



FIG. 2.  
PONY "MAGIC"

## **Sistemas Web Desconectados**

*Release 1*

**van Haaster, Diego Marcos; Defossé, Nahuel**

August 29, 2009



---

# Índice general

---

<b>I</b>	<b>Introduccion</b>	<b>3</b>
1.	Motivación	5
2.	Objetivos	7
3.	Alcance	9
<b>II</b>	<b>Tecnologías del servidor</b>	<b>11</b>
4.	Generación dinámica de contenido	13
4.1.	Servidor Web . . . . .	13
4.2.	CGI . . . . .	13
4.3.	Lenguajes de programación para la web . . . . .	15
4.4.	Python . . . . .	17
5.	Frameworks Web	19
5.1.	Frameworks . . . . .	19
5.2.	Model View Controler . . . . .	20
5.3.	Mapeador Objeto-Relacional . . . . .	21
6.	Django	23
6.1.	Introducción . . . . .	23
6.2.	Estructuración de un proyecto en Django . . . . .	24
6.3.	Mapeando URLs a Vistas . . . . .	25
6.4.	El sistema de plantillas . . . . .	25
6.5.	Estructura de una aplicación Django . . . . .	26
6.6.	El ciclo de una petición . . . . .	26
6.7.	Interactuar con una base de datos . . . . .	28
6.8.	Modelos . . . . .	28
<b>III</b>	<b>Tecnologías del cliente</b>	<b>31</b>
7.	Estructura de un navegador	33

7.1. Navegador Web . . . . .	33
7.2. HTML . . . . .	33
7.3. CSS . . . . .	33
7.4. JavaScript . . . . .	33
<b>8. Evolución del JavaScript</b>	<b>35</b>
<b>9. Google Gears</b>	<b>37</b>
9.1. Introducción . . . . .	37
<b>IV Desarrollo</b>	<b>39</b>
<b>10. Una biblioteca en JavaScript</b>	<b>43</b>
10.1. Protopy . . . . .	44
10.2. Organizando el código . . . . .	44
10.3. Creando tipos de objeto . . . . .	46
10.4. Extendiendo JavaScript . . . . .	49
10.5. Extendiendo DOM . . . . .	49
10.6. Envolviendo a gears . . . . .	49
10.7. Auditando el código . . . . .	50
10.8. Interactuando con el servidor . . . . .	50
10.9. Soporte para json . . . . .	50
10.10. Ejecutando código remoto . . . . .	51
<b>V Conclusiones y líneas futuras</b>	<b>53</b>
<b>11. Conclusiones</b>	<b>55</b>
<b>12. Líneas futuras</b>	<b>57</b>
12.1. Sitio de administración . . . . .	57
12.2. Historial de navegación . . . . .	57
12.3. Workers con soporte para Javascript 1.7 . . . . .	57
<b>VI Glosario</b>	<b>59</b>
<b>VII Índices, glosario y tablas</b>	<b>63</b>
<b>Bibliografía</b>	<b>67</b>
<b>Índice</b>	<b>69</b>

Índice:



## **Parte I**

# **Introduccion**





---

# Motivación

---

Hoy más que ayer, pero seguramente menos que mañana, Internet es “la red de redes”. El alto contenido de información Hoy en día Internet supone más que un medio para obtener información, su constante expansión a convertido a esta red en un terreno muy atractivo para la implementación de sistemas de información.

Las aplicaciones web son populares debido a lo práctico que resulta el navegador web como cliente de acceso a las mismas. También resulta fácil actualizar y mantener aplicaciones web sin distribuir e instalar software a miles de usuarios potenciales. En la actualidad, existe una gran oferta de frameworks web para facilitar el desarrollo de aplicaciones web. Una ventaja significativa de las aplicaciones web es que funcionan independientemente de la versión del sistema operativo instalado en el cliente. En vez de crear clientes para los múltiples sistemas operativos, la aplicación web se escribe una vez y se ejecuta igual en todas partes. Las aplicaciones web tienen ciertas limitaciones en las funcionalidades que ofrecen al usuario. Hay funcionalidades comunes en las aplicaciones de escritorio, como dibujar en la pantalla o arrastrar y soltar, que no están soportadas por las tecnologías web estándar. Los desarrolladores web, generalmente, utilizan lenguajes interpretados o script en el lado del cliente para añadir más funcionalidades, especialmente para ofrecer una experiencia interactiva que no requiera recargar la página cada vez. Recientemente se han desarrollado tecnologías para coordinar estos lenguajes con tecnologías en el lado del servidor. Los sistemas operativos actuales de propósito general cuentan con un navegador web, con posibilidades de acceso a bases de datos y almacenamiento de código y recursos. La web, en el ámbito del software, es 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 (Consortio World Wide Web). La mayor parte de la web está soportada sobre sistemas operativos y software de servidor que se rigen bajo licencias OpenSource1 (Apache, BIND, Linux, OpenBSD, FreeBSD). Los lenguajes con los que son desarrolladas las aplicaciones web son generalmente OpenSource, como e 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.



---

# Objetivos

---

Podemos decir que las aplicaciones tradicionales, que no hacen uso de la web, son más robustas ya que no dependen de una conexión. Por lo tanto, sería deseable poder dotar a las aplicaciones web de la capacidad de trabajar cuando no cuentan con conexión. Si bien los elementos necesarios para llevar a cabo esta tarea están disponibles actualmente, no están contemplados en los diseños de los frameworks web. Es decir, cuando una determinada aplicación web debe ser transportada al cliente, es necesario escribir el código de soporte específico para esa aplicación. Un framework no constituye un producto per sé, sino una plataforma sobre la cual construir aplicaciones. Consideramos que sería beneficioso aportar una extensión a un framework web OpenSource que brinde facilidades para transportar las aplicaciones web, basadas en éste, al cliente de manera que la aplicación que haga uso de nuestra extensión pueda ser ejecutada a posteriori en el navegador en el cual ha sido descargada. El framework web será elegido tras un estudio de las características que consideramos más importantes para el desarrollo veloz, como la calidad del mapeador de objetos (entre las características más importantes de éste buscaremos eficiencia en las consultas a la base de datos, ejecución demorada para encadenamiento de consultas, implementación de herencia, baja carga de configuración), la simplicidad para enlazar url's a funciones controladoras, extensibilidad del sistema de escritura de plantillas. Buscaremos frameworks que permitan la ejecución transversal de cierto tipo de funciones, para ejecutar tareas como compresión de salida, sustitución de patrones de texto, caché, control de acceso, etc.

La World Wide Web, o web, durante los últimos años ha ganado terreno como plataforma para aplicaciones del variado tipo. Diversas tecnologías fueron formuladas para convertir el escenario inicial, donde la web se limitaba a ser una gran colección de documentos enlazados (hipertexto), para llegar a ser...

Vamos a realizar un breve análisis sobre las tecnologías que son utilizadas en la web.

Luego un análisis de las tecnologías del cliente, haciendo hincapié en ...



---

# Alcance

---

Aca ponemos hasta donde nos vamos a llegar.



## **Parte II**

# **Tecnologías del servidor**





---

# Generación dinámica de contenido

---

**Nota:** Otros títulos

- De lo estático a lo dinámico (*previo*)

**Nota:** algo de html quiza

**Nota:** titulo en revisión

## 4.1 Servidor Web

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

El contenido que es enviado al cliente puede ser de origen *estático* o *dinámico*. El contenido estático es aquél que proviene desde un archivo en el sistema de archivos sin ninguna modificación. El contenido dinámico en contraposición al *contenido estático* es generado por algún programa, un *script* o algún tipo de API invocada por el web server, como SSI, *CGI*, *SCGI*, *FastCGI*, *JSP*, *ColdFusion*, *NSAPI* o *ISAPI*.

El cliente web accede a los recursos del web server mediante una dirección de recurso, u *URL*.

## 4.2 CGI

CGI <sup>1</sup> surge como un estándar de comunicación, o :term:API entre un servidor web y un programa de usuario. Al permitir ejecutar un proceso, CGI permite generar contenido de manera dinámica a través de un servidor web. Es decir, el usuario puede recuperar datos de un programa que se ejecuta en el equipo en el que reside el servidor web, donde la salida puede ser un documento HTML entendible para el navegador, o cualquier otro tipo de archivo, como imágenes, contenido multimedial, sonidos, etc.

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

Los mecanismos de comunicación de entrada son las variables de entorno y la entrada estándar, mientras que para la salida se utiliza la salida estándar del proceso.

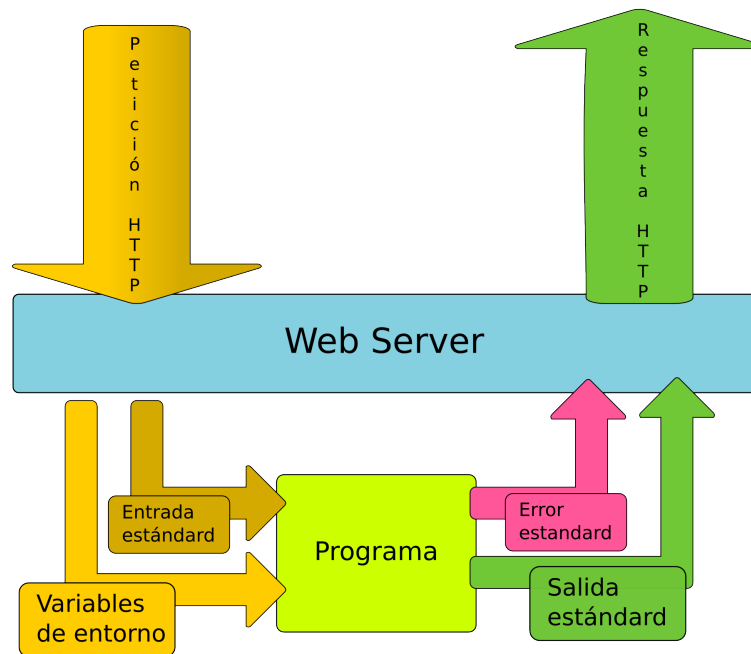
---

<sup>1</sup> A veces traducido como pasarela común de acceso.

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

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

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



Dentro de las variables de entorno, la Wikipedia [\[WikiCGI2009\]](#) menciona:

- **QUERY\_STRING** Es la cadena de entrada del CGI cuando se utiliza el método GET sustituyendo algunos símbolos especiales por otros. Cada elemento se envía como una pareja Variable=Valor. Si se utiliza el método POST esta variable de entorno está vacía.
- **CONTENT\_TYPE** Tipo MIME de los datos enviados al CGI mediante POST. Con GET está vacía. Un valor típico para esta variable es: Application/X-www-form-urlencoded.
- **CONTENT\_LENGTH** Longitud en bytes de los datos enviados al CGI utilizando el método POST. Con GET está vacía.
- **PATH\_INFO** Información adicional de la ruta (el “path”) tal y como llega al servidor en el URL.
- **REQUEST\_METHOD** Nombre del método (GET o POST) utilizado para invocar al CGI.
- **SCRIPT\_NAME** Nombre del CGI invocado.
- **SERVER\_PORT** Puerto por el que el servidor recibe la conexión.
- **SERVER\_PROTOCOL** Nombre y versión del protocolo en uso. (Ej.: HTTP/1.0 o 1.1)

Variables de entorno que se intercambian de servidor a CGI:

- **SERVER\_SOFTWARE** Nombre y versión del software servidor de www.
- **SERVER\_NAME** Nombre del servidor.

- **GATEWAY\_INTERFACE** Nombre y versión de la interfaz de comunicación entre servidor y aplicaciones CGI/1.12

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

**Nota:** Faltan referencias sobre la popularidad de los lenguajes

Históricamente las aplicaciones CGI han sido escritas en lenguajes interpretados, siendo muy popular Perl y más recientemente el lenguaje PHP. En los *lenguajes interpretados*, el código ejecutable es texto plano, por lo que puede ser editado en una terminal directamente en el servidor. Otra ventaja importante de ser texto plano, es que es más sencillo de mantener con alguna herramienta de :term:SCM.

## 4.3 Lenguajes de programación para la web

**Nota:** Acá introducimos los conceptos de

- Propósito General vs DSL
- Compilado vs Interpretado, -> VMs
- Dinámico vs Estático
- Plataforma Específica vs Multiplataforma
- Programación de sistema y programación de aplicaciones

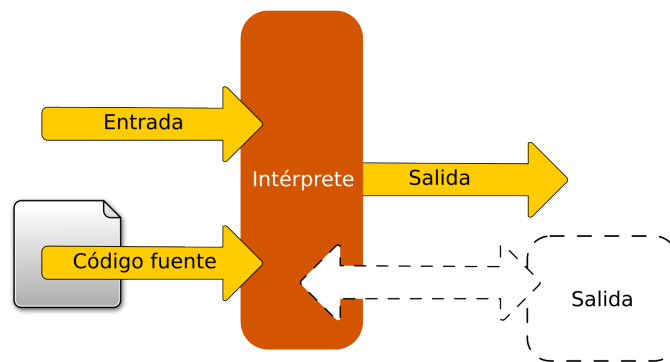
Una de las clasificaciones más generales que se suelen realizar sobre los lenguajes de programación es identificar su objetivo. Los lenguajes de programación de *propósito general* están orientados a resolver cualquier tipo de problema, mientras que los lenguajes de *propósito específico* o *DSL* están enfocados en resolver un tipo de problemas de manera más eficaz. Un ejemplo muy popular de DSL es la planilla de cálculo Microsoft Excel [DavidPollak2006].

Además de la división entre lenguaje de propósito general y DSL, existen otras clasificaciones de interés en el estudio de los lenguajes de programación populares en la Web, como la división entre lenguajes interpretados y compilados.

Un lenguaje de programación interpretado es aquel en el cual los programas son ejecutados por un intérprete, en vez de realizarse una traducción a lenguaje máquina, conocido como proceso de compilación.

En teoría cualquier lenguaje de programación podría ser compilado o interpretado.

En un lenguaje de programación interpretado se puede decir que el código fuente es el código ejecutable (a través del intérprete). Para llegar a la ejecución de un programa escrito en un lenguaje compilado, se debe pasar generalmente por dos etapas, una de compilación, donde se traducen las sentencias del lenguaje por código máquina y otra de enlace, donde se ensambla el código objeto resultado de la compilación y se resuelven los enlaces entre los diferentes módulos compilados.



Existe un mecanismo intermedio de ejecución, que se conoce como máquina virtual, en el cual, existe un proceso de compilación de el código fuente a un lenguaje intermedio, comunmente denominado *bytecode*. Este bytecode es luego ejecutado sobre un intérprete, al cual se denomina comunmente como 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 en un solo programa.

Los lenguajes de programación interpretados suelen ser de alto nivel y de *tipo dinámico*, es decir que la mayoría de las comprobaciones que se realizan en tiempo de compilación en otros lenguajes <sup>2</sup>, se realizan en tiempo de ejecución.

Si un lenguaje provee la posibilidad de ser ejecutado en varias plataformas <sup>3</sup> decimos que es multiplataforma. En el caso de los lenguaje compilados, es necesario disponer de un compilador para la plataforma, mientras que en un lenguaje interpretado, es necesario compilar el intérprete con el compilador de la plataforma.

Un lenguaje dinámico admite de manera directa en tiempo de ejecución, el agregado de código, extensión o redefinición de objetos o hasta inclusive modificar tipos de datos. Si bien estas tareas pueden ser realizadas en lenguajes no dinámicos, su implementación no es sencilla.

Jhon Cownan [Cowan2005] adempas hace una clasificación interesante sobre la utilización de los lenguajes de programación, diferenciando la *programación de sistema* de la *programación de aplicaciones*.

- **Programación de sistemas** En la programación de sistemas, el programador se centra en adaptar un nuevo dominio a interfaces genéricas preexistentes.
- **Programación de aplicaciones** En la programación de aplicaciones, por el contrario, modela el dominio de la aplicación de manera específica como paso inicial (análisis y diseño) y luego de la integración (si existe).

Podemos ejemplificar la programación de sistemas con la creación de un shellscript por parte de un administrador de sistemas basados en UNIX, donde existe un lenguaje de integración, shellscript <sup>4</sup> y las un set de utilidades de sistema con interfase conocida que le brindan funcionalidades como ordenamiento de cadenas, búsqueda y reemplazo de patrones, inicio y detención de procesos, interacción con el SO, etc.

**Java** es un lenguaje de programación orientado a objetos, multiplataforma y multipropósito, basado en máquina virtual, de compilación explícita. Java ha definido algunos elementos importantes en cuanto a la web dinámica, como los applets <sup>5</sup> y la especificación J2EE.

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

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

Su sintaxis es compleja y ambigua, una tarea puede ser realizada de muchas maneras diferentes, dado lugar a confusión.

**PHP** es un lenguaje interpretado, originalmente diseñado para ser embebido dentro del código HTML y procesado en el servidor (DSL) y que con los años ha intentado convertirse en un lenguaje de propósito general. Toma elementos de Perl y shellscript, C, y recientemente Java. Cuando el cliente solicita a través de una URL un módulo php, el interprete busca los los tags del lenguaje, y los reemplaza por la salida del bloque. Presenta una ventaja importante sobre la escritura de CGI, ya que no es necesario confeccionar un programa de usuario, la resolución de URLs está dada por la estructura del sistema de archivos. Si bien es muy popular <sup>6</sup> y está disponible en la gran mayoría de los servidores UNIX, simplificando el *deployment* es criticado por no poseer ámbito de nombres para los módulos,

---

<sup>2</sup> Estas comprobaciones comprenden el chequeo de tipos de datos, resolución de métodos entre otros.

<sup>3</sup> 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]

<sup>4</sup> Bash, Sh, Csh, Ksh, Zsh, etc.

<sup>5</sup> Pequeños programas que se ejecutan en el navegador web

<sup>6</sup> Utilización de PHP según php.net

promover el código desordenado, de difícil optimización y orientado a la web [BlogHardz2008] . Hoy es posible escribir aplicaciones gráficas mediante toolkits como Qt y GTK.

**Ruby** es un lenguaje orientado fuertemente objetos, multiplataforma, creado en 1995 por Yukihiro “Matz” Matsumoto, en Japón. A menudo comparado con *Smalltalk*, se suele decir que Ruby es un lenguaje de objetos puro, ya que *todo* es un objeto. Posee muchas características avanzadas como metaclasses, clausuras, iteradores, integración de expresiones regulares en el lenguaje, etc. Su sintaxis toma elementos de Perl, por lo cual, suelen existir convenciones de nombrado a las que cuesta acostumbrarse. Existen varios intérpretes de Ruby, siendo la oficial escrita en C, se concen YARV <sup>7</sup> , JRuby <sup>8</sup> , Rubinius <sup>9</sup> , IronRuby <sup>10</sup> , y MacRuby <sup>11</sup> .

Una de las Killer App <sup>12</sup> según el autor es el framework para contenido web denominado “Ruby on Rails” que alcanzó su versión 1.0 en el año 2005, que proponía un cambio radical al enfoque complejo de *J2EE*.

Su principales desventajas son la baja aceptación, quizás debido a que la documentación oficial solía estar en idioma Japonés (aunque la situación se ha venido revirtiendo ultimamente). Otra desventaja importante es que la velocidad del intérprete oficial es bastante baja cuando se la compara con otros lenguajes de programación y muestra variaciones importantes entre plataformas.

**Python** es un lenguaje interpretado, multiplataforma, orientado a objetos con dejos de programación funcional.

## 4.4 Python

El lenguaje de programación Python fue creado por Guido van Rossum en el año 1991. Se trata de un lenguaje dinámico, y se lo categoriza como multiparadigma, es decir, permite al programador utilizar diferentes formas de resolución de problemas:

- programación orientada a objetos
- programación estructurada
- programación funcional

También se soportan otros paradigmas mediante extensiones <sup>13</sup> .

Python usa tipo de dato dinámico y un recolector de basura basado en contadores de referencias. Gracias a su naturaleza dinámica, toda resolución de nombres (o símbolos) se realiza en tiempo de ejecución (*dynamic binding*). En la jerga del lenguaje se suele llamar a este hecho, *duck typing* y se lo asocia con el siguiente broma, *si tiene pico y hace cuac, se trata de un pato*.

Python puede ser extendido mediante módulos escritos en C o C++, y también se puede embeber el intérprete en otros lenguajes. Python permite actualmente cargar bibliotecas de enlace dinámico, de manera dinámica y realizar llamadas incluso con callbacks escritos en Python.

Python es considerado por parte de la comunidad UNIX, como una evolución de Perl, de sintaxis limpia y potente. Un caso sitable de esta “evolución” es el artículo escrito por Eric Raymond, “Why Python?”, donde explica su conversión de Perl a Python [EricRaymon2000].

Muchos de las sistemas webs basados en CGI, están escritos en Perl, por lo cual no es sorpresa encontrar una buena cantidad de proyectos del lenguaje Python orientados a la Web <sup>14</sup> .

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

<sup>7</sup> Yet Another Ruby VM, escrita por Sacasada Kiochi <http://www.atdot.net/yarv/>

<sup>8</sup> JRuby es una máquina virtual de Ruby escrita sobre la máquina virtual de **Java**, <http://jruby.codehaus.org/>

<sup>9</sup> Rubinius es una máquina virtual de Ruby escrita en **C++** <http://rubini.us/>

<sup>10</sup> IronRuby es una implementación de Ruby sobre la plataforma **.Net** <http://www.ironruby.net/>

<sup>11</sup> MacRuby es una implementación de Ruby sobre **Objective-C** para el sistema Mac OS X, <http://www.macruby.org/>

<sup>12</sup> Aplicación que populariza un lenguaje.

<sup>13</sup> PySWIP

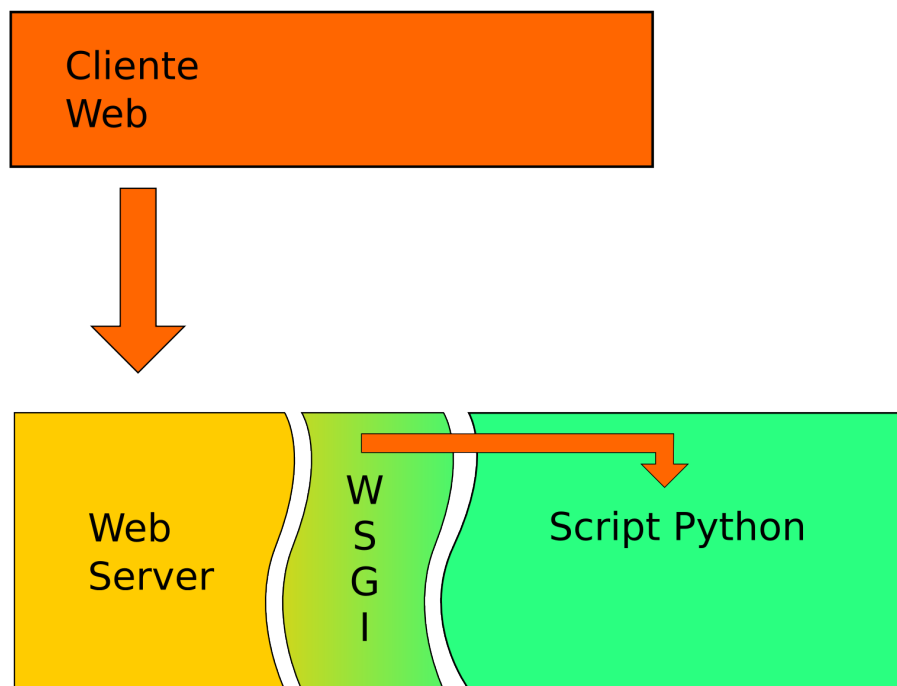
<sup>14</sup> En el repositorio de proyectos del lenguaje se encuentran más 1100 resultados para paquetes relacionados con el término “web”. <http://pypi.python.org/pypi/%3Aaction=search&term=web&submit=search>

### 4.4.1 WSGI

WSGI o Web Server Gateway Interfase es una especificación para que un web server y una aplicación se comuniquen. Es un estándar del lenguaje Python, descrito en el PEP <sup>15</sup> 333. Si bien WSGI es similar en su concepción a CGI, su objetivo es estandarizar la aparición de estructuras de software cada vez más complejas (frameworks *servidor-frameworks*)

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

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



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

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

---

<sup>15</sup> PEP *Python Enhancement Proposals* son documentos en los que se proponen mejoras para el lenguaje

---

# Frameworks Web

---

## 5.1 Frameworks

**Nota:** Poner CLI

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

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

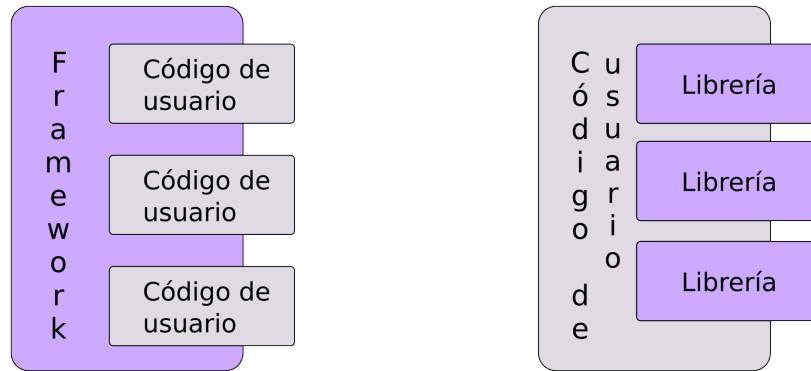
- **Inversion de control** Al contrario que las bibliotecas en las aplicaciones de usuario, en un framework, el flujo de control no es manejado por el llamador, sino por el framework. Es decir, cuando se utilizan bibliotecas o programas de usuario como soporte para brindar funcionalidad, estas son llamados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

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

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

Sin embargo, se suele argumentar que los frameworks pueden ser una carga, debido a la complejidad de sus APIs o la incertidumbre que genera la existencia de varios frameworks para un mismo tipo de aplicación. A pesar de tener como objetivo estandarizar y reducir el tiempo de desarrollo, el aprendizaje de un framework suele requerir tiempo

extra en el desarrollo, que debe ser tenido en cuenta por el quipo de desarrollo. Trás completar el desarrollo en un framework, el equipo de desarrrlo no debe volver a invertir tiempo en aprendizaje en sucesivos desarrollos.



**Nota:** Breve introducción a patrones arquitecturales y mvc

## 5.2 Model View Controler

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

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

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

Descripción del patrón:

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

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



controlador es el responsable de recibir los eventos de entrada desde la vista.

### 5.2.1 Framework Web

**Nota:** Ver diferencia entre sitio y aplicación [http://www.javahispano.org/contenidos/es/comparativa\\_de\\_frameworks\\_web/](http://www.javahispano.org/contenidos/es/comparativa_de_frameworks_web/)

Un framework web, es un framework de software que permite implementar aplicaciones web brindando soporte para tareas comunes como. Su objetivo es facilitar el desarrollo de aplicaciones web, .

En Wikipeda [WIKI002]

- Acceso a datos ORM
- Seguridad
- Mapeo de URLs
- Sistema de plantillas
- Caché
- AJAX
- Configuración mínima y simplificada

**Nota:** completar tema framework un poco, antes de caer en django

## 5.3 Mapeador Objeto-Relacional

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

Amazon.com, por ejemplo, es un buen ejemplo de un sitio que maneja una base de datos. Cada página de un producto es esencialmente una consulta a la base de datos de productos de Amazon formateada en HTML, y cuando se envían datos al servidor, como opiniones de cliente, estos son insertados en la base de datos de opiniones.

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

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

```
import MySQLdb
```

```
db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
cursor = db.cursor()
cursor.execute('SELECT name FROM books ORDER BY name')
names = [row[0] for row in cursor.fetchall()]
db.close()
```

Este enfoque funciona, pero presenta algunos problemas:

- Los parámetros de la conexión a la base de datos están codificando en duro (*hard-coding*).

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

Por otro lado y quizá más importante a la hora de desarrollar un programador que trabajé con programación orientada a objetos y bases de datos relacionales, debe realizar un cambio de contexto cada vez que requiera interactuar con la base de datos, escribiendo consultas en SQL y luego lidiar con los resultados obtenidos de las consultas entre los objetos. Este *cambio de contexto* es debido a una diferencia que existe entre los dos paradigmas involucrados. Mientras que el modelo relacional trata con relaciones, conjuntos y la lógica matemática correspondiente, el paradigma orientado a objetos trata con objetos, atributos

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

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

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

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

---

# Django

---

## 6.1 Introducción

**Nota:** Aquí tenemos que justificar por qué Django

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

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

En junio del 2008 fue anunciada la creación de la Django Software Foundation, la cual se hace cargo hasta la fecha del desarrollo y mantenimiento.

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

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

Django como framework de desarrollo consiste en un conjunto de utilidades de consola que permiten crear y manipular proyectos y aplicaciones. Este sigue el patrón MVC y como el controlador “C” es manejado por el mismo sistema los desarrolladores dieron a conocer a Django como un *Framework MTV*.

- *M* significa “Model” (Modelo), la capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.
- *T* significa “Template” (Plantilla), la capa de presentación. Esta capa contiene las decisiones relacionadas a la presentación: como algunas cosas son mostradas sobre una página web o otro tipo de documento.

---

<sup>1</sup> Del inglés “The Web framework for perfectionists with deadlines”

- V significa “View” (Vista), la capa de la lógica de negocios. Esta capa contiene la lógica que accede al modelo y la delega a la plantilla apropiada: puedes pensar en esto como un puente entre el modelos y las plantillas.

MVC o MTV la realidad es que ninguna de las interpretaciones es más “correcta” que otra. Lo importante es entender los conceptos subyacentes.

**Nota:** Justificación

- completitud
- popularidad
- simplicidad

## 6.2 Estructuración de un proyecto en Django

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

Un proyecto se crea mediante la siguiente orden:

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

Un proyecto es un paquete Python que contiene 3 módulos:

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

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

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

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

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

### 6.2.1 Módulo settings

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

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

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

### 6.2.2 Módulo manage

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

Los comandos de django permiten entre otras cosas:

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

### 6.2.3 Módulo urls

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

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

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

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

## 6.3 Mapeando URLs a Vistas

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

Es posible tambien en este momento definir algo de logica de negocios implementando vistas dentro del proyecto para dotar al sitio de algo de funcionalidad dinamica. Una función vista, es una simple función de Python que toma como argumento una petición Web y retorna una respuesta Web. En el momento de procesar una petición HTTP Django seleccionara y ejecutara la vista. Lo importante de este punto es como decirle a Django que vista ejecutar ante determinada url, es en este punto donde surgen las *URLconfs*.

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

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

## 6.4 El sistema de plantillas

Las vistas son las encargadas de retornar respuestas Web, entre estas respuestas esta el código HTML que debe ser enviado al cliente o navegador, Django separa el diseño de la página del código Python correspondiente a la logica de

negocio usando un *sistema de plantillas* para generar el HTML.

**Nota:** quizá completar un poco

## 6.5 Estructura de una aplicación Django

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

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

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

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

### 6.5.1 Módulo `models`

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

### 6.5.2 Módulo `views`

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

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

Dentro del módulo de urls

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

## 6.6 El ciclo de una petición

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

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

---

<sup>2</sup> Mediante el comando `syncdb` del módulo `manage` del proyecto

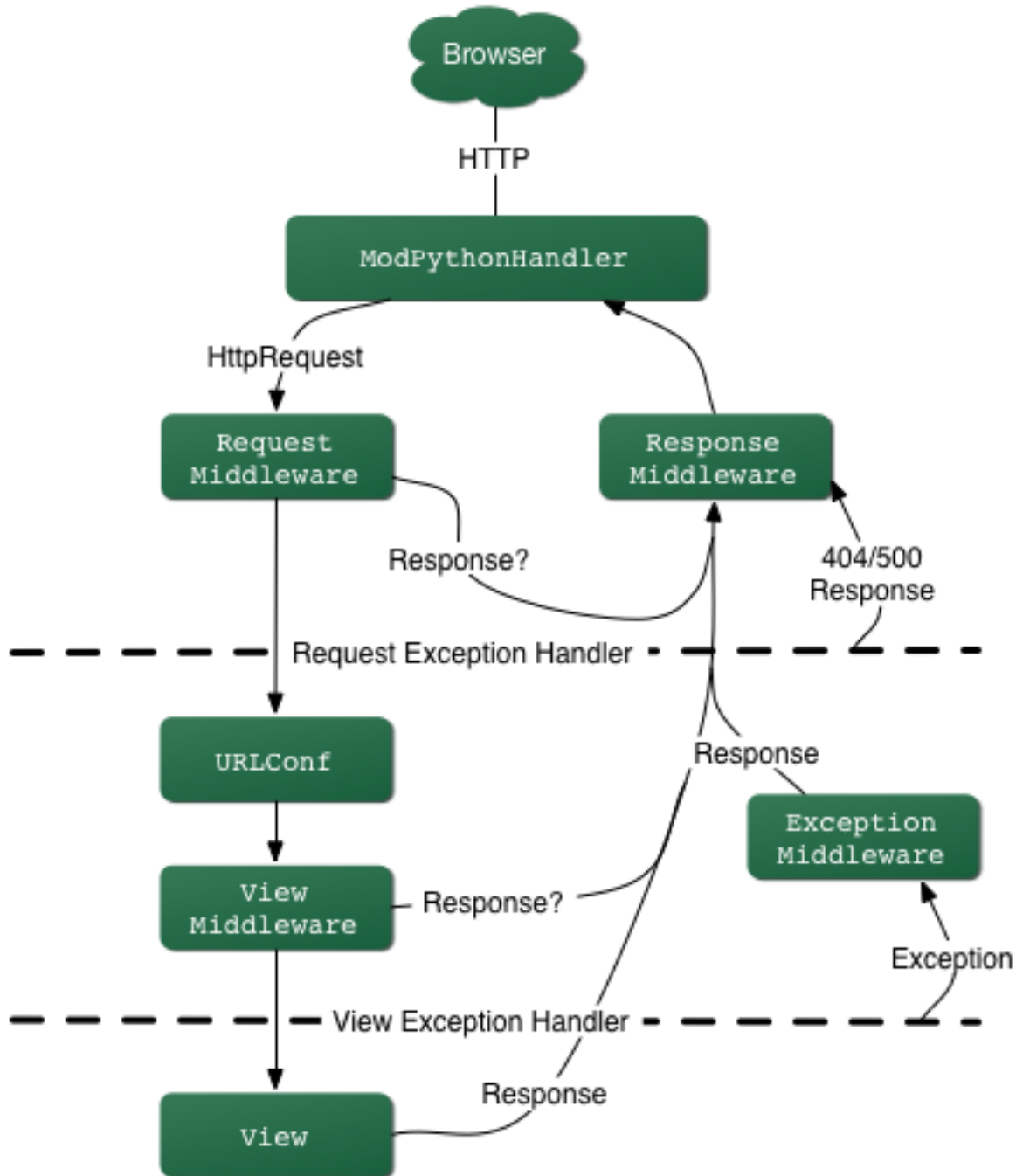
Luego de atravesar los middlewares de request, mediante las definiciones de URLs, se selecciona la vista a ser ejecutada.

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

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

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

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



## 6.7 Interactuar con una base de datos

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

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

Si bien existen pocas reglas estrictas sobre cómo desarrollar dentro de Django, existe un requisito respecto a la convención de la aplicación: “si se va a usar la capa de base de datos de Django (modelos), se debe crear una aplicación de Django. Los modelos deben vivir dentro de una aplicaciones”. Para crear una aplicación se debe proceder con el procedimiento ya mencionado en *manage*.

## 6.8 Modelos

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

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

Django define los modelos en Python por varias razones:

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

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

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

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

```
from django.db import models

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

nombre y apellido son atributos de clase



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

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

## 6.8.1 Usando la API

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

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

En estas líneas se ven algunos detalles de la interacción con los modelos:

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

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

Django provee también una forma de seleccionar, filtrar y obtener datos de la base a través de los administradores de consultas representado en el ejemplo anterior por `Persona.objects`.

## 6.8.2 Administradores de consultas

Los managers o administradores de consultas son los objetos que representan la interfase de comunicación con la base de datos. Cada modelo tiene por lo menos un administrador para acceder a los datos almacenados. Cada entidad presente en el modelo, tiene al menos un *Manager*. Este *Manager* encapsula en una semántica de objetos las operaciones de consulta (*query*) de la base de datos<sup>3</sup>. Un *Manager* consiste en una instancia de la clase `django.db.models.manager.Manager` donde se definen, entre otros métodos, `all()`, `filter()`, `exclude()` y `get()`.

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

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

<sup>3</sup> En el lenguaje SQL, las consultas se realizan mediante la sentencia `SELECT`.

<sup>4</sup> También conocida como *Lazy Evaluation*



## **Parte III**

# **Tecnologías del cliente**



---

# Estructura de un navegador

---

## 7.1 Navegador Web

Un navegador web o *web browser*, es un software encargado de representar documentos de hipertexto. El lenguaje en el cual se escriben estos documentos es HTML.

## 7.2 HTML

## 7.3 CSS

## 7.4 JavaScript

al archivo que contiene la funcionalidad. Mas tarde cuando el navegador descarga el documento y comienza su lectura al encontrar esta etiqueta solicita al servidor el archivo referenciado y lo interpreta, para continuar luego con la lectura del resto de las etiquetas.



---

# Evolución del JavaScript

---





---

# Google Gears

---

## 9.1 Introducción

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

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

Los tres componentes principales que incorpora gears son:

- **Local Server** Permite almacenar localmente datos correspondientes a las páginas webs. Tanto HTML, JavaScript e imágenes entre otros, pueden ser almacenados localmente por el cliente e interponerse entre el requerimiento del navegador al servidor en 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 cache del navegador la diferencia fundamental está en que la actualización de los recursos que almacena es mantenida y realizada por el desarrollador.

- **Database** Permite almacenar localmente datos que no correspondan a una página web pero son parte de la 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, disponer de un almacenamiento del tipo relacional en la máquina huésped.

- **Worker Pool** De manera similar a los “hilos” del sistema operativo, el manejador de hilos permite ejecutar acciones en segundo plano sin bloquear la ejecución del hilo principal del navegador.

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

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

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

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

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

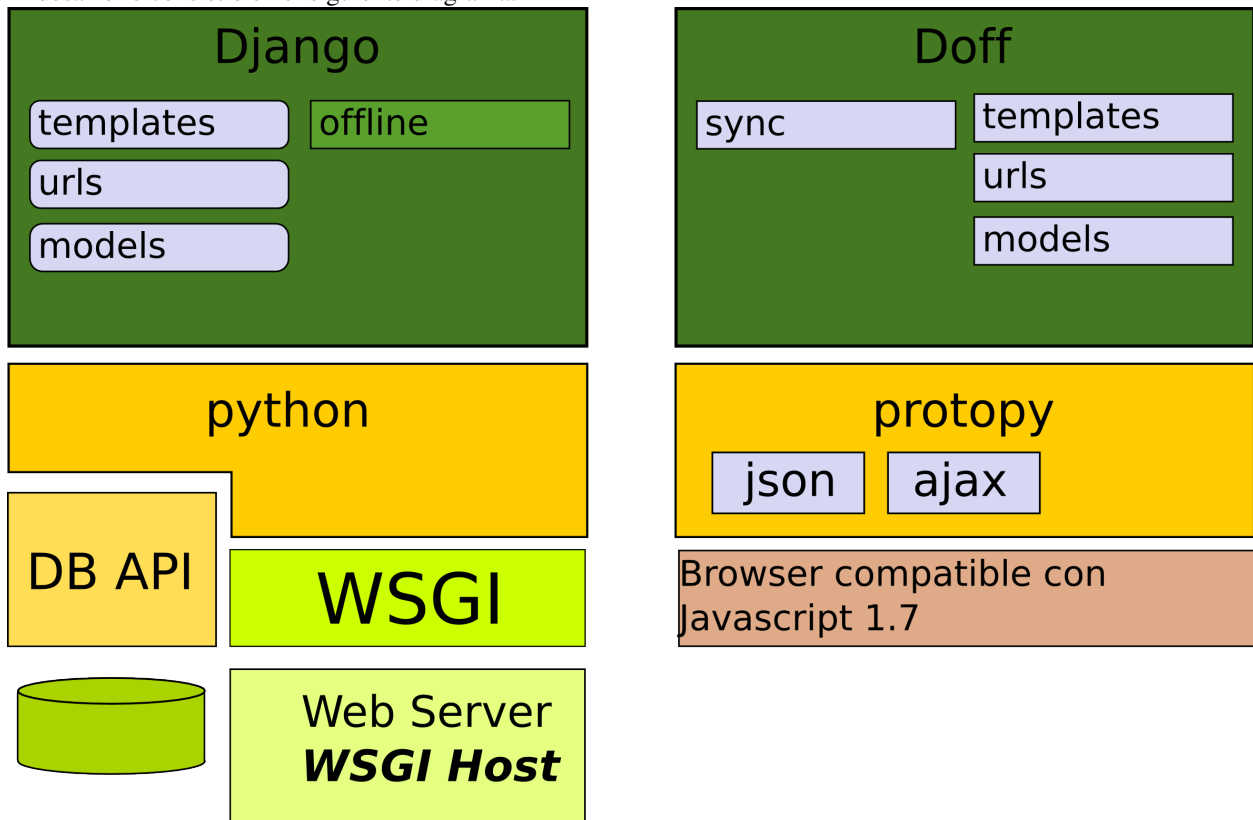
En todos los casos que se requiera tener instalada una aplicacion soportada por Gears vale la pena aclarar que el cliente debe haber accedido al menos una vez al servidor de la aplicacion web y otorgado los permisos de instalacion correspondiente.

# **Parte IV**

## **Desarrollo**



El desarrollo consistió en el siguiente diagrama:





---

# Una biblioteca en JavaScript

---

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

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

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

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

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

“La creación nace del caos”, la librería “Protopy” no escapa a esta afirmación e inicialmente nace de la integración de Prototype con las primeras funciones para lograr la modularización; con el correr de las líneas de código <sup>1</sup> el desarrollo del framework torna el enfoque inicial poco sustentable, requiriendo este de funciones más Python-compatibles se desecha la librería base y se continúa con un enfoque más “pythonico”, persiguiendo de esta forma acercar la semántica de JavaScript 1.7 a la del lenguaje de programación Python.

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

---

<sup>1</sup> Forma en que los informáticos miden el paso del tiempo.

## 10.1 Protopy

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

Para una referencia completa de la API de Protopy remítase al apéndice *Protopy*

## 10.2 Organizando el código

Como ya se vio en la sección dedicada a *JavaScript*, una de las formas tradicionales y recomendada de incluir funcionalidad en un documento HTML es mediante el tag *script*, haciendo una referencia en el atributo *src* a la url del archivo que contiene el código; en una instancia posterior, cuando el cliente accede al recurso, carga el archivo con las sentencias JavaScript y las interpreta en el contexto del documento.

El enfoque tradicional resulta sustentable para pequeños proyectos, donde el lenguaje brinda mayormente soporte a la interacción con el usuario (validación, accesibilidad, etc) y los fragmentos de código que se pasan al cliente son bien conocidos por el desarrollador, pero en proyectos que implican mayor cantidad de funcionalidad JavaScript, con grandes cantidades de código, este enfoque resulta complejo de mantener y evolucionar en el tiempo. Es por esto que para Protopy se buscó como primera medida una forma de organizar y obtener el código del servidor que resulte sustentable y escalable.

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

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

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

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

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

### 10.2.1 Obtener un módulo

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

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

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

---

<sup>2</sup> Ruta base desde la cual la biblioteca Protopy carga los módulos, por defecto esta es la url base del archivo `protopy.js` más el sufijo *packages*.



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

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

llamador

Para ver más *Apendice Protopy*.

## 10.2.2 Publicar un módulo

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

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

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

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

```
//Obtengo la funcion copy y deepcopy del modulo copy
require('copy', 'copy', 'deepcopy');

var SortedDict = type('SortedDict', [ Dict ], {
  __init__: function(object) {
    this.keyOrder = (object && instanceof(object, SortedDict)) ? copy(object.keyOrder) : [];
    super(Dict, this).__init__(object);
  },
  __iter__: function() {
    for each (var key in this.keyOrder) {
      var value = this.get(key);
      var pair = [key, value];
      pair.key = key;
      pair.value = value;
      yield pair;
    }
  },
  __deepcopy__: function() {
    var obj = new SortedDict();
    for (var hash in this._key) {
      obj._key[hash] = deepcopy(this._key[hash]);
      obj._value[hash] = deepcopy(this._value[hash]);
    }
    obj.keyOrder = deepcopy(this.keyOrder);
    return obj;
  },
  __str__: function() {
    var n = len(this.keyOrder);
    return "%s".times(n, ', ').subs(this.keyOrder);
  },
  set: function(key, value) {
    this.keyOrder.push(key);
    return super(Dict, this).set(key, value);
  }
});
```

```
    },
    unset: function(key) {
        without(this.keyOrder, key);
        return super(Dict, this).unset(key);
    }
});

//Publico el tipo de objeto SortedDict
publish({
    SortedDict: SortedDict
});
```

Asumiendo que el código presentado esta en un archivo llamado *sortedict.js* en la url base, esté representa un módulo en si mismo y el acceso se obtiene mediante la invocación de `require('sortedict')`. El ejemplo presenta muchas cosas nuevas, que seran abordadas a lo largo de este apartado, por ahora solo interesa la parte que corresponde al requerimiento de *copy* y la publicación de *SortedDict*.

La primera líneas del ejemplo incian requiriendo la funcion *copy* y *deepcopy* del módulo *copy*, ambas funciones se utilizan para realizar copias de objetos, superficial y en profundidad respectivamente. El ejemplo continua con una definición de un nuevo tipo de objeto llamado *SortedDict* que posteriormente publica en la ultima linea de código.

**Nota:** queda mas o menos claro?

## 10.3 Creando tipos de objeto

En el apartado teórico se menciono ya que en JavaScript todo es un objeto y que estos se crean y modifican dinamicamente en tiempo de ejecución.

Sin embargo, la creacion de objetos en Javascript carece de “clases” o si se quiere “tipos de objeto” y para crear dos o mas objetos de un mismo tipo se deben volver a declarar todos los miembros del objeto.

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

El objeto Function es utilizado como objeto instanciable en JavaScript, y el cuerpo de la función es el constructor del nuestros objetos. Una vez que tenemos un constructor, podemos llamarlo con el operador *new*.

En la programación basada en prototipos las “clases” no están presentes, y la re-utilización de procesos se obtiene a través de la clonación de objetos ya existentes y la extensión de funcionalidad.

Protopy agrega el concepto de clases al desarrollo, mediante un constructor de “tipos de objeto”. De esta forma los objetos pueden ser de dos tipos, las clases y las instancias. Las clases definen la disposición y la funcionalidad básicas de los objetos, y las instancias son objetos “utilizables” basados en los patrones de una clase particular.

**Nota:** hablar de las implementaciones de herencia y como es la nuestra

Creando ‘Clases’ en JavaScript

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

El objeto Function es utilizado como objeto instanciable en JavaScript, y el cuerpo de la función es el constructor del nuestros objetos. Una vez que tenemos un constructor, podemos llamarlo con el operador *new*.

*/clase de ejemplo/*

```
miClase = new Function();
nuevoObjeto = new miClase(); // instanciamos ‘miClase’
```

El método mas difundido para emular las clases en JavaScript es aprovechar el funcionamiento de la palabra clave `this` dentro de los constructores. Cuando una función es llamada con el operador `new`, `this` hace referencia al objeto que será retornado. Veamos como funciona.

*/clase de ejemplo/*

```
miClase = function(){
/*
agregamos miembros dinamicamente
al objeto que será retornado
*/
this.propiedad = "hola!";
this.metodo = function(){
/* aqui 'this' hace referencia al objeto al que pertenece el metodo */
alert(this.propiedad);
}
}

nuevoObjeto = new miClase(); // instanciamos 'miClase'
```

Ahora, `'nuevoObjeto'` va a referenciar al objeto que se retornó cuando se llamó a `'miClase'` usando el operador `new`, por lo tanto, va a tener todas las propiedades y metodos que se crearon dentro del constructor.

El problema de esta forma de crear clases es que cada vez que la función `'miClase'` es llamada, se crea una nueva función llamada `'metodo'`, de modo que cada objeto tiene su propia versión de `'metodo'` cuando, en realidad, todos los objetos deberían compartir la misma función.

La propiedad `'Prototype'`

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

Inicialmente esta propiedad es un objeto vacío, pero debemos modificarla para aprovechar esta característica del lenguaje. Reescribiendo el ejemplo anterior el código quedaría así:

```
/* creamos un constructor limpio */
miClase = new Function();
miClase.prototype.propiedad = "hola!";
miClase.prototype.metodo = function(){
/* aqui 'this' hace referencia al objeto al que pertenece el metodo */
alert(this.propiedad);
}

nuevoObjeto = new miClase(); // instanciamos 'miClase'
```

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

Adicionalmente, al utilizar la propiedad `prototype`, obtenemos otra ventaja, podemos usar la palabra reservada `instanceof`. Vemos como se usa si escribimos: `alert(nuevoObjeto instanceof miClase) // muestra 'true'`

El problema del uso de prototype, es que todas las propiedades hacen referencia a las mismas del prototipo, por lo que si modificamos estos objetos se van a modificar también en todos los objetos creados con esta clase. En siguiente ejemplo cambiamos nuestra propiedad a un array para poder modificarla sin cambiar nuestra referencia, de modo que podamos ver lo que explico con un ejemplo:

```
/* creamos un constructor limpio */
miClase = new Function();
miClase.prototype.propiedad = new Array("hola","hello");
miClase.prototype.metodo = function(){
/* aqui 'this' hace referencia al objeto al que pertenece el metodo */
alert(this.propiedad);
}
nuevoObjeto = new miClase(); // instanciamos 'miClase'
otroObjeto = new miClase(); // instanciamos 'miClase'
nuevoObjeto.propiedad.push("olá");
otroObjeto.metodo(); // muestra "hola","hello","olá"
```

Como pueden ver en el ejemplo anterior, 'nuevoObjeto' y 'otroObjeto' hacen referencia al mismo array, para solucionar este problema solo debemos asignar un objeto diferente a cada propiedad de nuestros objetos. El constructor es un buen lugar para realizar esta tarea:

```
miClase = function(){
this.propiedad = new Array("hola","hello"); // reemplazamos el valor de nuestra propiedad con un nuevo objeto
};
miClase.prototype.propiedad = new Array("hola","hello");
miClase.prototype.metodo = function(){
/* aqui 'this' hace referencia al objeto al que pertenece el metodo */
alert(this.propiedad);
}
nuevoObjeto = new miClase(); // instanciamos 'miClase'
otroObjeto = new miClase(); // instanciamos 'miClase'
nuevoObjeto.propiedad.push("olá");
otroObjeto.metodo(); // muestra "hola","hello" :)
```

Listo, tenemos clase y tenemos objetos que funcionan bien. Espero que les haya resultado claro, en el próximo post voy a explicar el mecanismo de la herencia en JavaScript (si!!, se puede!!). Hasta entonces.

Descargar archivo de ejemplo

Y ya para finalizar este tema en el que se trata el lenguaje JS a modo de introducción hablaremos de la herencia entre clases. En relación a este aspecto he de decir que tras buscar por internet la posibilidad de Herencia, me he encontrado con algunos ejemplos que principalmente no funcionan, como es el caso del ejemplo de herencias que nos muestra "Douglas Crockford".

En este caso y tras probar las posibilidades de herencia me di cuenta de que o no sabia exactamente como funcionaba o es que realmente no se soporta este tipo de herencia, yo he llegado a la conclusión de que no lo soporta.

Posteriormente vi que la librería prototype ha desarrollado un metodo por el cual se pueden realizar herencias entre clases. Pero aún así seguí buscando la forma de tener una metodología propia para poder implementar las herencias en nuestra propia librería de JS. Y así llegué a un post en el que se trataba el tema en cuestión y en el cuál “Harry Fuecks” nos ha abierto la puerta a dotar a nuestras propias librerías de herencia entre clases. ¿Y como es esto?, pues es tan sencillo como crearnos un objeto que lo que haga sea recorrer cada par “nombre:variable” del objeto que pasamos como original y asignárselo a nuevos pares “nombre:variable” del objeto al que deseamos copiar la herencia.

Bueno por si no me he explicado bien vamos a ver unos ejemplos para así ver su funcionamiento.

...

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

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

## 10.4 Extendiendo JavaScript

Si bien el *DOM* ofrece ya una *API* muy completa para acceder, añadir y cambiar dinámicamente el contenido del documento HTML, existen funciones muy utiles y comunes en los desarrollos que los programadores incorporan al HTML y se intentaron englobar y mantener dentro del nucleo de Protopy,

## 10.5 Extendiendo DOM

Si bien el *DOM* ofrece ya una *API* muy completa para acceder, añadir y cambiar dinámicamente el contenido del documento HTML, existen funciones muy utiles y comunes para la manipulación del DOM, que los programadores incorporan al proyecto en forma de bibliotecas propias o de terceros, este tipo de funciones se integraron al nucleo de Protopy con el objetivo de facilitar el desarrollo.

Funciones para modificar e incorporar elementos al documento son muy comunes y estan disponibles en Protopy, en conjunto con otras para el manejo de formularios, como serializacion, obtencion de valores.

En cuanto a los eventos

## 10.6 Envolviendo a gears

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

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

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

La función *create* del objeto *sys.gears* es la encargada de crear y retornar los objetos Gears, la creación de estos objetos puede ser directamente dependiente del objeto *Factory* original de Gears o estar asistida y extendida por un modulo de Protopy que se encuentre en el paquete gears en la url base.

Se busco extender algunos objetos Gears para el desarrollo del framework con la finalidad de interaccionar el backend de base datos con el acceso a la base propiamente dicho.

## 10.7 Auditando el codigo

## 10.8 Interactuando con el servidor

## 10.9 Soporte para json

JSON brinda un buen soporte al intercambio de datos, resultando de fácil lectura/escritura para las personas y de un rapido interpretacion/generacion para las maquinas. Se basa en un subconjunto del lenguaje de programación JavaScript, estándar ECMA-262 3ª Edición - Diciembre de 1999. Este formato de texto es completamente independiente del lenguaje de programación, pero utiliza convenciones que son familiares para los programadores de lenguajes de la familia “C”, incluyendo C, C++, C#, Java, JavaScript, Perl, Python y muchos otros.

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

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

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

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

Para mas informacion sobre JSON <http://www.json.org/>

Mientras que un cliente se encuentre sin conexión con el servidor web, es capaz de generar y almacenar datos usando su base de datos local. Al reestablecer la conexión con el servidor web, estos datos deben ser transmitidos a la base de datos central para su actualización y posterior sincronización del resto de los clientes. La transferencia de datos involucra varios temas, uno de ellos y que compete a este apartado, es el formato de los datos que se deben pasar por la conexión; este formato debe ser “comprendido” tanto por el cliente como por el servidor. Desde un primer momento se penso en JSON como el formato de datos a utilizar, es por esto que Protopy incluye un modulo para trabajar con el mismo.

No existe una razón concreta por la cual se deja de lado el soporte en Protopy para XML como formato de datos; aunque se puede mencionar la simplicidad de implementación de un parser JSON contra la implementación de uno en XML. Para el lector interesado agregar el soporte para XML en Protopy consta de escribir un modulo que realice esa tarea y agregarlo al paquete base.

El soporte para JSON se encuentra en el modulo “json” entre los modulos estandar de Protopy. Este brinda soporte al pasaje de estructuras de datos JavaScript a JSON y viceversa. Los tipos base del lenguaje JavaScript estan soportados y tienen su representación correspondiente, object, array, number, string, etc. pero este modulo interpreta además de una forma particular a aquellos objetos que implementen el metodo `__json__`, dejando de este modo en manos del desarrollador la representación en JSON de determinados objetos. La inclusión del metodo `__json__` resulta de especial importancia a la hora de pasar a JSON los objetos creados en base a tipos definidos por el desarrollador mediante el constructor “type”.

Con el soporte de datos ya establecidos en la librería, el framework solo debe limitarse a hacer uso de él y asegurar la correcta sincronización de datos entre el cliente y el servidor web, este tema se retomará en el capítulo de sincronización. ... TODO: retomar este tema para no ser un mentiroso :)

## 10.10 Ejecutando código remoto

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

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

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

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

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

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





## **Parte V**

# **Conclusiones y lineas futuras**



---

# Conclusiones

---

El desarrollo consistió en el siguiente diagrama:



---

# Lineas futuras

---

## 12.1 Sitio de administración

Django se caracteriza por brindar una aplicación `django.contrib.admin` de administración 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.

## 12.2 Historial de navegación

## 12.3 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 envidadas a segundo plano, de manera de no entorpecer el refresco de la GUI. Una característica 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.



## **Parte VI**

# **Glosario**





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

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

**JSON** [JavaScript-Object-Notation](#); formato ligero para el intercambio de datos.

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

**field** An attribute on a [model](#); a given field usually maps directly to a single database column.

**generic view** A higher-order [view](#) function that abstracts common idioms and patterns found in view development and abstracts them.

**model** Models store your application's data.

**MTV** hola

**MVC** [Model-view-controller](#); a software pattern.

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

**property** Also known as “managed attributes”, and a feature of Python since version 2.2. From [the property documentation](#):

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

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

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

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

the last bit (spring) is the slug.

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

**view** A function responsible for rendering a page.

**BSD** ve ese de

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

**HTTP** Hyper Text Transfer Protocol

**HTML** Lenguaje de hipertexto basado en etiquetas de marcado

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

**URL** Localizador universal de recursos, especificada en la [RFC 2396](#)

**deployment** Etapa en el desarrollo 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.

## **Parte VII**

# **Indices, glosario y tablas**



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



---

# Bibliografía

---

- [WikiCGI2009] *Interfaz de entrada común*, Wikipedia, 2009, último acceso Agosto 2009, [http://es.wikipedia.org/wiki/Common\\_Gateway\\_Interface#Intercambio\\_de\\_informaci.C3.B3n:\\_Variables\\_de\\_entorno](http://es.wikipedia.org/wiki/Common_Gateway_Interface#Intercambio_de_informaci.C3.B3n:_Variables_de_entorno)
- [DavidPollak2006] *Ruby Sig:How To Design A Domain Specific Language*, Google Tech Talk, 2:44, <http://video.google.com/videoplay?docid=-810328474422033344>
- [WikiPlataforma2009] *Multiplataforma*, Wikipedia, 2009, último acceso, Agosto de 2009, <http://es.wikipedia.org/wiki/Multiplataforma>
- [Cowan2005] *RESTful Web Services, An introduction to building Web Services without tears (i.e., without SOAP or WSDL)*, 2005, <http://home.ccil.org/~cowan/restws.pdf>
- [BlogHardz2008] <http://hardz.wordpress.com/2008/02/07/php-hipertexto-pre-procesado/>
- [EricRaymon2000] *Why Python*, Linux Journal, publicado el 1° de Mayo de 2000, <http://www.linuxjournal.com/article/3882>
- [WIK001] *Software Framework*, Wikipedia, 2009, [http://en.wikipedia.com/software\\_framework](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](http://en.wikipedia.org/wiki/Web_application_framework), última visita Agosto de 2009.





---

# Índice

---

## A

API, [61](#)

## B

BSD, [61](#)

## D

deployment, [61](#)

DOM, [61](#)

## F

field, [61](#)

## G

generic view, [61](#)

## H

HTML, [61](#)

HTTP, [61](#)

## I

i18n, [61](#)

## J

JSON, [61](#)

## M

model, [61](#)

MTV, [61](#)

MVC, [61](#)

## P

project, [61](#)

property, [61](#)

## Q

queryset, [61](#)

## R

RPC, [61](#)

## S

slug, [61](#)

## T

template, [61](#)

## U

URL, [61](#)

## V

view, [61](#)

## X

XHTML, [61](#)