).

El sitio remoto solo se encarga de publicar la estrcutra del proxy de modelos, los métodos opcionales con los que el desarrollador desee contar se deben implementar en un mixin. Para este fin existe una archivo por cada aplicación migrada llamado `mixins.js` donde se definen los métodos de los modelos como un arreglo asociativo. El nombre del arreglo debe conincidir con el nombre del modelo. Por ejemplo, para el modelo de `Book`:

```
var Book = {
    // Representación en cadena del modelo
    __str__: function () {
        return this.nombre;
    }
};

publish({
    Book: Book
})
```

En Django los modelos suelen tener métodos de utilidad, como `__unicode__` (en el cliente `__str__` [4]). Estos métodos se deben implementar en un arreglo asociativo que se utiliza a modo de `Mixin`. Como se mencionó en el capítulo sobre Protopy, la impmentación de clases soporta

[3] El usuario se ha autenticado en la aplicación en línea contra las entidades de `django.contrib.auth.models.User`.

[4] Una epxplicación sobre los detalles sobre este método se encuentra

herencia múltiple, cuando se requiere la url con la definción del modelo, se requieren automáticamente los Mixins definidos.

Durante la instalación del proyecto, cuando el `ManagedLocalStore` descarga el recurso, el sitio remoto provee automaticamente la conjunción de la introspección realizada sobre los modelos con los métodos agregados en el mixin.

En el siguiente esquema se muestra la intreacción entre el sitio remoto, los modelos de una apicacción, sus managers y los proxies de modelos.

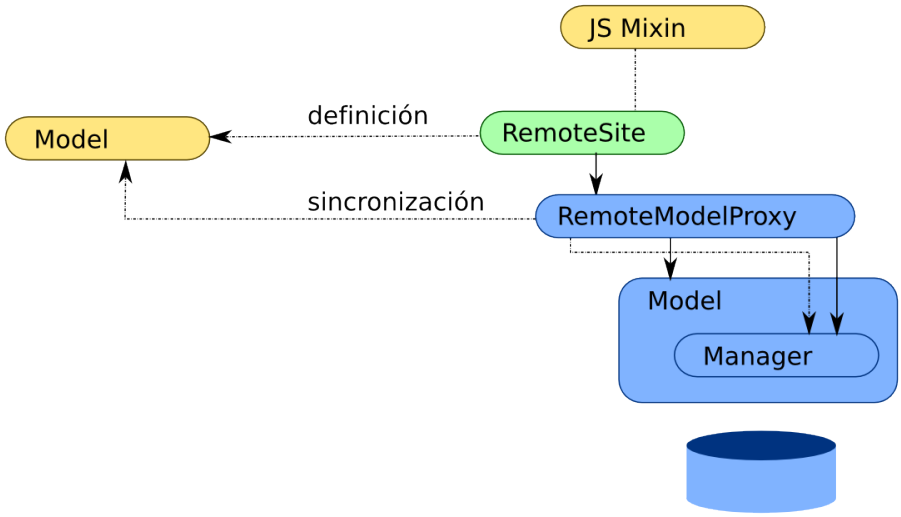

Figura 9.1: Modelos de una aplicación desconectada

Por ejemplo, para una definción de un modelo como sigue:

```
# ------------------------------------------------
# Definción del modelo en models.py
# ------------------------------------------------

from django.db import models

class Persona(model.Model):
    nombre = models.CharField(max_length = 40)
    apellido = models.CharField(max_length = 40)

    # Representación en cadena Unicode
    def __unicode__(self):
        reutrn u"%s %s" % (self.nombre, self.apellido)

# ------------------------------------------------
# Definción del RemoteSite
# ------------------------------------------------
```

```
site = RemoteSite('personas')
site.register(Persona) # Sin proxy
```

Luego de haber ejecutado el comando `migrate_app` el código del Mixin para implementar la funcionalidad de `__unicode__` sería:

```
// El Mixin es simplemente un arreglo asociativo, donde
// se pueden incorporar más métodos.

var Persona = {
    // En el servidor la salida del modelo se transforma en el encoding
    // del template, en cambio, en el cliente la codificación ya está
    // establecida, por lo que se utiliza el método __str__
    __str__: function() {
        return "%s %s".subs(this.nombre, this.apellido);
    }
}
```

9.1 Modelos de Solo Lectura

Si bien se tratará la sincronización en un apartado posterior, se ha mencionado que existe un mecanismo para sincroizar los datos de los modelos. La sincronización puede ocurrir del servidor al cliente, la cual es probablemente necesaria en la instalación y en sentido inverso para sincronizar los datos generados o modificados durante la ejecución desconectada del proyecto.

Cuando un modelo se registra en un sitio remoto, se concede permiso de modificación a los campos definidos en el proxy de modelos (tomados automáticamente del modelo). Si se registra una modelo que posee claves foraneas, y los modelos referenciados no son registrados, se genera un registro implícito de estos modelos como solo lectura.

Es decir, si contamos con un modelo como el siguiente:

```
from django.db import models

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

class Provincia(models.Model)
    provincia = models.ForeignKey(Provincia) # Referencia a Provincia
    nombre = models.CharField(max_length = 40)
    habitantes = models.PositiveIntegerField(default = 0)
```

continuando el ejemplo, si en la definición del sitio remoto solo se registra la entidad `Provincia`:

```
site.register(Provincia)
```

la entidad `Pais` se registra implicitamente como un modelo de solo lectura. Durante la sincronización los datos solo se transfieren del servidor al cliente. Los unicos campos transferidos son el `pk` (o `id`) y la representación en cadena del modelo (`__unicode__`).

CAPÍTULO 10

Publicación de un Sitio Remoto

La publicación de un sitio remoto es explícita y consiste en agregar al módulo `urls.py` del proyecto un patrón como el siguiente (suponiendo que `OFFLINE_BASE` sea "soporte_offline" y el nombre del sitio remoto sea "bookstore"):

```
from soporte_offline.bookstore import bookstore_site

(r'^%s/(.*)' % bookstore_site.urlregex, bookstore_site.root )
```

El atributo `urlregex` del sitio remoto calcula automáticamente la URL del sitio como la concatenación de `OFFLINE_BASE` y el nombre del sitio. En el caso anterior, para acceder al sitio desconectado se debe acceder a la URL:

```
http://misitio.com/soporte_offline/bookstore
```

Esta URL genera automáticamente la instancia del proyecto en el cliente.

Finalmente, podemos resumir el ciclo de migración mediante la siguiente figura:

A continuación se trata uno de los objetivos

Nota: HASTA ACÁ, ESTE NO ES MENTIROSO

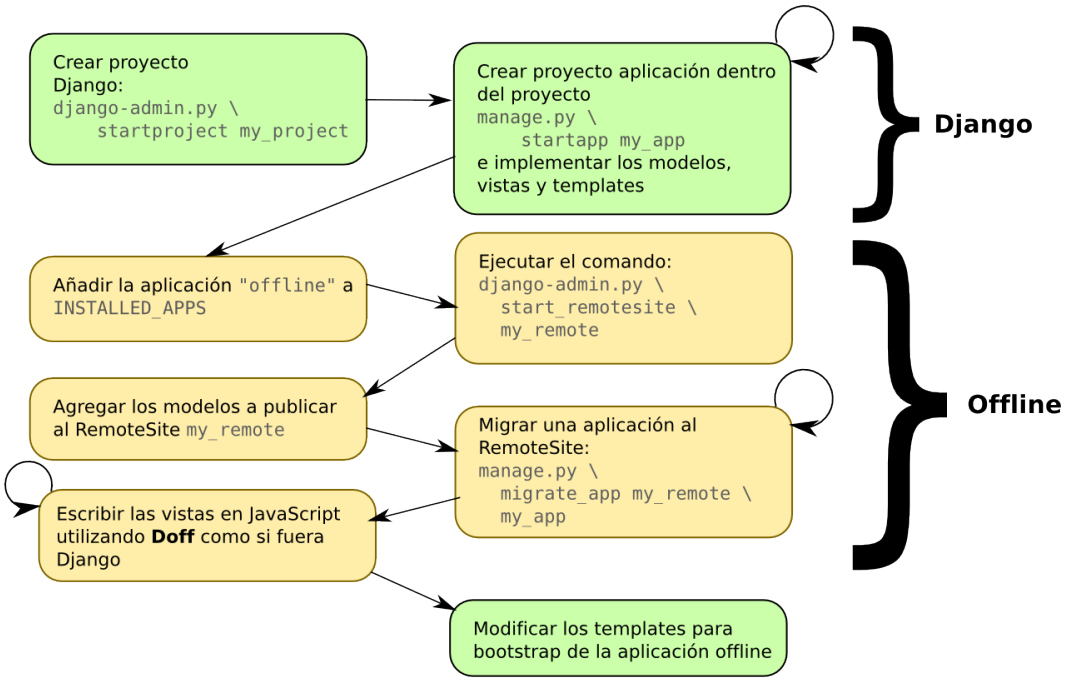

Figura 10.1: Esquema de trabajo con la aplicación offline

CAPÍTULO 11

Sincronización de Datos

Por simplcidad, en este apartado, se establece la siguiente terminología: Cuando se hable de *el cliente* se está refiriendo al proyecto desconectado,

el servidor es el proyecto en línea.

La sincronziación de datos permite que las entidades definidas en el servidor se repliquen en el cliente y que las creadas y modificadas por el cliente se transfieran a la aplicación en linea, permitiendo así que se porpaguen a el resto de los cileintes.

Sin embargo diversas situaciones en las cuales se pueden presentar conflictos entre las versiones de de una misma entidad:

- Cuando una entidad se modifica tanto en el cliente como en el servidor
- Cuando una entidad se modifica en dos clientes
- Cuando dos o más entidades equivalentes se crean en dos clientes
- Cuando se crea una entidad equivalente en el cliente y en el servidor equivalente
- Cuando se elimina una entidad en el servidor referenciada en el cliente
- Alguna combinación de los casos anteriores

La forma en que se resuelven estos conflictos debe ser abordada por el programador durante el diseño de la aplicación.

Se implementó una mecánica simple y extendible que resuelve casos simples en `doff.contrib.syncrhonization`.

Para implementar la sincronización se trataron los siguientes items:

- Integridad de la sincronización (orden de los modelos, transacciones)
- Transporte de datos
- Detección de cambios en instancias en el servidor

- Detección de cambios en el cliente

11.1 Transporte de Datos

El primer aspecto a considerar es el transporte de datos. Las bases de datos relacionales utilizan el lenguaje SQL como estándar para la definicón de su estructura y manipulación de datos. Django provee un ORM que encapsula el acceso al RDBMS y brinda una capa de abstracción en el lenguaje Python. Doff realiza lo propio en el lenguaje JavaScript y apoyándose en las extensiones del provistas por Protopy. Gracias al trabajo del sitio remoto, mediante el registro de modelos, se obtiene consistencia entre las defincciones de alto nivel del servidor con las del cliente, por lo cual se optó por realizar el transporte utilizando los mecanimos de alto nivel disponibles en Django implenetando el soporte necesario en Doff.

Django provee un módulo llamado `serializers` que brinda un conjunto de serializadores (*marshalling*) de instancias de `QuerySet`. Un serializador se encarga de *aplanar* las intancias en texto con algún formato. Los formatos de salida provistos por Django son: XML, JSON y Python. Cuando un serializador trabaja sobre un `QuerySet` almacena información sobre el modelo para poder luego recuperar las instancias medinate un des-serializador. L

Se optó por el serializador basado en JSON, debido a que presenta ventajas a la hora de depuración aunque teniendo como penalización el tiempo de procesamiento sobre conjuntos grandes de datos.

Los serializadores son obtenidos a través de un factory de clases con la signatura `get_serializer(nombre)`. Para el siguiente modelo:

```
class Pais(models.Model):
    nombre = models.CharField(max_length = 45)
    simbolo_moneda = models.CharField(max_length = 8, default = '$')

    def __unicode__(self):
        return self.nombre

    class Meta:
        verbose_name = u"País"
        verbose_name_plural = u"Paises"

class Provincia(models.Model):
    pais = models.ForeignKey(Pais)
    nombre = models.CharField(max_length = 140)

    def __unicode__(self):
        return self.nombre
```

se puede serializar en JSON de la siguiente forma

```
from core.models import Provincia # Importanción de modelos
# Importación del módulo de serialización
>>> from django.core import serializers
# Obtención de la calse de serialización en JSON
>>> SerializadorCls = serializers.get_serializer('json')
# Creación de una instancia del serializador
>>> serializador = SerializadorCls()

# Serialización
>>> plano = serializador.serialize(Provincia.objects.all())
# Los datos son texto plano
>>> plano

    '[
        {   "pk": 1,
            "model": "core.provincia",
            "fields":   {
                "pais": 1,
                "nombre": "Buenos Aires"
            }
        },
        {
            "pk": 2,
            "model": "core.provincia",
            "fields": {
                "pais": 1,
                "nombre": "C\\u00f3rdoba"
            }
        }
     ]'

# Para recuperar los datos, se obtiene una función des-serializador
# que retorna instncias guardables

>>> deserializador_func = serializers.get_desserializer('json')

# Invocación del deserializador sobre el texto serializado
>>> for obj in deserializador_func(plano):
...     # Guardado de la isntancia
...     obj.save()
```

Como se mencionó anteriormente, los modelos expuestos al cliente se encuentran en registrados en el sitio remoto mediante un proxy. El Manager definido en este proxy es el que se utilizará a la hora de serializar los datos para su envío al cliente. Por defecto, si no se define un Manager, se utiliza `_default_manager` que es un alias de `objects`.

Se implementó sobre Doff el modulo de serialización para poder recuperar los datos y generar las

instancias correspondientes en el cliente.

Si bien con la utilización de los serializadores brinda un mecanismo extensible y simple de transporte de datos, se deben tener en cuenta que en el caso de contar con relaciones en los modelos, los datos serializados deben ser enviados en orden para no violar la integridad diferencial, provocando una falla en el método `save()` en la deserialización. Desde el punto de las relaciones, los modelos conforman una gerarquía de árbol con múltiples raices. Gracias a la metainformación que brindan las definciones de alto nivel, el orden de presedencia fue calculado de manera sencilla.

11.2 Versionado de Modelos

Mediante los serializadores y se puede realizar una transferencia de datos entre el cliente y el servidor, pero no se ha expuesto aún un mecanismo para controlar los cambios realizados sobre las instancias tanto en el proyecto en línea como en el proyecto basado en Doff.

Para realizar el análisis de la información necesaria sobre los cambios de las instancias se planteó un escenario hipotético que se describe a continuación:

> Existe un proyecto en linea, que cuneta con un sitio remoto publicado. Existe uno o más instancias del proyecto creado a partir del sitio remoto. Los clientes poseen conexión ocacional, durante la cual realizan la tarea de sincronización en la cual envían y reciben los cambios del sitio remoto. Ante un conflicto de datos insalvable de manera automática se debera adoptar una política: adecuación por el usuario, prevalecen datos del servidor, prevalecen datos del cliente, el criterio de permanencia está dado por el la antiguedad o privilegios del usuario.

Los camibos en las instancias ocurren ante los eventos de creación, modificación y eliminación. Sin embargo, esto eventos se producen tanto en la aplicación del cliente como en la aplicación en línea, por lo que se realiza un anális por separado de cada una de estas situaciones:

- Creación de una entidad

  Ocurre cuando se genera una instancia de alguna entidad del ORM y se invoca el método `save()`

  - En el servidor

    El cliente debe copiar la nueva intancia en la próxima sincronización.

  - En el cliente

    La entidad no existe en el serivor, por lo que se debe crear una clave transitoria en el cliente. Durante la sincronización la entidad será creada en el proyecto en línea, devolviendo al cliente el identificador del servidor.

- Modificación de una entidad

Ocurre cuando se recupera una instancia mediante un Manager y se modifican sus valores, llamando posteriormente al método `save()`.

- En el servidor

  Se debe crear un registro de que la entidad ha sido modificada.

- En el cliente

  Si la entidad se creó en el cliente, no ocurre nada, la entidad sigue siendo nueva para el servidor. En cambio si la entidad fue sincronizada con el servidor se debe crear un registro de que la entidad ha sido modificada.

- Eliminación de una entidad

  Ocurre cuando se invoca `delete()` sobre una entidad recuperada mediante un Manager, o cuando se invoda directamente sobre el Manager.

  Debido a que las entidades pueden tener relaciones, es necesario establecer una política. En Doff por defecto se adoptó que las entidades referenciadas están en estado de baja lógica hasta que se invoca una función de purgado.

  - En el servidor

    Puede que el cliente tenga entidades que referencien a la instancia eliminada por lo tanto se debe usar una baja lógica. Eventualmente cuando el programador considere adecuado se realiza la baja física mediante la llamada a una función de purgado.

  - En el cliente

    La eliminación en un cliente debe provocar una baja lógica en el servidor.

Del breve análisis expusto se deduce que se debe almacenar información extra referente al estado de una instancia.

Junction [JunctionDocsSync09] es un framework de desarrollo de aplicaciones web que abordó el problema de sincronización agregando una serie de campos en las entidades:

```
- id          integer primary key autoincrement
- created_at  datetime
- updated_at  datetime
- active      integer
- version     integer
- id_start    integer
- id_start_db varchar(40)
- synced_at   datetime
```

En el caso de Junction, la información de versionado se almacena tanto en el servidor como en el cliente. Aplicar este enfoque sobre un proyecto existente de Django no es factible debido a que involucra modificar todas los modelos pre-existentes.

Se optó por el agregado de información de manera asimétrica: En el cliente las entidades se crean con campos extras, mientras que en el servidor se utilizan relaciones genéricas.

11.2.1 Información de Sincronización en el Cliente

La información sobre sicronización en el cliente se basó en parte en los campos que propone Junction, agregándoselos a cada entidad, pero se modeló la

sincronización como entidad por separado.

Se creo la aplicación genérica `doff.contrib.offline` y donde se definió `SyncLog`. Cada vez que el cliente realiza una sincronización con el servidor, se almacena la fecha sobre una instancia de esta entidad, de manera que durante la sincronización siguiente, solo se trabaje con los modelos afectados en el intervalo de tiempo transcurrido.

Los datos que se agregaron a cada modelo del cliente fueron:

- `server_pk`

  Es la calve de la entidad en el servidor. El campo `pk` se refiere al identificador de la entidad en el cliente. Si la entidad se creo en el cliente, este campo es `NULL`

- `active`

  Indica la baja lógica en el servidor. Por defecto toma el valor `false`.

- `sync_log`

  Referencia la entidad de `SyncLog` en el cual fue sincronizado.

- `status`

  Este campo se creó con el objeto de indicar el estado de una entidad con respecto a su contraparte en el servidor, sus valores posibles son los siguientes:

  - `"C"` (created)

    La entidad se creó en el cliente mediante el método `save()`. El atributo `sync_log` es `NULL`.

  - `"S"` (synched)

    Indica que la entidad se encuentra sincronizada con el servidor. En esta caso, el atributo `sync_log` no es `NULL`

  - `"M"` (modified)

La entidad sincronizada ha sido modificada en el cliente. Ante una sincronización se acutaliza la referencia de `sync_log` y se vuelve al estado de sincronizado (`"S"`).

- `"D"` (deleted)

  Indica que la entidad fue borrada en cliente. Ante una sincronización el servidor pasa a estado `"S"` y el atributo `active` pasa a `false`.

Se agregó en el cliente la lógica para la manipulación de los campos `active` y `status`, dejando el trabajo de manipulación de los campos `server_pk` y la relación a `sync_log` para el proceso de sincronización.

11.2.2 Metainformación de Sincronización en el Servidor

Se mencionó en el capítulo en el cual se introdujo Django que el ORM de este framework brinda lo que se conoce como el framework de `ContentType` (no confundir con la cabecera HTTP), que consiste en una serie de `Fields` del ORM que permiten crear relaciones genéricas. Además se mencionó que el framework posee un sistema de señales que permiten el conexionado de eventos del ORM a funciones.

Con el objeto de registrar los cambios sobre entidades en el servidor, se decidió crear una entidad denominada `SyncData`. Una instancia de esta entidad se crea antes los eventos de guardado (modificación) y eliminación de las entidades registradas en el sitio remoto mediante el método `register(model, proxy = None)`.

De esta manera solo las entidades que forman parte del proyecto desoncetado son "vigiladas" por cambios, registrandose ante estos la fecha. Cada vez que se registra un modelo a un sitio remoto internamente también se registran manejadores de eventos ante las señales de guardado, `post_save` y eliminación, `post_delete`.

11.3 Protocolo de Sincronización

Se eligió RPC sobre HTTP como mecanismo de comunicación entre el cliente y el servidor por sobre llamadas asincrónicas simples o AJAX, debido de que se requiere pasaje de parámetros más complejos que los contemplados en la codificación `application/x-www-form-urlencoded` o `multipart/form-data` [W3CFormEncoding09]. Se eligió JSONRPC como mecanismo de comunicación y se publicó en la URL `/rpc` del sitio remoto.

Basados en las primitivas de sincronización de los sistemas de control de versiones distribuidos como Git [GitDocs09], Mercurial [MercurialDocs09] o Bazaar [BazaarDocs09] se crearon dos métodos:

- `pull()`

Requiere los cambios del servidor

- `push()`

  Envía los cambios al servidor

Dentro del modulo `doff.contrib.offline.sync` se crearon los métodos `pull()` y `push()` que hacen las llamadas a las funciones homónimas publicadas en `/rpc` del sitio remoto. Para utilizarlo se reliza lo siguiente:

```
>>> require('doff.contrib.offline.sync')
// Descargar los datos del serivdor
>>> sync.pull()
```

Si bien en el código expuesto la función no recibe parámetros, internamete se envía la última instancia del modelo `SyncLog` del proyecto desconectado.

En Doff se plantearon dos tipos de sincronización: la *sincronización inicial*, que ocurre cuando un cliente instala el proyecto desconectado a partir del sitio remoto y la *sincronización normal* que sucede cada vez que el cliente desea enviar sus cambios y obtener las novedades sobre las instancias del modelo del proyecto. La sincronización es explícita para el cliente, debdio a que en ocaciones donde se requiera asistencia del usuario para dirmir los conflictos esto no ocurra de manera sorpresiva.

La aplicación genérica `doff.contrib.offline` y en especial su módulo `syncrhonization` son los encargados del manejo de sincronización. Esta aplicación se instala por defecto (durante la ejecución del comando `start_remotesite`).

El sitio remoto publica por cada proxy (modelo registrado) el manager defindo (`objects` por defecto) mediante JSONRPC en la URL `/rpc`. Cuando se requiere un `QuerySet` de algún Manager expuesto, éste se serailzia utilizando el mecanismo de serializers de Django.

Se definieron dos métodos:

- `push()`

  Esta operación envía los cambios del cliente al servidor.

  1. Busca en el cliente la última sincronización (última instancia de `SyncLog`) si existe (en el caso de tratarse de la sincronización inicial se envía `null`).
  2. Crea la instancia de instancia de `SyncLog` actual.
  3. Reliza una llamada a `begin_syncrhonization()` con la instancia de `SyncLog` recuperada en **1** como parámetro.
  4. El servidor responde con los modelos afectados, el orden y la fecha del servidor.
  5. Por cada modelo afectado se recuperan las instancias mediante la invocación de métodos del `RemoteManager` (los serializadores realizan el *marshlling* de los `QuerySets`). El `RemoteManager` se encarga de filtrar los modelos

modificados a partir de la fecha del último `SyncLog` y agregar los nuevos. Cada modelo es deserializado y guardado en la base de datos del cliente creando la referencia al `SyncLog` actual y estableciendo el campo `status` en `"S"`.

6. Se almacena en el `SyncLog` actual la fecha del servidor.

- `pull()`

1. El clinete envía

Esta operación trae los cambios del servidor al cliente.

En el siguiente diagrama se muestran las entidades que tienen entran en juego en la sincronización.

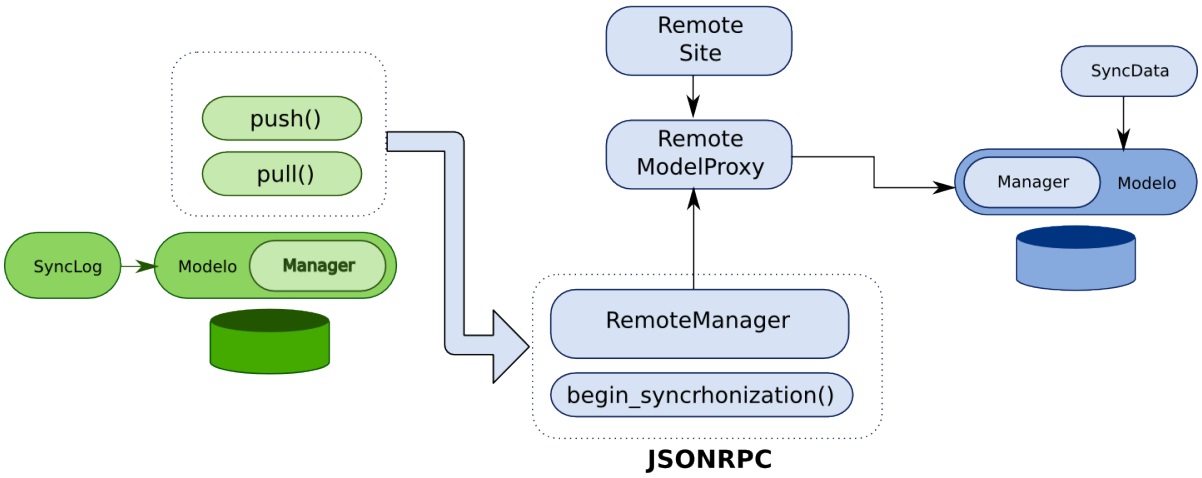

Figura 11.1: Esquema de sincronización en el que inteviene el módulo `synchronization` y el sitio remoto publicando los `RemoteManager` mediante JSONRPC.

Durante la sincronización ocurren cambios sobre el atributo `status` de las instancias del cliente.

11.4 Considreaciones sobre Sincronización

Doff provee un mecanismo básico de sincronziación adecuado para peuqueñas aplicaciones que permite al programador contar con las herramientas para transporte, comunicación RPC y control de cambios. Debido a que no existe una política universal aplicable a todas las aplicaciones, y debido a que Doff se trata de un framework extendible, se pueden implmentar mecanismos más sofisticados en base a las herramientas expuestas.

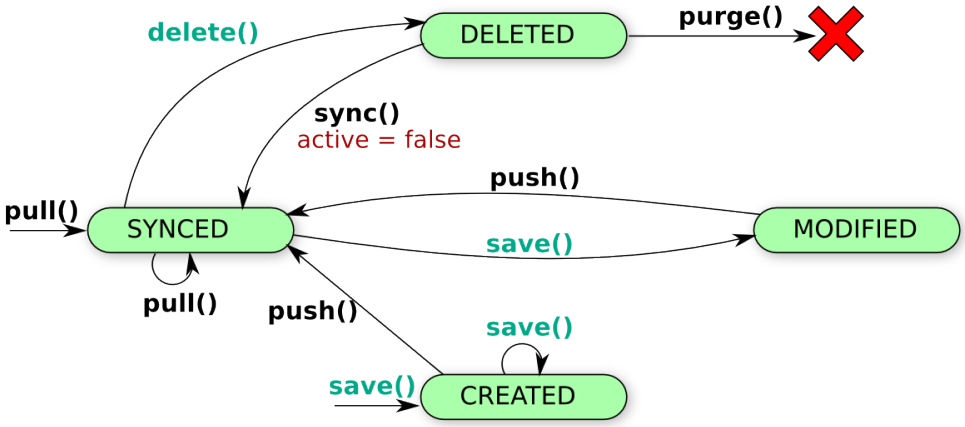

Figura 11.2: Estados de sincronización de los modelos en el cliente

CAPÍTULO 12

Aplicación de Demostración

Se diseñó una aplicación denominada `salesman` (agente de ventas viajante) para ejemplificar el funcionamiento de Protopy y Doff. La aplicación modela una empresa dedicada a la venta de productos mediante agentes viajantes. Cada viajante tiene asignada una serie de ciudades, donde realiza las ventas.

El parámetro `OFFLINE_BASE` es `"off"`.

Esta aplicación

12.1 Modelo

12.2 Vistas

La mayor parte de las vistas se implementaron mediante vistas genéricas. Este tipo de vistas se encuentran implementadas dentro del paquete `django.views.generic` y prveen funcionalidad muy frecuente como CRUD, consultas, comentarios, redirección, etc.

Por ejemplo:

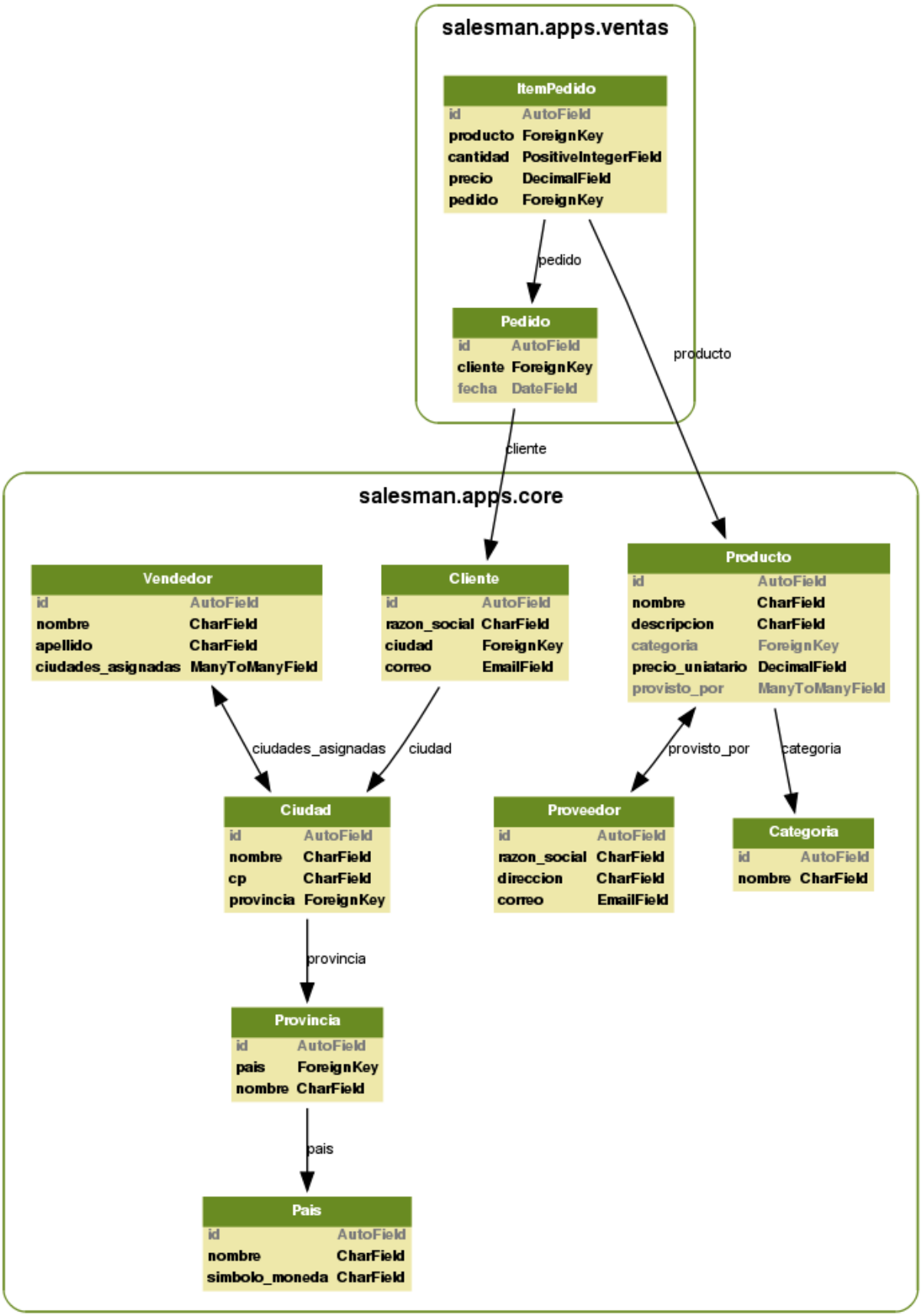

Figura 12.1: Diagrama de Base de Datos de Salesman (generado con `graph_models`)

CAPÍTULO 13

Conclusiones y Líneas Futuras

13.1 Conclusiones

El desarrollo consistió en el siguiente diagrama:

La solución propuesta se basó integramente en Open Source. Se utilizaron los siguientes productos:

- Python

  Lenguaje de programación.

- Django

  Framework de aplicaciones web escrito en Python.

- Aspen

  Servidor Web basado en WSGI

- Mozilla Firefox

  - Firebug

    Plugin de depuración.

- Mercurial

  Sistema de control de versiones basado en Python.

- Sphinx

  Sistema de generación de documentación basado en REST (REStrcutred Text). Sistema de documentación actual del lenguaje Python y varios proyectos importantes como Matplotlib.

  Se utilizó la salida a LaTeX y el comando `pdflatex` para generar la salida en PDF.

- Eclipse

  Entorno de desarrollo integrado escrito en Java/SWT multiplataforma. Posee complementos para varios lenguajes como Java, C/C++, PHP, Ruby, etc.

  - MercurialEclipse

    Plugin para control de versiones mediante Mercurial

  - Pydev

    Plugin para proyectos basados en Python, con resaltado de sintaxis, autocompleción, indicación de errores, etc.

- Kate

  Editor de texto basado en el entorno KDE con resaltado de sintaxis.

13.2 Lineas futuras

13.2.1 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.

13.2.2 Workers con soporte para Javascript 1.7

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 interfase del navegador. Una caractersística de los worker pools, es que se ejecutan en un ámbito de nombres diferente al del hilo principal y el único mecanismo de comunicación con el hilo principal es un sistema de cola de mensajes.

Se podría aprovechar este esquema para ejecutar ciertas partes de Doff en el uno o más Workers.

13.2.3 Compatibilidad con ES5 y HTML 5

EcmaScrtip 5 o JavaScript 2.0 resulve varios aspetctos que fueron abordados por Protopy/Doff, como un sistema de clases.

CAPÍTULO 14

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

APÉNDICE A

Referencia sobre el lenguaje Python

A.1 Filosofía del lenguaje

El código que sigue los principios de Python de legibilidad y transparencia se dice que es *pythonico*. Contrariamente, el código opaco u ofuscado es bautizado como *no pythonico* (*unpythonic* en inglés). Estos principios fueron famosamente descritos por el desarrollador de Python Tim Peters en **El Zen de Python**.

1. Bello es mejor que feo.
2. Explícito es mejor que implícito.
3. Simple es mejor que complejo.
4. Complejo es mejor que complicado.
5. Plano es mejor que anidado.
6. Ralo es mejor que denso.
7. La legibilidad cuenta.
8. Los casos especiales no son tan especiales como para quebrantar las reglas.
    - Aunque lo práctico gana a la pureza.
9. Los errores nunca deberían dejarse pasar silenciosamente.
    - A menos que hayan sido silenciados explícitamente.
10. Frente a la ambigüedad, rechaza la tentación de adivinar.
11. Debería haber una -*y preferiblemente sólo una*- manera obvia de hacerlo.
    - Aunque esa manera puede no ser obvia al principio a menos que usted sea Holandés
12. Ahora es mejor que nunca

    - Aunque nunca es a menudo mejor que ya

13. Si la implementación es dificil de explicar, es una mala idea.
14. Si la implementacion es fácil de explicar, puede que sea una buena idea
15. Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

Desde la versión 2.1.2, Python incluye estos puntos (en su versión original en inglés) como un huevo de pascua que se muestra al ejecutar

```
import this
```

A.2 Ámbito de nombres

Nota: O espacio de nombres

Cuando se ejecuta el intérprete de Python de forma interactiva o cuando se invoca para ejecutar un script, están diponibles un conjuto de funciones, clases, tipos de datos y excepciones. El conjunto de estos nombres se conoce como ámbito de nombres __builtin__ (*incorporado*) y en este encontramos: int(), char(), str(), object, Exception, TypeError, ValueError, True, False, range, map, zip, locals, globals, entre otras.

Pueden verse en el interprete interactivo mediante

```
>>> dir(__builtins__)
```

Un *ámbito de nombres* o *espacio de nombres* en Python es un mapeo de nombres a objetos. La mayoría de los ámbitos de nombres están implementados mediante diccionarios, but that's normally not noticeable in any way (except for performance), y podría cambiar en el futuro. Otros ejemplos de ámbitos de nombre son los nombres globales en un módulo; los valores locales en la invocación de una función y en cierta forma, los atributos de un objeto también forman un ámbito de nombres. Es importante tener en cuenta que no existe ninguna relación entre nombres de diferentes ámbitos de nobres.

Un *módulo* es un archivo con definiciones y sentencias en Python. El conjunto de todas las sentencias que no están indentadas, forman el ámbito de nombres global del modulo, las funciones, variables y clases que están suelen decirse que estan a *nivel módulo*. Los elementos que pueden ser parete del ámbito de nombres del módulo son:

- una definición de una función
- una vairable
- una expresión lambda
- el nombre de otro módulo importado

Un número, el llamado a función o el lanzado de una excepción no alteran el ámbito de nombres. Un módulo puede ser cargado desde el intérprete interactivo en el ámbito de nombres global o desde otro módulo (en su propio espacio de nombres) a través de la sentencia *import*

```
import os
```

En este caso, el intérprete buscará secuencialmente en el PYTHONPATH un módulo llamado **os**, en caso de encontrarlo, lo cargará y en el ámbito de nombres actual generará una referencia con el nombre *os*. A esta actividad se la llama importación y el interprete garantiza que un módulo se cargará de unicamente una sola vez, en otras palabras, existe una única instancia de un módulo. En este caso, si varios módulos importan a *os*, solo la primer ocurrencia realiza la importación efetiva. El resto de las sentencias import solamente generan referencias a la primera *instancia* de un módulo.

La sentencia *import* permite a su vez hacer una importación selectiva de símbolos de un módulo, por ejemplo:

```
from os import path
```

En este caso, import crea la instancia del módulo en la máquina virtual, pero no expone todo el contenido al ámbito de nombres local, sino que únicamente genera una referencia al símbolo *path*

El comodín * permite importar todos los símbolos definidos en un módulo al espacio de nombres local.

```
from os import *
```

Está técnica debe ser utilizada con cuidado, debido a que "contamina" el ámbito de nombres local. Para mitigar este problma, puede definirse una variable *__all__* al comienzo del archivo, con una tupla en la cual se definen los símbolos que se desean exportar. Suponendo que el siguiente código se encuentra en el módulo promedio:

```
__all__ = ('calcular_promedio', )

def calular_promedio(numeros):
    return suma (numeros) / len( numeros )
def suma(lista):
    return float( sum( lista ) )
```

Al realizar .. code-block:: python

    from promedio import *

El único símbolo que encontraremos en nuestro espacio de nombres será, *calcular_promedio*.

A.2.1 Sobrecarga de operadores

A.2.2 Funciones en Python

En python, una función se define de la sieguiente manera:

def funcion(argumento1, argumento2): ‘” Doc de la funcion ‘” print argumento1, argumento2

Python soporta lo que se ha dadon en llamar *empaquetado de argumentos* tanto en la definición como en la invocación de una función.

El empaquetado más simple es el del tipo lista, que consiste en la hablidad de una función de recibir argumentos variables en formato lista. Para usar esta caractrística es necesrio utilizar una signatura especial (agregando un asterisco).

```
def promedio(*lista):
    ''' Calcula promedio '''
    return sum(lista) / float(len(lista)) # utilizando builtins


    # Ejemplo de utilizacion
    >>> promedio(1, 2, 4, 5, 5)
    3.3999999999999999
```

Esta signatura de argumentos se puede combinar con argumentos posicionales.

```
def dividir_suma_por( numero, *lista ):
    return sum(lista) / numero
```

La capiacidad de recibir listas puede ser utilizada a la inversa. Es decir, conociendo que una función recibe una cantidad de argumentos, ‘‘acomodar” en una lista los argumentos que se desean utilizar.

```
# Utilizanodo la funcion promedio anterior
>>> lista = [1, 2, 4, 6, 7]
>>> promedio( *lista )
4.0
```

Módulos

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

Por ejemplo, consideremos el módulo `funciones.py`

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

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

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

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

Paquete

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

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

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

Módulo de expresiones regulares "re"

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

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

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

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

Metaprogramación mediante metaclases

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

Objetos *callables*

Un objeto *callable*

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

[PEP333] PEP *Python Enhancement Proposals* son documentos en los que se proponen mejoras para el lenguaje Python, publicados en el sitio oficial http://www.python.org.

[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/

[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

[DjangoDocsContentType2009] *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 Septiembre 2009, http://docs.djangoproject.com/en/dev/ref/models/instances/#the-pk-property

[MSFTJScript09] *JScript*, Microsoft Developer Network, último acceso Septiembre de 2009, http://msdn.microsoft.com/es-es/library/72bd815a%28VS.80%29.aspx

[ECMAScript09] *Especificación ECMA*, TC39 - ECMAScript (formerly TC39-TG1), último acceso de Agosto 09, http://www.ecma-international.org/memento/TC39.htm

[WikiJavascript09] *Javascript* http://es.wikipedia.org/wiki/JavaScript

[Firebug09] *Firebug*, Plugin de depuración integral, último acceso Septiembre 2009, http://getfirebug.com/

[StephenChapmanJS2009] *Javascript and XML*, Stephen Chapman, About.com, último acceso Agosto 2009, http://javascript.about.com/library/blxhtml.htm

[W3cCSS2009] *Guía breve de CSS*, W3C, español, último acceso Agosto 2009, http://www.w3c.es/divulgacion/guiasbreves/HojasEstilo

[ProtypeOrgSrc09] *Código Fuente de Prototype, versión 1.6.3*, Prototype.org, http://www.prototypejs.org/assets/2008/9/29/prototype-1.6.0.3.js

[StuartLangridgeClosures09] Stuart Langridge, *Secrets of JavaScript Closures*, último acceso Octubre 2009, http://www.kryogenix.org/code/browser/secrets-of-javascript-closures/

[MDCDOM09] Mozilla Developer Center, *DOM*, último acceso Agosto de 2009, https://developer.mozilla.org/en/DOM

[W3CDOM09] World Wide Web Consortium, *Document Object Model*, último acceso octubre 2009, http://www.w3.org/DOM/

[W3CDomLevels09] World Wide Web Consortium, *Niveles de Document Object Model*, último acceso octubre 2009, http://www.w3.org/DOM/DOMTR

[WIKIDOM09] Wikipedia, *DOM*, último acceso Octubre 2009, http://es.wikipedia.org/wiki/Document_Object_Model

[WIKIAJAX09] *AJAX*, Wikipedia, último acceso Octubre de 2009, http://es.wikipedia.org/wiki/AJAX

[JSONOrg2009] *JSON*, Sitio Oficial del Estándar, último acceso Octubre de 2009, http://json.org/

[JSONOrgJS09] *Utilización de JSON en Javascript*, Json.org, ultimo acceso Agosto de 2009, http://json.org/js.html

[SimonWillson24Ways09] http://24ways.org/2005/dont-be-eval

[MozillaMDCNativeJSON09] https://developer.mozilla.org/en/Using_JSON_in_Firefox

[IEBlogNativeJSON09] http://blogs.msdn.com/ie/archive/2008/09/10/native-json-in-ie8.aspx

[DaveWardEncosia2009] Dave Ward, Improving jQuery's JSON performance and security, Julio de 2009, http://encosia.com/2009/07/07/improving-jquery-json-performance-and-security/

[GGGears09] Sitio oficial para desarrolladores Google Gears, Google, ultimo acceso Agosto 2009, http://gears.google.com/

[BretTaylor09] Blog de Bret Taylor, último acceso Agosto de 2009, http://bret.appspot.com/

[IronRubyNet09] IronRuby, implementación de Ruby sobre .NET, ultimo acceso Septiembre 2009, http://www.ironruby.net/

[InforQDjangoIP09] InfoQ, *Django On IronPython*, último acceso Octubre 2009, http://www.infoq.com/news/2008/03/django-and-ironpython

[PythonMailistMay07] Lista Oficial sobre el lenguaje Python, *Silverlight, a new way for Python?*, ultimo acceso Septiembre de 2009, http://mail.python.org/pipermail/python-list/2007-May/610021.html

[MichaelFroodIP09] Michael Frood, Blog Oficial de Michael Frood, *explicación de como ejecutar IronPython sobre .Net*, http://www.voidspace.org.uk/ironpython/silverlight/index.shtml#id2

[PythonDocAPI09] Python.org, *Listado de Módulos de la API standard*, ultimo acceso Octubre 2009, http://docs.python.org/modindex.html

[MSDNSilverlightDOM09] Microsoft Developer Network, Silverlight Programming Models, XAML, and the HTML DOM, último acceso Octubre 2009 http://msdn.microsoft.com/en-us/library/cc838215%28VS.95%29.aspx

[IronPythonFAQ2009] Sitio oficial de IronPython, *Diferencias entre IronPython y CPython*, último acceso Septiembre 2009,

http://ironpython.codeplex.com/Wiki/View.aspx?title=IPy2.0.xCPyDifferences&referringTitle=Home

[SwOnCodeSlvlgth09] Switch On The Code, *Silverlight Tutorial - Interaction With The DOM*, ultimo acceso Octubre 2009 http://www.switchonthecode.com/tutorials/silverlight-tutorial-interaction-with-the-dom

[DinoEspositoSlvlght09] Dino Esposito, *Isolated Storage in Silverlight 2.0*, ultimo acceso Agsoto de 2009, http://www.ddj.com/windows/208300036

[AshishShettySlvlght09] Ashish Shetty, *Silverlight out-of-navegador apps: Local Data Store*, ultimo acceso Agosto 2009, http://nerddawg.blogspot.com/2009/04/silverlight-out-of-navegador-apps-local.html

[SteveLeeJs17Py09] Steve Lee, Open Source Eduspaces, *Mozilla's Javascript 1.7 includes some Python goodness*, http://eduspaces.net/stevelee/weblog/450964.html

[GuyonMoreePythonBraces09] Guyon Morée, *Pythonic Javascript, it's Python with braces!*, útlimo acceso Septiembre 2009, http://www.gumuz.nl/weblog/pythonic-javascript-its-python-braces/

[AtulVarma2009] Atul Varma, *Python For Javascript Programmers*, ultimo acceso Septiembre 2009, http://hg.toolness.com/python-for-js-programmers/raw-file/tip/PythonForJsProgrammers.html

[ScottLoganbillHTML5Gears09] Scott Loganbill, *How HTML 5 Is Already Changing the Web*, último acceso Septiembre 2009, http://www.webmonkey.com/blog/How_HTML_5_Is_Already_Changing_the_Web

[W3CHTML5OffWebApp09] World Wide Web Consortium, Apartado sobre Aplicaciones Web Desconectadas en el borrador sobre la especificación HTML5, último acceso Septiembre 2009 (Revision 1.2852), http://www.w3.org/TR/html5/desconectado.html#desconectado

[UrielKatzJStORM09] Uriel Katz, *Introducing JStORM*, último acceso Septiembre 2009, http://www.urielkatz.com/archive/detail/introducing-jstorm/

[GearsOnRails09] *Gears On Rails*, GoogleCode, ultimo acceso Septiembre 2009, http://code.google.com/p/gearsonrails/

[DjangoOffline09] *Django desconectado*, Google Code, ultimo acceso Septiembre 2009, http://code.google.com/p/django-offline/wiki/Goals

[NickCarterJazzModels09] Nick Carter, *Model - JazzRecord JavaScript ORM Documentation* último acceso Septiembre 2009, http://www.jazzrecord.org/docs/model#finders

[KrisKowal09] Kris Kowal, Proyecto module.js, útlimo acceso Septiembre 2009, http://modulesjs.com/

[YahooYUILoader09] YUI Team, Documentación del módulo YUI Loader, último acceso Septiembre 2009, http://developer.yahoo.com/yui/yuiloader/

[JamesDonaghuePeppyDocs] James Donaghue, *Documentación de Peppy*, último acceso Noviembre 2009, http://jamesdonaghue.com/static/peppy/docs/

[W3CCSelCSS309] W3C, Documentación de Selectores CSS de nivel 3, ultimo acceso Septiembre 2009, http://www.w3.org/TR/css3-selectors/

[AspenWebServer09] Lawrence Akka, Christopher Baus, Chris Beaven, Steven Brown, Chad Whitacre, *Servidor Web Aspen*, ultimo acceso Diciembre 2008, http://www.zetadev.com/software/aspen/

[DojoLibDjangoTmpl09] The Dojo Foundation, Documetnación sobre los templates de Django portados a Dojo, ultimo acceso Septiembre de 2009, http://www.dojotoolkit.org/book/dojo-book-0-9/part-5-dojox/dojox-dtl/

[DjangoNewFormsAdminBranch09] Brian Rosner, Django Trac, *The newforms-admin branch*, último acceso Octubre 2009, http://code.djangoproject.com/wiki/NewformsAdminBranch, http://code.djangoproject.com/changeset/7967

[JunctionDocsSync09] Steve Yen, *TrimPath Junction Syncrhonization*, http://trimpath.googlecode.com/svn/trunk/junction_docs/files/junction_doc_sync-txt.html

[W3CFormEncoding09] Dave Raggett, Arnaud Le Hors, Ian Jacobs, *Especificación de codificación de contenidos de formularios en HTML 4.1*, ultimo acceso Noviembre de 2009, http://www.w3.org/TR/html401/interact/forms.html#form-content-type

[GitDocs09] Sistema de Control de Versiones Git, último acceso Noviembre de 2009, http://git-scm.com/

[MercurialDocs09] Sistema de Control de Versiones Mercurial, último acceso Noviembre de 2009, http://mercurial.selenic.com/guide/

[BazaarDocs09] Martin Pool *Sitio Oficila de Sistema de Control de Versiones Bazaar*, último accseo Noviembre 2009, http://bazaar-vcs.org/en/

APÉNDICE B

Referencia sobre Django

B.1 Instalación de Django

Para realizar la instalación de la versión oficial más reciente se debe acceder a:

`http://www.djangoproject.com/download/.`

Django usa el método `distutils` (estándar de distribución de Python), que se utiliza de la siguiente manera:

```
$ Baja el tarball, que se llamará algo así como *Django-0.96.tar.gz*

$ tar xzvf Django-*.tar.gz

#. ''cd Django-*''

#. ''sudo python setup.py install''
```

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

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

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

B.2 Comandos del módulo manage

B.2.1 El comando syncdb

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

B.2.2 El comando runserver

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

B.2.3 El comando validate

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

B.3 Comandos de usuario

Django permite

Índice

X