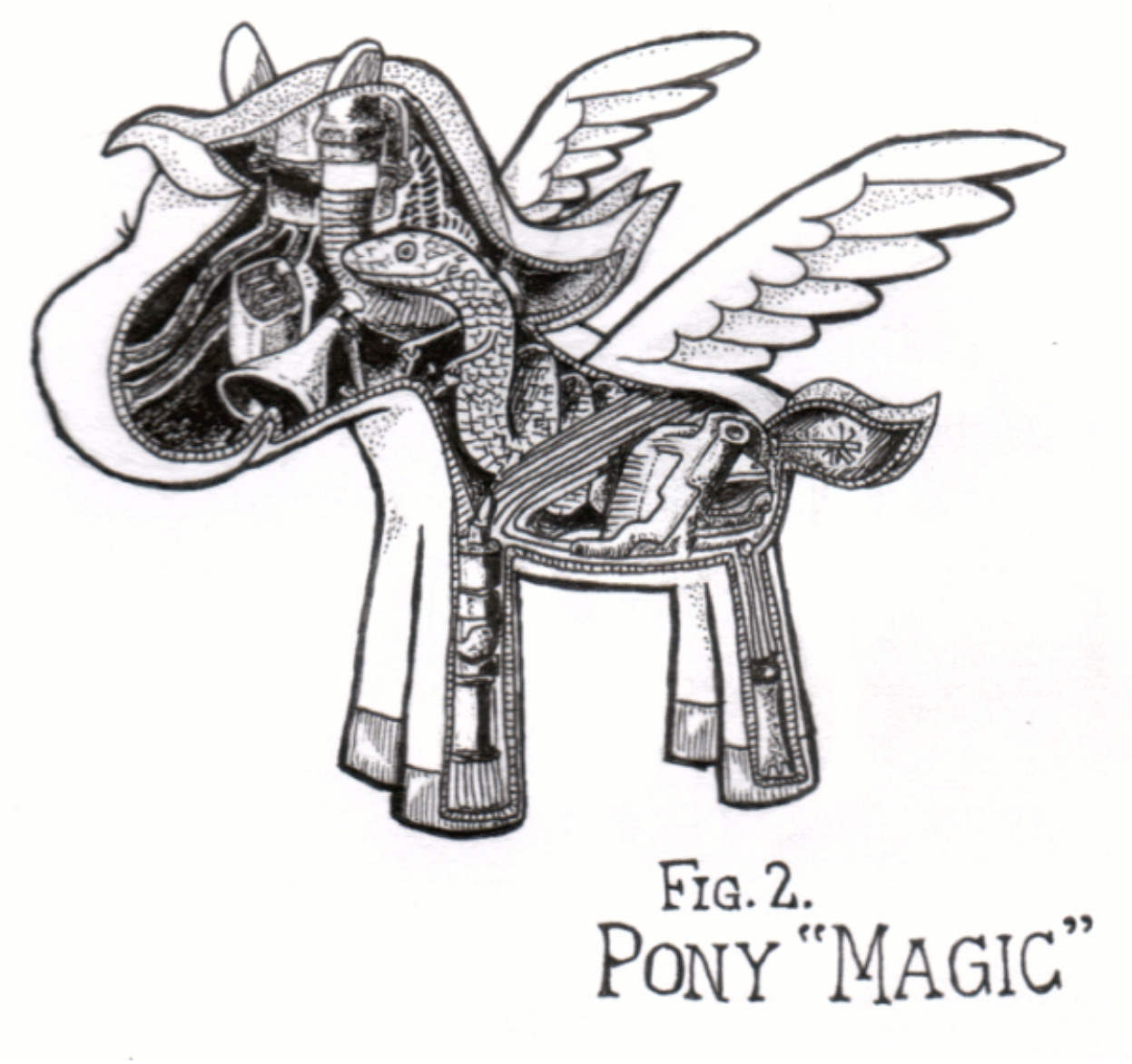

Aplicaciones Web Desconectadas

Release 1

Defossé, Nahuel; van Haaster, Diego Marcos

October 26, 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.).

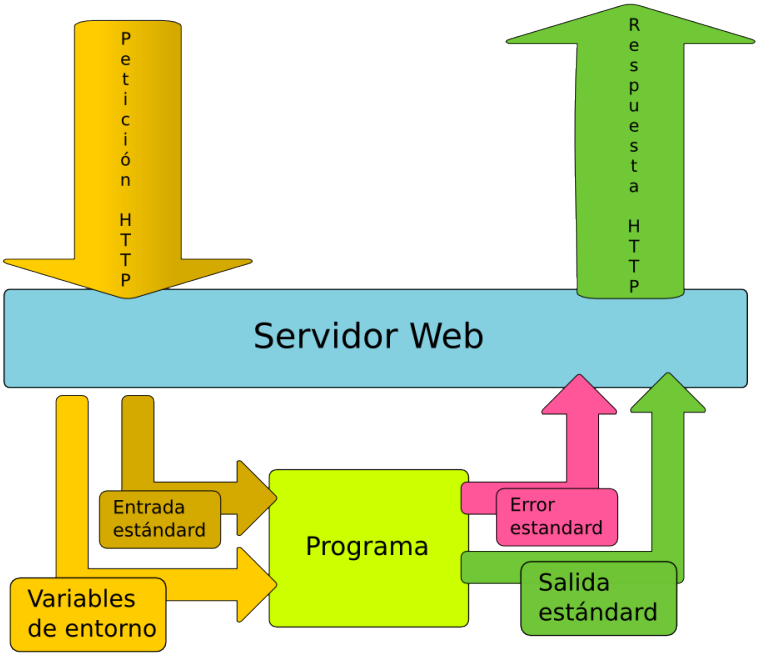

Figura 2.1: Proceso de una solicitud con CGI.

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.

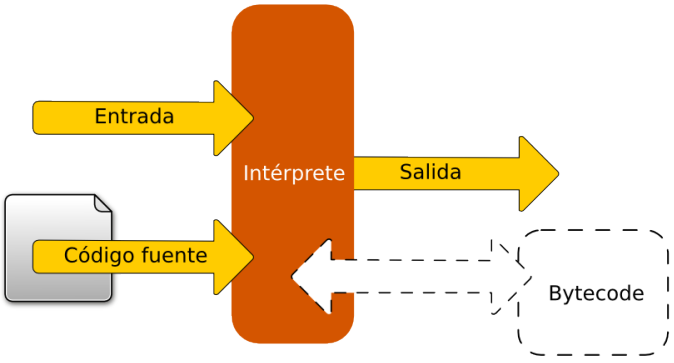

Figura 2.2: Lenguaje interpretado con máquina virtual.

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]

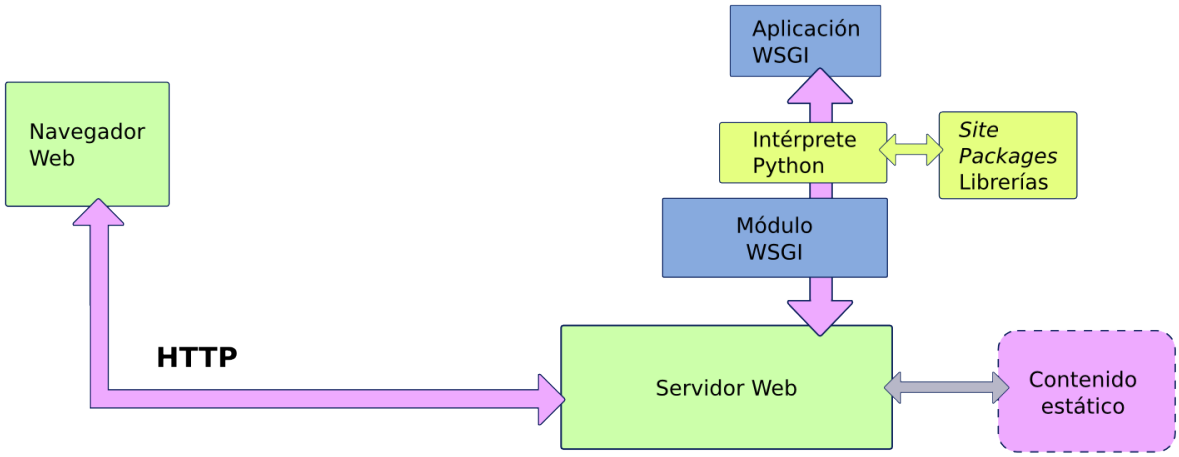

Figura 2.3: Esquema WSGI.

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.

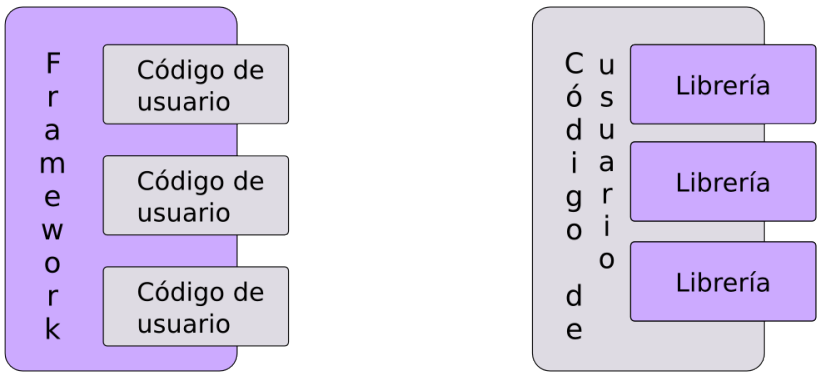

Figura 2.4: Framework vs Librería.

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.

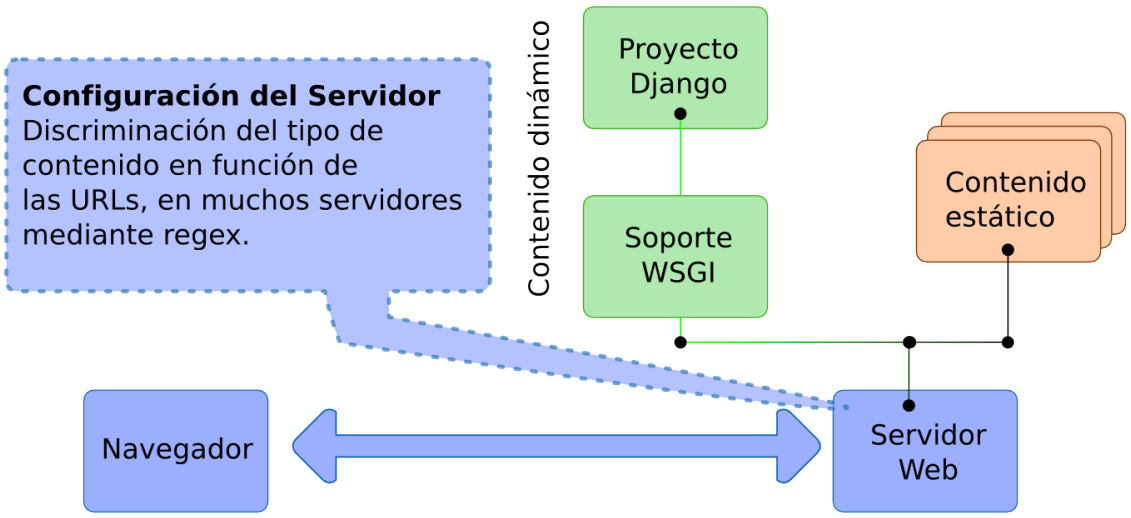

Figura 2.5: Puesta en producción de un proyecto en Django

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`.

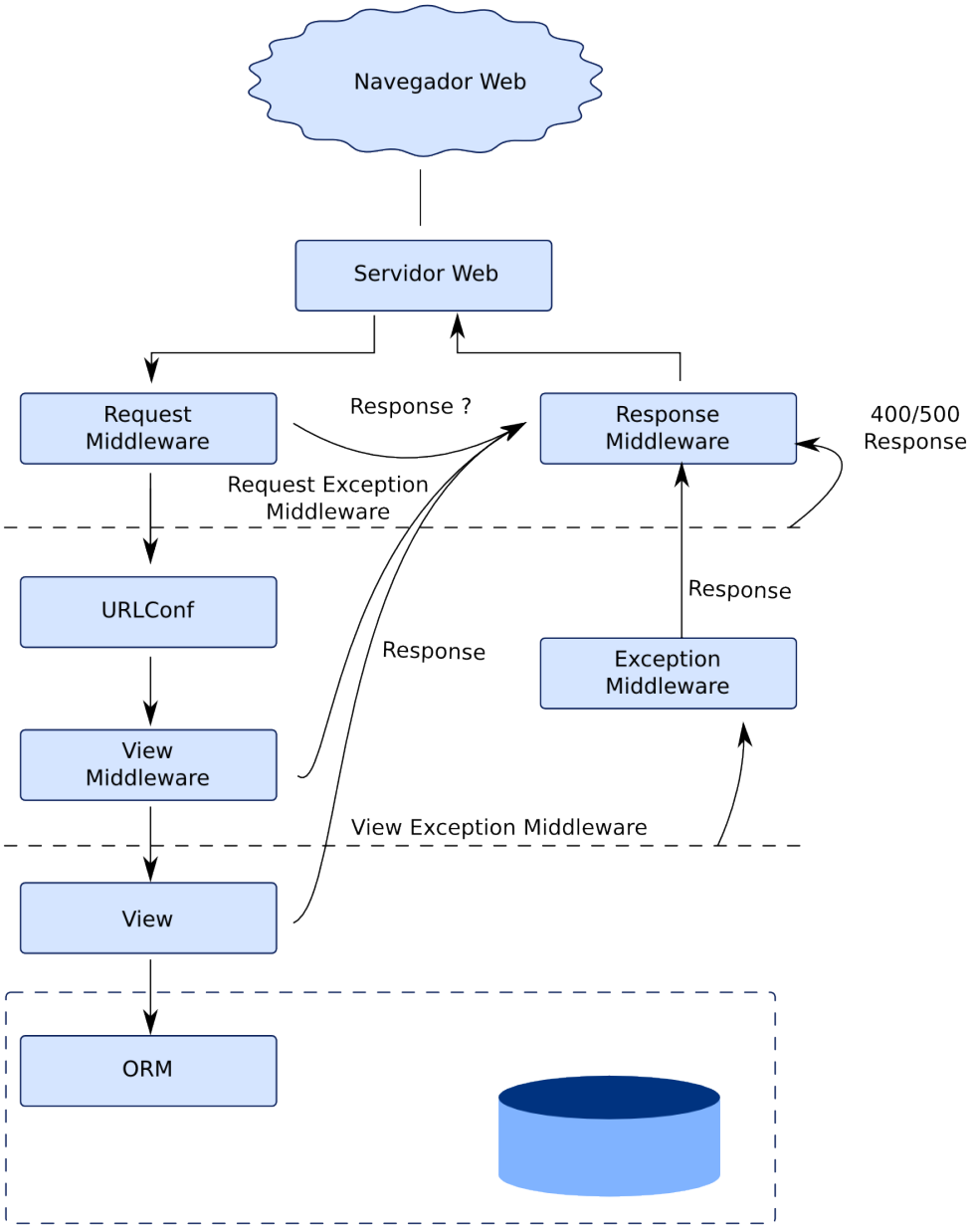

Figura 2.6: Procesamiento de un request con la interacción de los middlewares.

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.

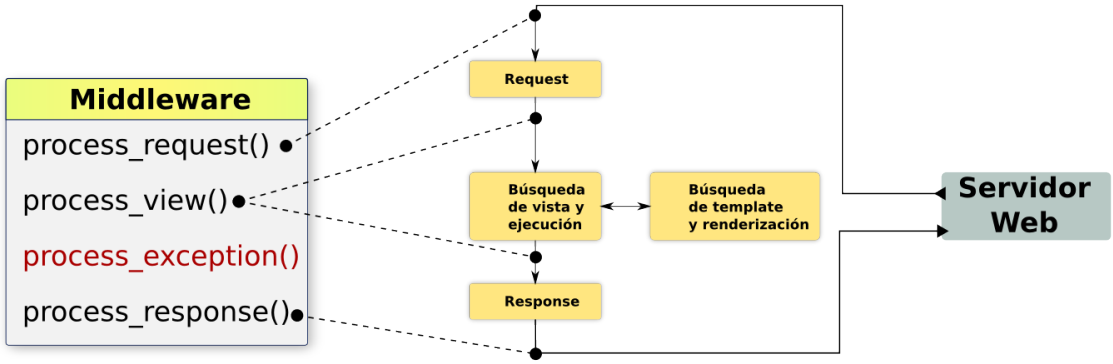

Figura 2.7: Interacción de los métodos de una clase de middleware en el proceso del `request`

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.

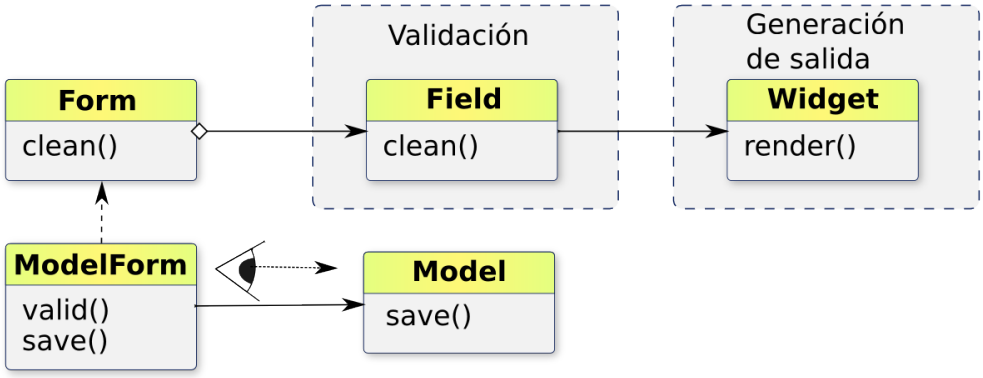

Figura 2.8: Diagrama de clases de un formulario en Django

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:

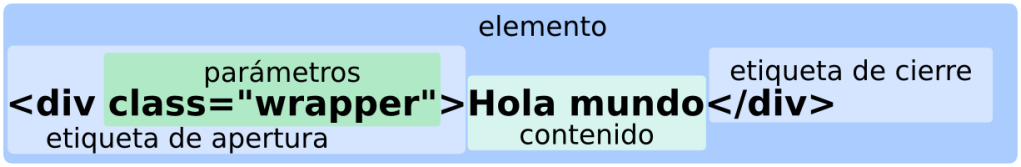

Figura 3.2: Formato de una etiqueta HTML.

```
<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.

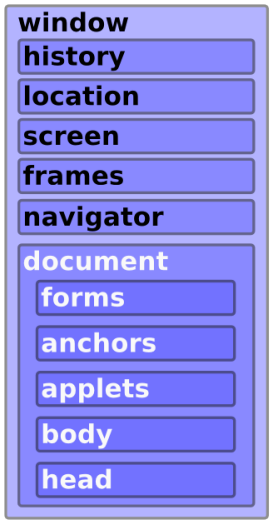

Figura 3.3: Modelo de objetos de DOM

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.

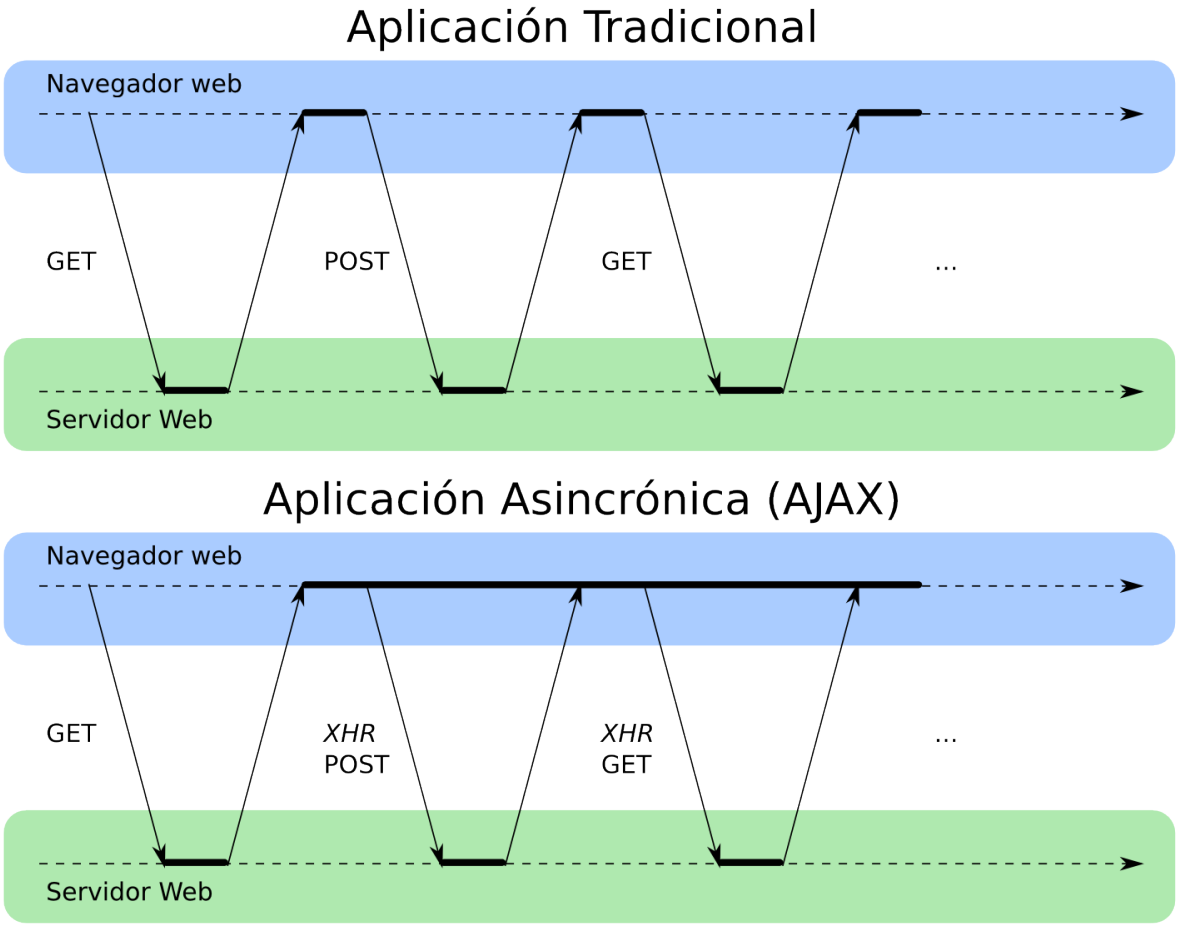

Figura 3.4: Compraración entre una aplicación tradicional y una asincrónica. La línea gruesa indica el ciclo de vida de la página en el cliente y las peticiones en el servidor.

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.

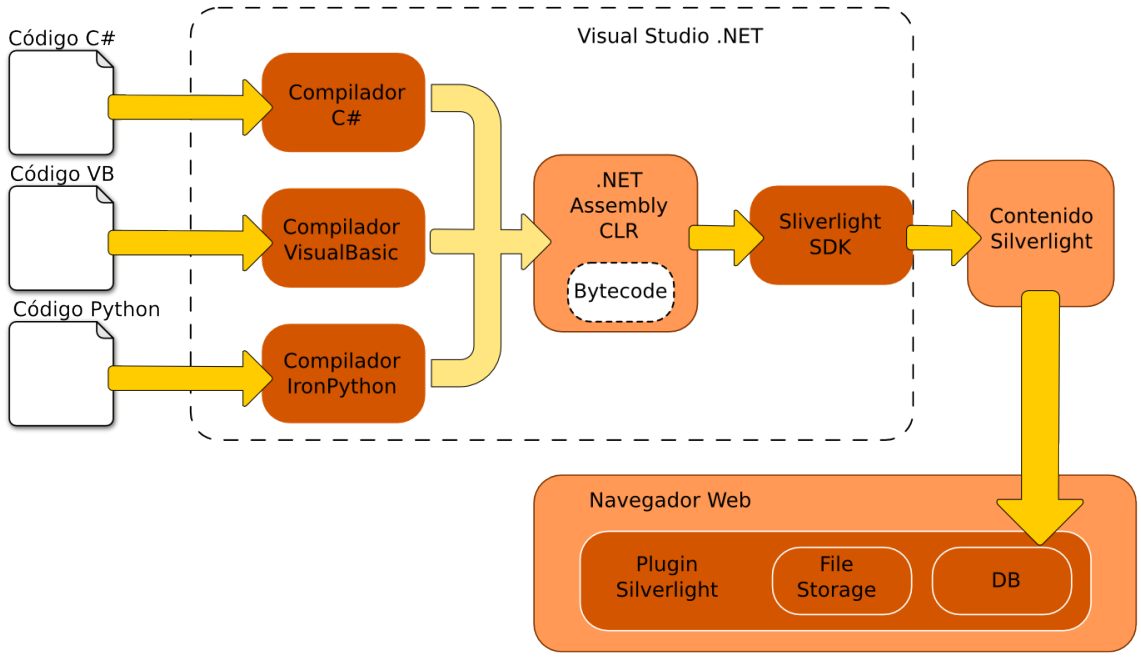

Figura 4.1: Esquema de desarrollo de Silverlight

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:

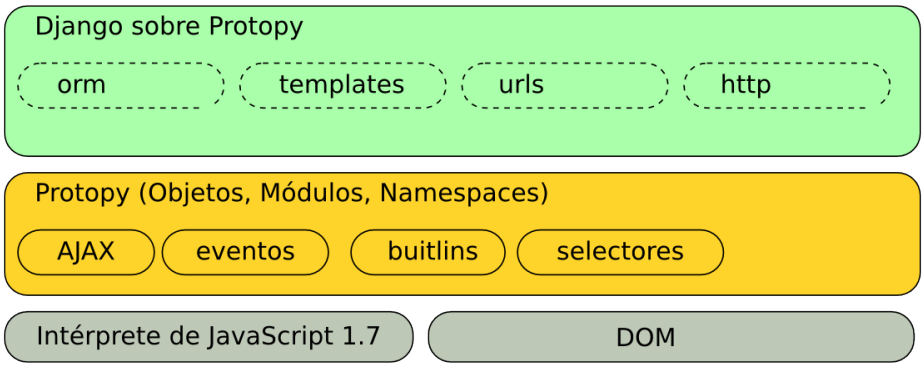

Figura 4.2: **Protopy**, Librería desarrollada en JavaScript para soporte de Python sobre JavaScript 1.7

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 implementa 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 una capa intermedia entre JavaScript y Django "desconectado". Una de las razones mas relevantes es emular la API estándar de Python.

Nota: No quedaron más que ideas de prototype en Protopy :P

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 bautizó en honor a sus dos "padres", por un lado *Portotype JavaScript Library* y el lenguaje de programación *Python*:

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

5.1 Objetivos de la Librería

- 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 cerca de 43000 líneas, sin contar las aplicaciones contribuidas (administración, GIS, data, 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.

- Orientación a Objetos "Pythonica"

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

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.

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 estos 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 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 búsqueda de elementos dentro de un documento de tres maneras: mediante un identificados, mediante un nombre de tag, o el acceso jerárquico tipo árbol (estas técnicas se pueden combinar).

  Los selectores CSS simplifican la tarea de seleccionar un conjunto de elementos que cumple 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 seleccionar todos 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, Portotype, jQuery, Dojo, Ext JS entre otras poseen este tipo de selector. Su utilización libra al desarrollador de incompatibilidades o implementaciones pobres de DOM, a la hora de interactuar con los elementos del documento.

5.1.1 Compatibilidad de Protopy

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

5.1.2 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.

```
<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 gerarquía de directorios */ruta/a/protopy/* en el ejemplo expuesto.

5.1.3 Argumentos de Inicialización

Dentro del 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 analizaran en el capítulo siguiente.

Organización del Código

Como se mencionó en el apartado teórico dedicado a JavaScript, la inclusión de código en un documento HTML se realiza mediante el tag `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, esta pasa a ser miembro del objeto `window`).

La carga de JavaScript mediante tags de inclusión es práctico para proyectos pequeños (donde se utiliza JavaScript para validación, enriquecimiento de formularios, accesibilidad) pero resulta limitado y poco mantenible cuando 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, donde la modularidad y la creación de ámbitos de nombres están íntimamente relacionados.

Nota: No, no la inclusión de js asincrónica viene de la mano de Dojo

En vez de cargar los recursos mediante la inclusión de tags, 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 agregó al ámbito global la función `require( "nombre_modulo" )` que recibe como cadena el nombre del módulo que se desea cargar y lo descarga anacrónicamente para luego evaluarlo un contexto aislado.

En Protopy, un módulo es un recurso JavaScript que publica explícitamente una interfase. 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()`. 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 jer rquicamente en directorios, a las cual 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, estos no incluye funcionalidad per se [2] la única función de estos es la de establecer estructura.

5.2 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 modulo como un objeto,

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

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

[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`)

- Obtener un símbolo de un modulo

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

- O se pueden importar todas las definiciones del módulo en el espacio de nombres del llamador.

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

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

5.3 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 modulo.

A continuación se presenta un fragmento de código que ejemplifica el uso de las dos funciones presentadas sobre un código similar al de listado en Python:

```
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 contar_palabras(linea) {
```

```
10      return linea.split(' ').length
11  }
12
13  publish({
14      promedio: promedio,
15      contar_palabras: contar_palabras
16  });
```

5.4 Módulos Nativos

Como se introdujo al principio del 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.

Nota: Describir de forma rápida los módulos principales

- builtin

  Este módulo cuenta con los tipos y funciones disponibles en el ámbito global ni bien se inicia el intérprete de Python. Está construida por los tipos básicos, las funciones como `filter`, `sum`, `map`, etc.

- dom

  Funciones de envoltura de DOM.

- 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.

- excepcionas

  Conjunto de excepciones.

- timer

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

5.5 Orientación a Objetos Basado en Clases

En el capítulo dedicado a las tecnologías del cliente se introdujo 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 intentar 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. 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 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. Su aparición posibilito nuevas formas de construir clases, similares a las que brinda la función homónima en Python, a la cual debe su nombre.

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

```
1  // Creación de un diccionario, que hereda del tipo object
2  var Dict = type('Dict', object, {
3      ...
4  });
5  // Creación de una clase que hereda de Dict, observar que es una lista ya
6  // se permite herencia múltiple.
7  var SortedDict = type('SortedDict', [ Dict ], {
8      __init__: function(object) {
9          this.keyOrder = (object && isinstance(object, SortedDict))? \
10             copy(object.keyOrder) : [];
11         super(Dict, this).__init__(object);
12     },
13     // Iterador, retorna pares clave, valor
14     __iter__: function() {
15         for each (var key in this.keyOrder) {
16             var value = this.get(key);
17             var pair = [key, value];
18             pair.key = key;
19             pair.value = value;
20             yield pair;
21         }
22     },
23     // Método utilizado para la copia profunda
24     __deepcopy__: function() {
25         var obj = new SortedDict();
26         for (var hash in this._key) {
27             obj._key[hash] = deepcopy(this._key[hash]);
28             obj._value[hash] = deepcopy(this._value[hash]);
```

```
29          }
30          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, seguidamente un arreglo con los tipo base y por último un arreglo asociativo con los atributos y métodos para los objetos de ese tipo.

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

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

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

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

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

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

5.6 Inicialización

En el ejemplo 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.

5.7 Métodos Especiales

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

- `__str__`

  Este método se llama cuando es necesario proveer de una representaci n en texto del objeto, JavaScript provee para este objetivo el método `toString`, pero por compatibilidad con Python y consistencia con la filosofía de Protopy de apegarse a la estructura de Python se recomienda utilizar el nombre `__str__`.

- `__iter__`

  Protopy se vale de versiones modernas de JavaScript para brindar este método, la función debe retornar un objeto que implemente el método `next`, los bucles `for` hacen esto automáticamente, por ejemplo:

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

- `__cmp__`

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

- `__len__`

  Se invoca con la llamada a la función *len*. La función incorporada len devuelve la longitud de un objeto. Funciona sobre cualquier objeto posea longitud (listas, diccionarios, etc.):

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

- `__copy__` y `__deepcopy__`

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

- `__json__` y `__html__`

  Métodos para sacralización (o *marshalling*) del objeto en HTML o JSON respectivamente.

Protopy tiene otros métodos especiales, general todos estos orientados a emular algún comportamiento de Python en JavaScript.

5.8 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` de la capacidad de hérnica similar a la de Python, ya lo que se intenta 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.

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.

Nota: Hasta acá

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.
..

> Bien, ya se tiene un mecanismo de herencia casi completo, solo falta ver como se accede a las funciones de tipos base cuando se esta redefiniendo un método, para esto

existe la función `super`, nuevamente y similar a Python esta función asocia un tipo con una instancia, con lo cual logra el objetivo de llamar a un método que se encuentre en un tipo base. Este comportamiento, es el equivalente, en JavaScript, al de llamar al método del tipo base con la función `apply` o `call`.

Para acceder a las funciones de tipos base cuando se esta redefiniendo un método existe la función `super`. De manera similar a Python, esta función

determina el tipo de la instancia, y mediante este 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__`).

5.9 Métodos y Atributos de Clase

Protopy contempla la definición de métodos y atributos de clase. Esta tarea serializa 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.

```
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++
    }
});
```

El ejemplo presentado define el tipo "Planeta", este tipo internamente lleva un contador que las instancias utilizan para numerarse en la construcción y una función de `reset` para reiniciar el contador. Dentro de los métodos de clase la palabra reservada `this`, hace referencia a la clase.

A continuación se ve como usar el clase.

```
// Creamos una instancia
>>> p = new Planeta('Tierra')
window.Planeta name=Tierra count=0 __name__=Planeta
// Volvemos a cero al conteador mediante el método de clase reset()
```

```
>>> Planeta.reset()
// Creamos la estrella Sol
>>> sol = new Planeta('Sol')
window.Planeta name=Sol count=0 __name__=Planeta __module__=window
// Creamos el planeta Mercurio
>>> mercurio = new Planeta('Mercurio')
window.Planeta name=Mercurio count=1 __name__=Planeta
// Creamos el planeta Venus
>>> venus = new Planeta('Venus')
window.Planeta name=Venus count=2 __name__=Planeta
// Creamos el planeta Tierra
>>> tierra = new Planeta('Tierra')
window.Planeta name=Tierra count=3 __name__=Planeta
```

5.10 Objetos Nativos

Los objetos nativos tienen como objetivo emular un tipo de datos de Python o realizar una adaptación de sintaxis.

- 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 de los cuales existe una única instancia de cada elemento como máximo. Permiten realizar operaciones de conjuntos como unión, intersección, diferencia, etc. Este tipo es una adaptación del tipo `set` de Python.

- Arguments

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

5.11 Manejo de DOM y Agregados a JavaScript

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

Además, basados en la idea de extender el `protopye` de los `HTMLElement` de la biblioteca de JavaScript Protoype, se agregaron funciones como 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, manejos de fecha, etc.

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

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

5.12 Envoltura de Google Gears

Protopy provee una interfase "pythonica" para la migración del framework, la construcción de aplicaciones y para el código referente a la interfase de usuario.

Como se mencionó en el capítulo anterior, para la migración del framework a un framework desconectado, son necesarios los componentes `DataBase` y `Local Server`. Gears provee una API que se integra al DOM. Esta fue envuelta en con una API más cercana a la filosofía Pythonica de Protopy, 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.

Dentro del módulo `sys`, se publica el objeto `gears` que almacena información como la disponibilidad de gears, que permisos le ha otorgado el usuario, que versión del plugin está disponible, el factory de Gears e incluso 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. Esta función es un recubrimiento del factory original de gears. Esta función, tras la creación de la instancia del tipo solicitado, analiza si se encuentra alguna funciona liad 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 acedan a Gears a través de `sys.gears`, su utilización es recomendada, ya que simplifica y uniforma la interfase entre Gears y la aplicación.

El desarrollo del framework implico extender algunos componentes de Gears:

- `desktop`

  El componente `desktop` de Gears permite interactuar con el escritorio del cliente. Se extendio 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 agrego funcionalidad uniformar el acceso a la base de datos por los módulos de Protopy y encapsular los `ResulSet` en cursores a los que se incorporo iteradores, registro de funciones para tipos de datos, etc.

5.13 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 sea pulsado 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, descomentando el camino por el cual desea realizar la depuración.

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

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

Protopy agrega un sistema de *logging* sobre las funciones antes mencionadas, permitiendo configuración sobre las salidas, el formateo y la prioridad, de manera similar a *log4j*. El desarrollador no necesita suprimir las sentencias de depuración del producto final, simplemente anula la salida en la configuración.

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.

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

```
// 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, {});
```

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

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': []
        }
    }
}
```

5.14 Interactuando con el servidor

Protopy provee una interfase de recubrimiento sobre las peticiones asincrónicas (AJAX) con el objeto de simplificar la utilización de `XMLHttpRequest`, trabajar con varias codificaciones de dato de manera segura (JSON), etc.

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

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

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. 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 halla completado, como 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 manejador se le pasa un objeto que representa la respuesta obtenida y que esta en relación con el evento capturando.

Otros manejadores que se pueden definir son:

- `onUninitialized`
- `onLoading`
- `onLoaded`
- `onInteractive`
- `onComplete`
- `onException`

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

De igual manera que el resto de las opciones, es posible agregar 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.

5.15 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 los almacenamientos de recursos locales, datos de modelos y llamadas a procedimiento remoto.

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.

5.16 RPC (Remote Procedure Call)

RPC consiste en la ejecución de una 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 fue incorporado a la librería Protopy como mecanismo para comunicación entre el framework desconectado y la aplicación en linea.

CAPÍTULO 6

Doff

En apartados teóricos se mencionó, entre otras cosas, que la creación de un framework generalmente surge de la identificación de objetos reusables en el desarrollo de software. Posteriormente los objetos identificados constituyen luego componentes que forman parte de una arquitectura. A estos se accede mediante una API especifica y se añade o modifica su funcionlaidad mediante configuración o extensión.

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

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

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

6.1 Arquitectura Inicial del Framework Desconectado

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

Como se mencionó en el capítulo introductorio, los compoentes que requerían una migración directa a JavaScript eran el ORM (`django.db.models`) y el sistema de templates

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

(`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 solo 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()‘‘s correspondientes en la consola.

Mientras que un cliente se encuentre sin conexion con el servidor web, es capaz de generar y almacenar datos usando su base de datos local. Al reestablecer la conexion con el servidor web, estos datos deben ser transimitos a la base de datos central para su actulizacion y posterior sincronizacion del resto de los clientes. Para plantear la tarea de sincronización es necesario contar con el framework funcionando desconecatdo, por lo que se postergó su analisis en esta etapa.

El siguiente componente en ser analizado y migrado fue el sistema de plantillas (`django.templates`). La librería de JavaScript Dojo provee un sistema de renderizado de planitllas basado en la sintaxis de Django [DojoLibDjangoTmpl09]. Sin embargo se optó por portar la propia de Django ya que la implementación de Dojo no está basada en el sistema de módulos de Protopy. Esta funcionalidad se situó en el módulo `doff.templates`.

Trás haber implementado los dos componentes más importantes de Django en JavaScript, se creó una aplicación genérica que tendría como objetivo conectar el fremework desconectado con el proyecto Django. Esta aplicación se bautizó `offline` e inicialmente contó con tres vistas. La primera fue un servidor estático de archivos, que permitía cargar Protopy en el navegador dentro del proyecto. Basados en esta vista y utilizando el almacenamiento del `LocalServer` provisto en Protopy en `sys.gears`, se obtuvo una página estática offline, que proveía el sistema de plantillas y de base de datos.

Las otras dos vistas estaban relacionadas entre si, una realizaba una búsqueda exahustiva de plantillas en el proyecto [2] y la otra servía las plantillas estáticamente. Con estos tres componentes se realizaron las perimers pruebas de aplicación desconectada.

A partir de estos componentes se comenzó a implementar el proyecto desconectado como un conjunto de aplicaciones que se definían en el módulo `settings` de el proyecto Django en linea. Las aplicaciones desconectadas consistían en paquetes de Protopy con los módulos `views.js` y `models.js`. Las vistas se ejecutaban de manera manual ya que el sistema de URLs no se pudo traducir de manera automática debido a que no se cuenta con una implementación de expresiones regulares con recuperación de grupos nombrados y los elementos `HTTPRequest` requerían una implementación adecuada a la ausencia de transporte HTTP.

El proyecto desconectado debía pasar por una etapa de instalación, en la cual se descargaba el framework y el proyecto en el almacencamiento local y se creaban las tablas de la base de datos.

Los siguientes apartados tratan sobre las implementaciones de la API del ORM

[2] Utilizando la constate `TEMPLATE_DIRS` definida en el módulo `settings` del proyecto.

6.1.1 Modelos

Doff intenta implementar la misma API que Django, realizando las adaptaciones a la sintaxis de JavaScript y la librería Protopy.

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

A continuación se ejemplifica la implementación del ORM basado en el sistema de clases y `doff.models` del modelo Libro - Autor - Editor:

```
.. code-block:: javascript
```

```
// 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 es representado por un clase de Protopy que extiende a `doff.db.models.model.Model`.

Cada modelo se corresponde con una tabla única de la base de datos, y cada atributo de un modelo con una columna en esa tabla. El nombre de atributo corresponde al nombre de columna, y el tipo

de campo corresponde al tipo de columna de la base de datos. Por ejemplo, el modelo `Publisher` es equivalente a la siguiente tabla:

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

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

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

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

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

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

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

6.2 Acceso básico a datos

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

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

```
>>> print(array(publisher_list));
[<Publisher: Publisher object>]
```

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

6.3 Insertando y actualizando datos

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

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

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

6.4 Seleccionar objetos

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

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

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

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

6.5 Filtrar datos

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

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

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

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

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

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

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

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

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

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

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

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

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

6.6 Obteniendo objetos individuales

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

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

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

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

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

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

6.7 Ordenando datos

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

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

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

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

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

Podemos ordenar por cualquier campo que deseemos:

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

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

y por múltiples campos:

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

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

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

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

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

6.8 Encadenando búsquedas

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

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

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

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

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

6.9 Rebanando datos

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

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

Esto se traduce, someramente, a:

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

6.10 Eliminando objetos

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

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

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

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

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

6.10.1 Plantillas

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

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

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

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

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

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

6.11 Creación de objetos Template

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

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

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

- Bloques de etiquetas inválidos
- Argumentos inválidos de una etiqueta válida
- Filtros inválidos

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

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

6.12 Renderizar una plantilla

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

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

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

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

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

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

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

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

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

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

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

6.13 Cargador de plantillas

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

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

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

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

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

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

6.13.1 Formularios

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

Autor invitado: Simon Willison

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

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

6.13.2 Búsquedas

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

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

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

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

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

Aquí han surgido algunas cosas que todavía no vimos. La primera, ese `request.GET`. Así es cómo accedes a los datos del GET desde Django; Los datos del POST se acceden de manera

similar, a través de un objeto llamado `request.POST`. Estos objetos se comportan exactamente como los diccionarios estándar de Python, y tienen además otras capacidades, que se cubren en el apéndice H.

Así que la línea:

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

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

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

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

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

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

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

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

    {% if results %}
      <ul>
```

```
      {% for book in results %}
        <li>{{ book|escape }}</li>
      {% endfor %}
      </ul>
    {% else %}
      <p>No books found</p>
    {% endif %}
  {% endif %}
</body>
</html>
```

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

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

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

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

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

6.13.3 El "formulario perfecto"

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

- Debería pedirle al usuario cierta información, obviamente. La accesibilidad y la usabilidad importan aquí. Así que es importante el uso inteligente del elemento `<label>` de HTML, y también lo es proporcionar ayuda contextual útil.

- Los datos suministrados deberían ser sometidos a una validación extensiva. La regla de oro para la seguridad de una aplicación web es "nunca confíes en la información que ingresa". Así que la validación es esencial.
- Si el usuario ha cometido algún error, el formulario debería volver a mostrarse, junto a mensajes de error detallados e informativos. Los campos deberían rellenarse con los datos previamente suministrados, para evitarle al usuario tener que volver a tipear todo nuevamente.
- El formulario debería volver a mostrarse una y otra vez, hasta que todos los campos se hayan rellenado correctamente.

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

6.13.4 Creación de un formulario para comentarios

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

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

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

```
from django import newforms as forms

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

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

Un formulario de Django es una subclase de `django.newforms.Form`, tal como un modelo de Django es una subclase de `django.db.models.Model`. El módulo `django.newforms` también contiene cierta cantidad de clases `Field` para los campos. Una lista completa de éstas últimas se encuentra disponible en la documentación de Django, en http://www.djangoproject.com/documentation/0.96/newforms/.

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

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

y en `contact.html`:

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

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

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

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

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

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

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

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

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

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

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

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

6.13.5 Procesamiento de los datos suministrados

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

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

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

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

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

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

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

```
from django.core.mail import send_mail

# ...

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

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

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

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

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

6.13.6 Nuestras propias reglas de validación

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

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

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

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

    def clean_message(self):
        message = self.clean_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

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

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

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

6.13.7 Una presentación personalizada

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

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

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

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

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

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

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

En caso de que hubieran errores de validación, se agrega la clase "errors" al `<div>` contenedor y se muestran los errores en una lista ordenada.

6.13.8 Creando formularios a partir de Modelos

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

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

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

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

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

PublisherForm = form_for_model(Publisher)
```

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

```
from forms import PublisherForm

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

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

Ahí se muestra un atajo más. Dado que los formularios derivados de modelos se emplean a menudo para guardar nuevas instancias del modelo en la base de datos, la clase del formulario creada por `form_for_model` incluye un conveniente método `save()`. Este método trata con el uso

común; pero puedes ignorarlo si deseas hacer algo más que tenga que ver con los datos suministrados.

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

CAPÍTULO 7

La aplicación

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

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

7.1 Definición de un proyecto en el cliente

Nota: hacer la referencia al capítulo de django

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

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

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

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

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

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

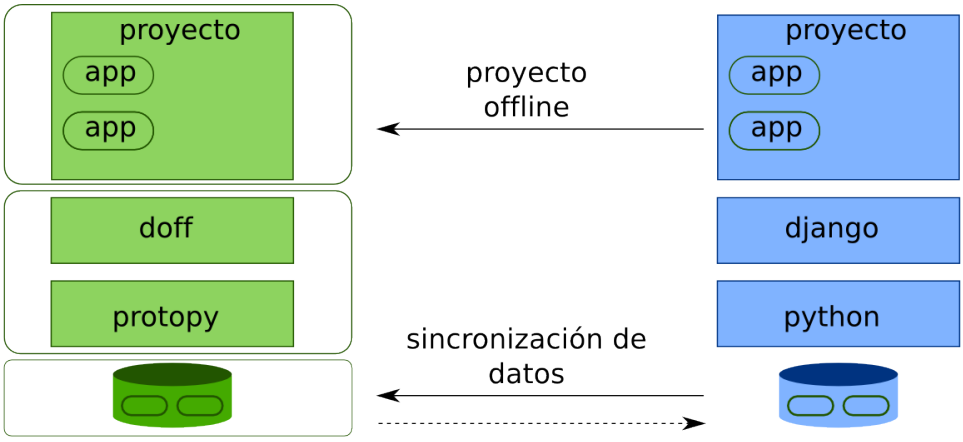

Figura 7.1: Análisis inicial de elementos a transferir para lograr una versión offline de un proyecto

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

7.2 Módulo de vistas y urls en una aplicación offline

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

7.3 Bootstrap

7.4 Transferencia de los modelos

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

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

7.5 Sitio Remoto

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

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

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

7.6 Ciclo de Trabajo

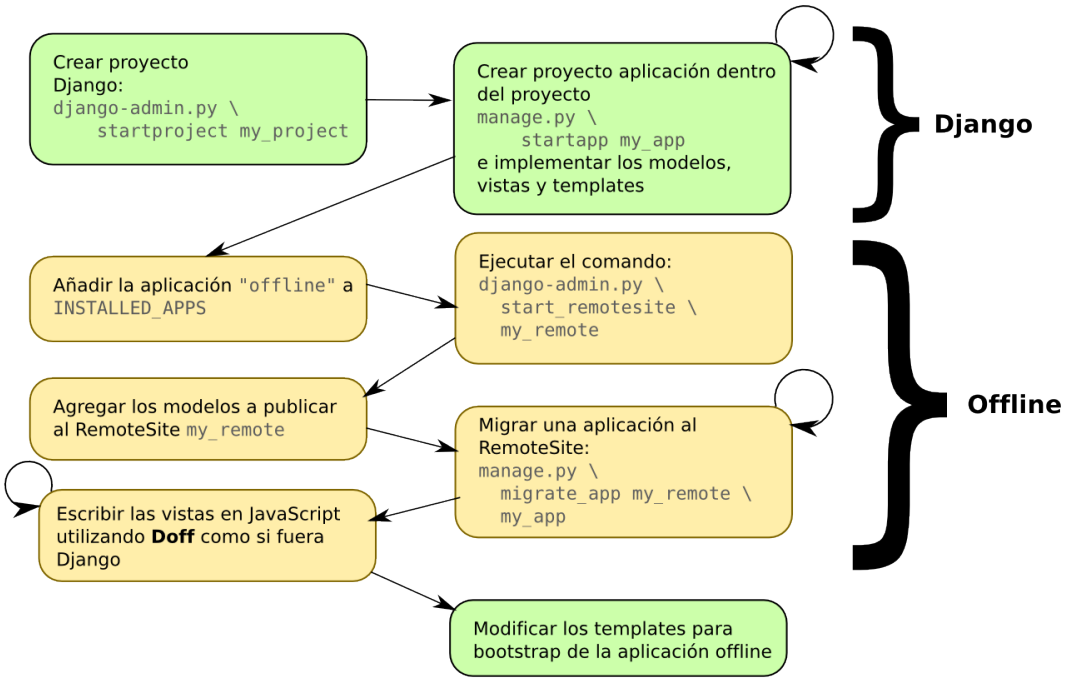

Figura 7.2: Esquema de trabajo con la aplicación offline

CAPÍTULO 8

Sincronizacion

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

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

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

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

8.1 Sincronización simple de servidor a cliente

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

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

Figura 8.1: Esquema de sincronización simple

8.2 Identificación de instancias en el servidor

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

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

8.2.1 Modelos

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

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

CAPÍTULO 9

Conclusiones y lineas futuras

9.1 Conclusiones

El desarrollo consistió en el siguiente diagrama:

9.2 Lineas futuras

9.2.1 Sitio de administración

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

9.2.2 Historial de navegación

9.2.3 Workers con soporte para Javascript 1.7

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

CAPÍTULO 10

Glossary

.net Plataforma de desarrollo creada por Microsoft.

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

BSD ve ese de

CGI Common Gateway Interfase

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

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

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

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

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

HTML Lenguaje de hipertexto basado en etiquetas de marcado

HTTP Hyper Text Transfer Protocol

i18n La internacionalización es el proceso de diseñar software de manera tal que pueda adaptarse a diferentes idiomas y regiones sin la necesidad de realizar cambios de ingeniería ni en el código. La localización es el proceso de adaptar el software para una región específica mediante la adición de componentes específicos de un locale y la traducción de los textos,

por lo que también se le puede denominar regionalización. No obstante la traducción literal del inglés es la más extendida.

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

Killer App Aplicación que populariza un lenguaje

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

model Models store your application's data.

MTV hola

MVC Model-view-controller; a software pattern.

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

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

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

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

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

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

Script Porgrama escrito en un lenguaje interpretado

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

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

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

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

URL Localizador universal de recursos, especificada en la RFC 2396

view A function responsible for rending a page.

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

Bibliografía

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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/

[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/

Índice

X