Django Framework

Fecha 24/12/2013

Realizado por:

Guarini Tolón, Fernando - Enzo

Jarana Pérez, José Andrés

Merchán Cachinero, Borja Manuel

Ruiz Moreno, Lorenzo

Realizado para:

La asignatura Evolución y Gestión de la Configuración

Página intencionadamente dejada en blanco

Control del documento

Registro de cambios en el documento

VERSIÓN	MOTIVO	REALIZADO POR	FECHA
1.0	Creación	Fernando	11/11/2013
1.0.1	Modificación de la plantilla del documento	Borja	18/12/2013
1.1	Edición de cada apartado	Fernando Borja José Lorenzo	19/12/2013
2.0	Finalización del documento	Fernando Borja José Lorenzo	23/12/2013

Página intencionadamente dejada en blanco

Resumen

Django es un framework escrito en Python para el desarrollo rápido de aplicaciones web. Con Django se pueden tener listo un sitio web funcional (corriendo en un servidor) en pocos minutos para empezar a desarrollar el resto a partir de ahí.

En las siguientes páginas describiremos exhaustivamente teniendo como modelo la documentación oficial de Django los procesos que se siguen en el desarrollo de esta herramienta para gestionar su código fuente, las incidencias (bugs) que surjan, así como su puesta a disposición al público y su esquema de versionado.

Explicaremos cómo se inicia un proyecto Django de cero y los pasos que hay que dar para poner en marcha un pequeño servidor local. Y también descubriremos la gran comunidad que trabaja en este magnífico proyecto para facilitar el día a día de los programadores web.

Sería conveniente poner algo de vuestro proyecto en más detalles, dando algunas pinceladas de los problemas abordados y las soluciones propuetas.

Índice

En principio se podría poner en un capítulo priopio

1. Introducción

En este documento se explica de una manera clara y concisa el desarrollo del proyecto un análisis acerca de la gestión de Django, realizado por:

- Guarini Tolón, Fernando Enzo
- Jarana Pérez, José Andrés
- Merchán Cachinero, Borja Manuel
- Ruiz Moreno, Lorenzo

Esto está en la portada. Eliminar.

En el proyecto de este documento se realizará un análisis sobre cómo gestionan en Django el código fuente, las incidencias, la comunicación, etc...

Django es un framework de desarrollo web de codigo abierto, escrito en Python, que respeta el paradigma conocido como Model Template View. Está liberado al público bajo una licencia BSD. Como curiosidad, el nombre del framework hace alusión al guitarrista de jazz gitano, Django Reinhardt.

El diseño de Django, proporciona una serie de características que facilitan el desarrollo rápido de páginas orientadas a contenidos.

Toda la documentación de Django la podemos encontrar en la web del proyecto, la cual se encuentra alojada en la siguiente url: https://www.djangoproject.com/

Hemos elegido Django, por lo bien documentado que se encuentran todos los aspectos estudiados en este documento, además de que creemos, que siguen muy buenas prácticas a la hora de gestionar el proyecto, lo que puede llevarnos a aprender bastantes cosas acerca de la gestión de la evolución.

Nuevamente falta información de lo que habéis hecho vosotros en el proyecto. Un resumen y una introducción deben ser autocontenidos. Alguien lee vuestro resumen + intro y aún no sé qué habéis hecho, qué problemas abordados y qué soluciones habéis propuesto todo ello en resumen.

Demasiado rápido a la herramienta. Mejor primero los conceptos antes que las herramientas concretas

Pondré así las cosas que no vea claras y que necesitan de explicación en este caso a q se refieren con este rol

2. Gestión del código fuente

Para la gestión del código fuente se usa git como software de gestión de versiones. El repositorio está hospedado en GitHub y el equipo de Django recomienda que todos los contribuidores trabajen con esta herramienta para agilizar el trabajo de los commiters del proyecto.

¿esto no es para gestionar los cambios en el código?

Este proyecto se basa en tickets para gestionar el código fuente. Los contribuidores crearan tickets para informar de bugs o de nuevas funcionalidades. En el apartado 6 de este documento se detallará la creación de dichos tickets.

Para empezar a escribir código para Django debemos solicitar primeramente que se asigne un ticket. Una vez que se ha asignado un ticket a un contribuidor, éste se compromete a trabajar en él y a solucionarlo en un tiempo razonable.

Empezáis por un caso concreto y en este apartado se espera algo más general aunque luego se vaya a lo concreto

2.1. Gestión de ramas

Parece pronto hablar de ramas directamente . Sería bueno definir qué consideráis una rama en este caso.

El código de Django está compuesto de las siguientes ramas activas:

- Rama master: En esta rama se centra el desarrollo hacia la última versión con pequeñas mejoras, arreglos de bugs,… siendo revisada a diario.
- Rama stable/1.5.x: En esta rama se encuentra la versión 1.5 aún soportada por Django. Mayormente se centra en arreglar bugs críticos
- Rama stable/1.4.x: Rama para la versión LTS 1.4
- Para el desarrollo de grandes características de la nueva versión se usarán forks de la rama principal del repositorio. Estas ramas se unirán a master una vez finalizad su desarrollo.

El primer paso para trabajar en el código es crear una nueva rama de nombre ticket_xxxx basada en la rama upstream/master. Este proceso se realizará para todos los tickets, incluso para las tareas más simples. Por ejemplo:

Vais demasiado rápido a la herramienta con creta. Debéis dar mejor una visión general para luego concretar

```
git checkout -b ticket_6668 upstream/master
```

Con este comando estaríamos trabajando en el ticket 6668, que trata de optimizar una clase de utilidades (https://code.djangoproject.com/ticket/6668)

Note que como estamos trabajando como rama base master, la funcionalidad a añadir será para la versión en desarrollo. Si se quisiera trabajar en la versión 1.4 o 1.5 la nomenclatura sería del tipo ticket_XXXX_1_4 y estaría basada en la rama upstream/stable1.4.X

Una vez que hemos trabajado en el código procederemos a realizar el commit.

Comentar cada cuánto hay que hacer el commit, quién lo hace, cómo se controla, qué roles se definen o se deberían definir,...

2.2. Roles

(perdón por el comentario anterior sobre los roles pues veo que lo comentáis aquí)

Hay dos tipos de personas que puede realizar un commit:

- Core commiters: Personas que tienen una larga lista de contribuciones al proyecto y han demostrado una larga dedicación al desarrollo de Django. Este grupo tiene acceso completo a realizar un commit.
- Partial commiters: Personas que son expertos del dominio del problema. Ellos tienen acceso al subsistema que se ha decidido que esté bajo su jurisdicción. Por otra parte, los 'partial commiters' tienen voz y votos en votaciones relacionada con su subsistema. Este tipo de acceso es propio de contribuidores que aportan código a una parte específica del proyecto y desean mantenerlo.

Está bien reportado pero sería bueno hacer un análisis de esta propuest

Por otra parte, cualquier persona que quiera ser miembro de los commiters del proyecto es bienvenida. Para ello debe pedir acceso a un commiter existente y tras una votación realizada por la comunidad se tramitará su solicitud.

En general, hasta este punto, hacéis la doc del proyecto pensando que todo el mundo entiende el "argot".

2.3. Pautas para aplicar un patch

Ya que el proyecto Django está alojado en la plataforma GitHub, la forma de gestionar los parches es a través de los pull requests de GitHub.

El contribuidor que desee aportar código al proyecto deberá realizar un commit en forma de pull request. El commit debe seguir las pautas de estilo que se describirán en apartados sucesivos.

El primer paso es crear una nueva branch con base upstream/master. La convención de nombres del proyecto dice que esta rama se debe llamar pull_xxxx donde xxxx es el código del patch a aplicar. A continación se procede a bajar el parche del repositorio y a aplicarlo con el comando git am.

Todo esto hay que explicarlo mucho más

```
git checkout -b pull_xxxxx upstream/master

curl https://github.com/django/django/pull/xxxxx.patch | git am
```

En este punto, el contribuidor puede trabajar en el código. Una vez concluido procederemos como sigue:

```
# Traemos la rama master y comprobamos que está lista para recibir cambios

git checkout master

git pull upstream master

# Hacemos el merge como "fast-forward" para evitar un merge commit.

git merge --ff-only pull_xxxxx

# Revisa que sólo los cambios esperados son llevados al upstream.
```

Todo esto hay que explicarlo más y mejor

```
git push --dry-run upstream master

# Hacemos el push

git push upstream master

# Por último nos deshacemos de la rama en la que hemos estado trabajando con el patch.

git branch -d pull_xxxxx
```

Cabe destacar que GitHub provee una herramienta para realizar pull-request de manera rápida sólo con un click, el equipo de desarrollo recomienda usar esta opción sólo cuando el contribuidor está completamente seguro de que el pull está 100% listo y libre de errores ya que este método se salta los test unitarios programados por el equipo.

Explicar más esto.

es muy interesante pero le falta elaboració

Por último, el historial de cambios de un pull-request debe ser lo más usable posible, para ello se debe seguir los siguientes procedimientos:

- Si un parche contiene 'back-and-forth commits' (estos son commits en los que se vuelve hacia atrás y luego se sigue trabajando en ellos) se deben reescribir en uno solo.
- Separar cambios de diferentes commits organizados por lógica de grupo: si en un commit se realizar una revisión de estilo y al mismo tiempo se realiza cambios en el fichero se debe separar el commit en dos diferentes para facilitar la lectura del historial.
- Los test deben ser pasados y la documentación debe ser construida después de cada commit. Ni los test ni la documentación debe emitir 'warnings'
- Pequeños o parches triviales deben ser realizados en un commit. Trabajos medianos o grandes deben ser divididos en múltiples commits si es posible.

Está esto traducido del inglés?

Esto es gestión de la construcción

La calidad del código es muy importante para el equipo de Django, por eso se pide ser responsable y estar seguro de que el código añadido no tiene errores. Aun así, si se detecta un error y se tiene que revertir un commit se debe seguir el siguiente procedimiento:

- Si es posible dejar que el contribuidor del código con errores revierta su propio código.
- Sólo se debe revertir código del cual se tenga permiso expreso del contribuidor original.
- Usar git revert, así se mantiene el commit original en el historial del commit.

Esto debe ser analizado y discutido

- Si no se puede contactar con el autor y el problema es grave (crashing bug, grandes fallos en los tests, …) se debe contactar con la lista de correo.

- Si no se llega al acuerdo entre el autor original y el contribuidor que crea que el código tiene problemas se debe contactar con la lista de correo y someterse a votación de la comunidad.
- Si el commit introduce una vulnerabilidad de seguridad entonces debe ser revertido inmediatamente sin necesidad de contar con el permiso del autor original.

2.4. Guía de estilo a la hora de realizar commit

El equipo de Django recomienda seguir las siguientes pautas al trabajar con su repositorio git:

- Nunca se debe modificar el historial de commit de las principales ramas en activo (django/django).
- Si se desea aplicar cambios grandes al código se debe consultar primero en la lista de correo con la comunidad.
- Se debe escribir los mensajes de commit en inglés y en 'past tense'. El mensaje del commit debe ser como máximo de 72 caracteres y debe haber una línea en blanco entre el asunto del commit y el mensaje del commit.
- Si el commit se realiza a una rama se debe indicar como prefijo al asunto del commit la rama en la que se realiza. Por ejemplo [1.4.x] Asunto
- Se deben separar los commit de arreglo de fallos de los de nuevas características.

Podriáis poner un ejemplo de mensaje de commit bueno y otro de commit malo

Esto se debería elaborar mucho, mucho más.

2.5. Políticas de estilo

Django se basa en la Guía de Estilo para Código Python PEP8, una serie de convenciones que todo desarrollador que escriba código en la distribución oficial de Python debe seguir.

Se puede consultar dicha guía de estilos en la página oficial de la documentación de Python: http://www.python.org/dev/peps/pep-0008/

En resumen, las recomendaciones más importantes son:

- Usar cuatro espacios de identación. Se recomienda no usar el tabulador por motivos de compatibilidad.
- Usar el carácter subrayado (_) para nombres de variables, de funciones y de métodos. No usar el estilo camelCase.

referencia?

- Comenzar las clases con letra mayúscula.
- Evitar usar espacios y líneas en blanco innecesarias así como sentencias import que se queden sin uso cuando se refactoriza el código.

Hay que tener en cuenta de que el equipo de Django considera que algunas de las recomendaciones de PEP8 no son necesarias. Principalmente, no se recomienda que el tamaño de línea sea como máximo de 79 caracteres.

Respecto a las políticas de estilo propias del framework:

- Cuando trabajamos con plantillas recomiendan poner un espacio entre la tag de la plantillas y las llaves. Por ejemplo: {{ foo }} sería lo preferido respecto a {{foo}}.
- En las vistas, el primer parámetro en una función debe ser llamado request, se debe abstenerse de usar abreviaciones como reqs o similares.
- En los modelos:
    - Los atributos deben ser palabras en minúsculas y usar el carácter subrayado. No se debe usar camelCase.
    - La clase Meta debe aparecer después de que los atributos hayan sido definidos y separada de éstos por una línea en blanco.
    - Si se define el método __str__ se debe usar python_2_unicode_compatible().
    - El orden de los atributos y clases internas debe ser el siguiente: atributos de la base de datos, atributos de gestión propios, clase Meta, método __str__(), método save(), método get_absotute_url() y resto de métodos propios.
    - Si se desean definir choices, se deben definir como tuplas de tuplas, todas en mayúsculas y con algún atributo del modelo. Por ejemplo:

```
class MyModel(models.Model):

    DIRECTION_UP = 'U'

    DIRECTION_DOWN = 'D'

    DIRECTION_CHOICES = (

        (DIRECTION_UP, 'Up'),

        (DIRECTION_DOWN, 'Down'),

    )
```

Por último, dos aspectos importantes a tener en cuenta son:

- No se debe poner el nombre del autor en el código con el que se desee contribuir al proyecto.
- Es necesario marcar todas las cadenas para que puedan ser internacionalizadas con el código de estilo que se detalla en el siguiente apartado de este documento.

Ejercicio propuesto

Aplique el parche del pull-request 2105. Puede consultarlo en https://github.com/django/django/pull/2105

Solución

El primer paso sería solicitar el parche. Suponemos que hemos realizado dicho paso y continuamos con el proceso.

Explicar estos pasos. Ponerlo en formato código.

git checkout -b pull_2105 upstream/master

curl https://github.com/django/django/pull/2105.patch | git am

En este punto podemos trabajar con el parche. Para aplicarlo:

git checkout master

git pull upstream master

git merge --ff-only pull_2105

git push --dry-run upstream master

git push upstream master

git branch -d pull_2105

3. Gestión de la construcción y de los entregables

Dado que Django es un framework escrito en Python, un lenguaje puramente interpretado, este software no necesita procesos de construcción al estilo de otros proyectos escritos por ejemplo en C, donde se realizan una serie de compilaciones periódicas para construir un software ejecutable.

En Django en cambio nos encontramos cómo en su propia página web de descargas se nos explica que, si queremos, somos libres de tener siempre instalada la última versión del código en nuestro ordenador haciendo pulls diarios de su repositorio oficial en GitHub. Se nos avisa, eso sí, de los riesgos que conlleva esto (posibles fallos de retrocompatibilidad o pequeños bugs que se cuelan en el desarrollo normal de cualquier software).

Sin embargo, si lo que preferimos es usar una versión estable hasta que la siguiente sea lanzada, el equipo de Django se encarga de empaquetar sin más en un fichero .tar.gz ciertos estados del código en Git que ellos consideran sólidos para que alguien pueda desarrollar una aplicación web completa sin encontrarse con errores del framework.

El esquema actual de versionado de Django es 1.X.Y donde X representa un gran cambio respecto de su anterior versión e Y representa una revisión del código, que suele centrarse en la corrección de errores.

Entiendo el tema de la construcción en algo que es puramente interpretado pero el tema de los entregables podría haberse elaborado mucho más.

Ejercicio propuesto:

Descargue los últimos cambios del código Django y empaquételos en un fichero Django-1.7.0.tar.gz

Solución:

Abrimos un terminal y escribimos las tres siguientes líneas:

```
git clone https://github.com/django/django.git

rm -r django/.git* django/.hgignore

tar czfv Django-1.7.0.tar.gz django/
```

4. Gestión del despliegue

Dado que Django trabaja sobre Python, un lenguaje interpretado, no existe un mecanismo puramente dedicado al despliegue. Necesariamente será necesaria la ejecución de tareas de forma manual.

Esto no necesariamente debería ser así.

Otro punto a tener en cuenta es que desplegar Django es un proceso trivial ya que es un framework dedicado al desarrollo de sitios webs. Lo interesante del proyecto Django es el despliegue de las aplicaciones desarrolladas con el framework, por lo que se ha considerado interesante describir dicho proceso en este apartado.

El primer paso para desplegar Django, fundamental, es tener instalado el intérprete de Python 2.7, 3.2 o 3.3. Este trabajo no trata sobre la instalación de Python así que para más información sobre su instalación en las diversas plataformas existentes, se recomienda visitar su página oficial www.python.org.

Una vez instalado Python, la instalación de Django es trivial. Usando el gestor de paquetes pip solo habría que ejecutar:

Poner una referencia aquí

```
pip install Django
```

Cabe destacar que pip es una herramienta multiplataforma para instalar y manejar diversos paquetes de Python de forma fácil. Su página web es http://www.pip-installer.org/en/latest/

Alternativamente se puede instalar Django desde la página oficial de liberaciones del proyecto en https://www.djangoproject.com/download/. Una vez bajada la última versión y descomprimida basta con ejecutar:

```
python setup.py install
```

Una vez en este punto podemos continuar con el proceso de despliegue.

La principal plataforma de despliegue de Django es WSGI, un estándar de Python para servidores webs y aplicaciones. Según se indica en la página oficial de WSGI http://wsgi.readthedocs.org/en/latest/what.html:

"WSGI es el acrónimo de Web Server Gateway Interface. Es una especificación que describe como el servidor web se comunica con las aplicaciones web, y como las aplicaciones web se encadenan juntas para procesar una solicitud.

WSGI es un estándar de Python descrito en detalle en el PEP 3333 http://www.python.org/dev/peps/pep-3333"

Django posee el comando startproject para crear una configuración simple WSGI automáticamente que cumpla dicho estándar. Su uso es simple:

```
django-admin.py startproject proyecto /Ruta/Absoluta
```

El archivo de configuración creado es settings.py dentro del paquete proyecto. Este archivo de configuración puede ser modificado a gusto del desarrollador.

El concepto principal del despliegue con WSGI es la 'aplicación' la cuál será llamada por el servidor de aplicaciones para comunicarse con el código creado. Físicamente, la 'aplicación' es un módulo Python accesible por el servidor. Con el comando startporject automáticamente se crea el archivo <proyecto>/wsgi.py que contiene dicha 'aplicación'.

Cabe destacar que dicha configuración es usada en entornos de desarrollo tanto como de producción.

Respecto al citado servidor de aplicaciones, en entornos de producción el servidor más usado y testeado, por tanto el recomendado, es Apache con mod_wsgi, un módulo de Apache que puede ejecutar cualquier aplicación Python que cumpla con la especificación WSGI.

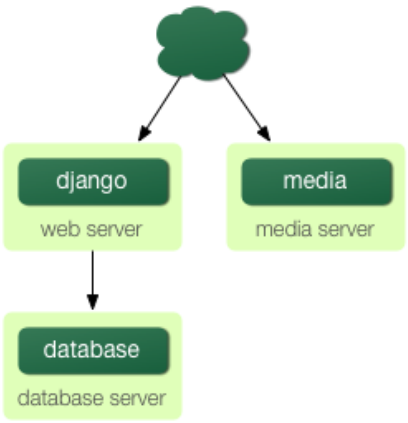
Una vez instalado y activado mod_wsgi, la configuración más básica que debemos realizar al archivo httpd.conf del servidor Apache es la siguiente:

```
WSGIScriptAlias/ /ruta/absoluta/proyecto/wsgi.py
WSGIPythonPath /ruta/absoluta
<Directory /path/absoluta/proyecto>
<Files wsgi.py>
Order deny,allow
Require all granted
</Files>
</Directory>
```

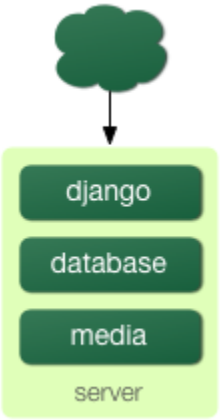
Con estas líneas nos aseguramos de dos cosas: que el sitio sea accesible por Python y que el servidor pueda acceder al archivo wsgi.py, dos cosas clave para el buen funcionamiento del proceso de despliegue.

Para detalles sobre la configuración del servidor de aplicaciones se debe consultar con la documentación oficial de mod_wsgi en http://code.google.com/p/modwsgi/ . Concretamente es especialmente interesante la parte que trata de la integración con Django https://code.google.com/p/modwsgi/wiki/IntegrationWithDjango.

Respecto al servidor de ficheros, cabe notar que Django no sirve archivos por sí mismo, sino que deja el trabajo al servidor web a elección nuestra. El equipo del proyecto Django recomienda usar un servidor web separado para servir media y archivos estáticos (por ejemplo archivos css, imágenes, favicon.ico, robots.txt,…).



Aun así nada impide que el servidor de archivos y de Django corran en el mismo servidor, en este caso se recomienda separar ambos servidores en distintos servidores virtuales en caso de Apache.



Fuente de las imágenes: http://www.djangobook.com/en/2.0/chapter12.html

La elección de una u otra opción es cuestión de recursos y de rendimiento. Obviamente la primera opción es la más cara en recursos pero el rendimiento es notablemente mejor.

4.1. Herramientas alternativas

Aparte del proceso descrito anteriormente, existen herramientas alternativas en el proceso de despliegue con Django. Algunas de las más interesantes son:

- **Gunirorn ('Green Unicorn'):** Es un servidor WSGI puramente escrito en Python para máquinas UNIX. No tiene dependencias y es fácil de instalar y usar. Para integrarlo con Django simplemente hay que añadirlo a las INSTALLED_APPS del archivo de configuración http://docs.gunicorn.org/en/latest/run.html#django-manage-py. Para más información sobre el proceso de despliegue consultar http://gunicorn.org/index.html#deployment
- **uWSGI:** Servidor WSGI escrito en C. Tiene una amplia documentación alojada en su repositorio de GitHub https://github.com/unbit/uwsgi-docs

Ejercicio propuesto

En su distribución Linux preferida realice el despliegue de una aplicación Django en un servidor apache con configuración por defecto.

Solución

Usando Fedora:

En la mayoría de distribuciones Python ya viene instalado por defecto así que procederemos a instalar Django con el gestor de paquetes pip:

yum -y install python-pip

pip install Django

Procederemos a instalar el módulo mod_wsgi. Note que es posible que, como prerrequisito, deba:

- Instalar el servidor apache si no está instalado ya: # yum -y install httpd
- Instalar la versión de desarrollador del servidor: # yum -y install httpd-devel.x86_64
- Instalar las cabeceras de desarrollo de Python: # yum -y install python-devel.x86.64

Una vez cumplidos todos los prerrequisitos nos dirigimos a la página del proyecto mod_wgsi y nos bajamos la última versión estable:

https://code.google.com/p/modwsgi/downloads/list

A continuación nos dirigimos al directorio en el que se ha descargado el comprimido y ejecutamos:

tar xvfz mod_wsgi-X.Y.tar.gz

./configure

make

make install

make clean

Activamos el módulo en el servidor apache:

En /etc/httpd/conf editamos el archivo httpd.conf y añadimos la línea

LoadModule wsgi_module modules/mod_wsgi.so

Reiniciamos el servidor apache: # systemctl restart httpd.

Una vez en este paso creamos la aplicación por defecto con:

django-admin.py startproject miproyecto /var/www/html

Finalmente añadimos al archivo httpd.conf lo siguiente:

```
WSGIScriptAlias / /var/www/html/myproyecto/wsgi.py
WSGIPythonPath /var/www/html
<Directory /var/www/html/myproyecto>
<Files wsgi.py>
Order deny,allow
Require all granted
</Files>
</Directory>
```

Y volvemos a reiniciar el servidor apache.

Note que estos pasos pueden ser tomados con cualquier aplicación ya creada que queramos desplegar.

5. Gestión de incidencias y depuración

Cabe mencionar que los problemas de seguridad solo están disponibles para un conjunto de miembros cerrados de la comunidad Django, es decir solamente los desarrolladores de alta confianza, los que llevan más tiempo, son los responsables de algún modo de los problemas que puedan ocurrir en cuanto a la seguridad de este framework, dicho esto en la misma página de la comunidad se puede observar cuáles son sus políticas en cuanto a temas de seguridad (https://docs.djangoproject.com/en/dev/internals/security).

En la misma página se advierte que antes de reportar un bug (fallo) y/o solicitar una nueva funcionalidad se deben tener en cuenta los siguientes aspectos:

- Comprobar que alguien no haya presentado el error o la funcionalidad solicitada mediante la búsqueda o haya ejecutado consultas personalizadas en el ticket tracker.
- No usar el sistema de etiquetado para pedir ayuda sobre preguntas. Para eso ya existe la lista de usuarios de Django (https://docs.djangoproject.com/en/dev/internals/mailing-lists/#django-users-mailing-list) o el canal en IRC (irc://irc.freenode.net/django) para este tipo de consultas.
- No reabrir cuestiones que se hayan marcado como *"wontfix"* por un desarrollador del núcleo. Esta manera de marcar temas significa que no se ha podido encontrar solución a esa cuestión en particular. En caso de no estar seguro el usuario en cuestión podrá preguntar el porqué de ese etiquetado a su pregunta en la comunidad de Django-developers.
- No usar el ticket tracker para largas discusiones, ya que es probable que los usuarios que sigan ese ticket se puedan llegar a perder.

5.1. Reporte de bugs

Los bugs que se detallan y redactan de manera correcta son de gran ayuda. Sin embargo hay una cierta cantidad de sobrecarga que implica trabajar con cualquier sistema de seguimiento de errores para que la ayuda del usuario sirva para mantener operativo y en correcto funcionamiento el sistema de ticket tracker. En particular:

- Se recomienda la lectura del FAQ (https://docs.djangoproject.com/en/dev/faq/) para comprobar si el problema podría ser uno muy común entre los usuarios de Django.
- Preguntar, antes de nada, en el canal de IRC o la lista de django-users si no se está seguro de si lo que se está viendo es un error o no.
- Escribir o redactar informes completos, reproducibles y específicos, de errores. Para ello se debe incluir una descripción clara y concisa del problema, y un conjunto de instrucciones para replicarlo. Añadir, a su vez, toda la información de depuración como se pueda, tales como fragmentos de código, casos de prueba, trazas de excepción, capturas de pantalla, etc. Un pequeño caso de prueba realmente buena es la

Se podría poner un ejemplo de un buen mensaje y de uno malo. Algo de la plantilla

mejor manera de informar de un error, ya que gracias a esta técnica se podría confirmar más rápidamente el error sucedido.

- No se debe publicar en Django-developers sólo para anunciar que se ha presentado un informe de un bug. Todos los tickets son enviados a otra lista de correo, django-updates (https://docs.djangoproject.com/en/dev/internals/mailing-lists/#django-updates-mailing-list) la cual es gestionada por los desarrolladores y miembros interesados de la comunidad.

Para comprender mejor el ciclo de vida del ticket una vez que se haya creado se recomienda visitar el centro de entrenamiento de tickets disponible en la web; más adelante se explicaran los mecanismos involucrados en el manejo de tickets.

5.1.1. Manejo de tickets

referencia?

Django usa un "*Track*" para gestionar el trabajo sobre la base de código. Trac es un "cuidado jardín" comunitario de los errores encontrados por persones.

Desde la propia web se aclara que se basan en la comunidad para seguir participando, hay que gestionar los tickets de manera precisa y cuando se tenga algún tipo de duda acudir a las listas de correo anteriormente mencionadas.

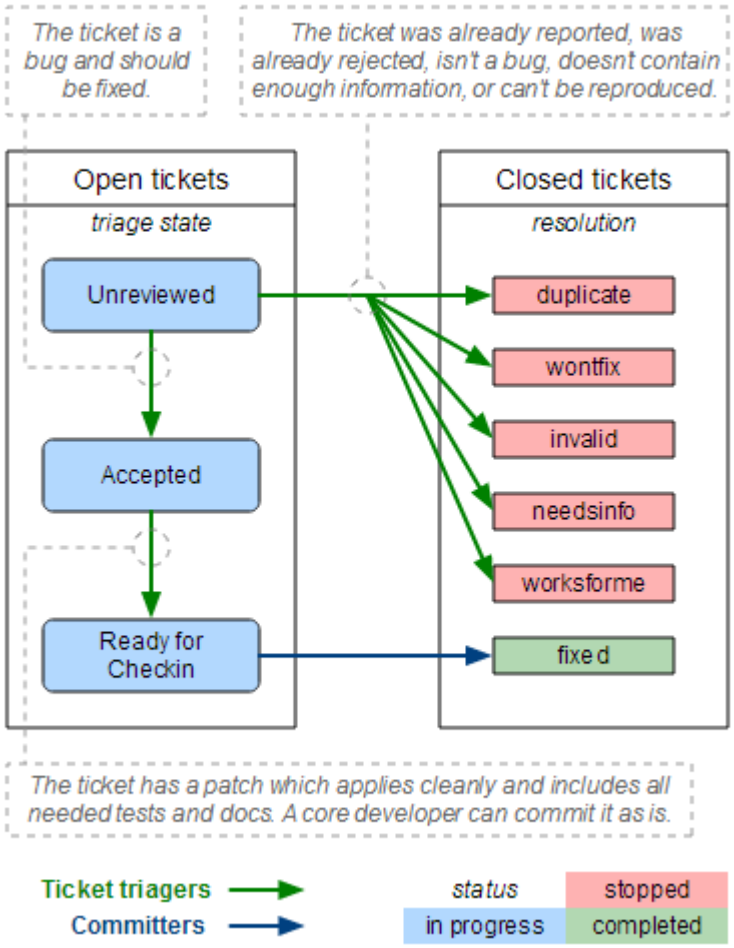
5.1.1.1. Flujo de trabajo *"Triage"*

Por desgracia, no todos los informes de errores y solicitudes de funcionalidades en el sistema de ticket tracer proporcionan los detalles necesarios. Un número de tickets tienen parches, pero esos parches no cumplen con todos los requisitos de un buen parche.

Una forma de ayudar es clasificar las entradas que han sido creadas por otros usuarios. El equipo central y varios miembros de la comunidad trabajan en esta regularidad, pero desde la web se agradece ayuda extra.

La mayor parte del flujo de trabajo se basa en el concepto de etapas de *"triage"* de un ticket. Cada etapa describe en que parte de su ciclo de vida está un ticket en cualquier momento. Esto además de un conjunto de flags, indican fácilmente de quien es cada ticket y que se espera de ese ticket.

A continuación se muetra una imagen alusica sobre el ciclo de vida de un ticket. Dicha imagen ha sido obtenida de https://docs.djangoproject.com/en/dev/internals/contributing/triaging-tickets/



Se disponen dos roles en el diagrama:

- *"Commiters"* (también llamados desarrolladores principales): son personas con acceso de confirmación que se encargan de hacer las grandes decisiones, escribiendo grandes porciones del código y la integración de los aportes de la comunidad.
- *"Tiket Triagers":* alguien en la comunidad Django que decide involucrarse en el proceso de desarrollo de Django.

A continuación se expone un modelo de ejemplo tomado de la página anteriormente mencionada:

- Alicia crea un ticket, y carga un parche incompleto (no hay pruebas, implementación incorrecta)
- Bob revisa el parche, lo marca como "aceptado", "necesita pruebas" y "mejora de las necesidades de parche", y deja un comentario diciendo a Alicia cómo se podría mejorar el parche.

- Alicia actualiza el parche, añadiendo pruebas (pero no el cambio de la implementación). Quita, a su vez, las dos banderas que tenía el comentario de Bob.
- Carlos revisa el parche y restablece el indicador "mejora necesidades de conexión" con otro comentario sobre la mejora de implementación.
- Alicia actualiza el parche, arreglando la implementación. Quita, esta vez, el flag de "mejora necesidades de conexión".
- Desiré revisa el parche, y marca la "*RFC"*.
- Jacobo, un desarrollador del núcleo, revisa el parche RFC, lo aplica a su salida y lo confirma.

Algunos tickets requieren muchos menos feedbacks que este, pero otros, en cambio, necesitan más información adicional para documentarlos.

5.1.1.2. Estados de Triage

A continuación se procederá a describir con más detalle las distintas etapas por las que un ticket puede pasar a lo largo de su vida útil.

No revisados.

El ticket aún no ha sido revisado por alguien, o no se ha hecho un juicio sobre dicho ticket, ya que no se sabe aún si el ticket contenía una cuestión valida, o debe ser cerrado por cualquier razón.

Aceptados.

El significado absoluto de "aceptado" es que el problema descrito en el ticket es válido y está en alguna etapa de ser procesado. A continuación se detallan ciertas circunstancias:

- ***Aceptado sin flags:***

  El ticket es válido pero nadie ha presentado un parche para el todavía. A menudo, esto significa que se puede comenzar de forma segura a escribir un parche para dicho ticket. Esto es generalmente más cierto para el caso de errores aceptados sobre alguna funcionalidad. Un ticket para un error que ha sido aceptado significa que el tema ha sido verificado por al menos un *"triager"* como un error legítimo, y probablemente se debería arreglar lo antes posible. Una nueva funcionalidad aceptada solo puede significar que un *"triager"* pensó que debería ser bueno incluirla.

- ***Aceptado con flags:***

  El ticket está esperando a que la comunidad revise el parche suministrado. Esto significa que hay que descargar el parche y probarlo, verificando que contiene pruebas y documentación, además hay que ejecutar el conjunto de pruebas incluidas en el parche y dejar un feedback sobre el ticket.

- ***Aceptado con parche pero necesita…***

  Esto significa que el ticket se ha revisado y se ha encontrado que necesita más trabajo. Se necesitan pruebas o algún tipo de documentación. Siempre se debe acompañar el ticket con un comentario que explique lo que necesita dicho parche para mejorar el código.

Listos para hacer Checkin.

El ticket fue revisado por cualquier miembro de la comunidad que no fuera la persona que lo suministro y se encontró que cumplía con todos los requisitos para aplicar el commit a dicho parche.

Un commiter del núcleo tiene que proporcionar ahora una revisión final antes de que se le haga el commit al parche.

Algún día / tal vez.

Esta etapa no se muestra en el diagrama. Solo es utilizado por los desarrolladores principales para realizar un seguimiento de las ideas de alto nivel o de las peticiones de nuevas funcionalidades a largo plazo.

Estos tickets son poco frecuentes y en general menos útiles ya que no describen cuestiones viables concretas. Son las solicitudes de mejora las que podemos considerar.

5.1.1.3. Otros atributos de Triage

Un número de flags, que aparecen como casillas de verificación en el Trac, pueden enviarse con el ticket, estas son:

Tiene parche.

Esto significa que el ticket tiene un parche asociado. Estos serán revisados para ver si el parche es “bueno”.

Los tres campos siguientes (las necesidades de documentación, las pruebas, o el de que el parche necesite mejoras) se aplican solo si se ha suministrado un parche con el ticket.

Necesita documentación.

Este indicador se utiliza para tickets con parches que necesitan algún tipo de documentación extra. La documentación completa de funcionalidades es requisito previo antes de que se pueda revisar en el código base.

Necesita pruebas (tests).

Este flag indica que el parche necesita algún tipo de prueba unitaria. Una vez más, esta es una parte necesaria para que un parche sea válido.

El parche necesita mejoras.

Este indicador significa que aunque el ticket tiene un parche, no está del todo listo para realizar el checkin del mismo. Esto podría significar que el parche no se aplica limpiamente, hay una falla en la aplicación, o que el código no cumple con los estándares.

"Easy picking".

Tickets que requieren un pequeño, y fácil parche.

Tipo.

Los tickets deben ser calificados según:

- ***Nueva funcinalidad*** para añadir algo nuevo.
- ***Bug:*** cuando algo existente está mal o no se comporta como se esperaba.
- ***"Cleanup"/ optimización:*** para cuando no hay nada que falle, pero algo se podría hacer mejor, más rápido y más consistente.

Componente.

Los tickets deben ser clasificados en componentes que indiquen en que área del código base de Django pertenece. Esto hace que los tickets estén mejor organizados y sean más fáciles de encontrar.

Severidad.

El atributo de la severidad (gravedad) se utiliza para identificar los "*blockers"*, esto es, cuestiones que se deben arreglar antes de la lanzar la próxima versión de Django. Normalmente esos temas son bugs que causan regresiones a versiones anteriores o que podrían causar la perdida de datos severos. Este atributo es muy raro que utilice y la gran mayoría de los tickets tienen un nivel de severidad "normal".

Versión.

Es posible usar el atributo versión para indicar en que versión se ha identificado el bug reportado.

"UI / UX".

Este indicador se utiliza para las entradas que se refieren a las experiencias de interfaz de usuario y de preguntas sobre alguna experiencia del usuario. Este indicador sería apropiado, por ejemplo, para indicar las características de cara al usuario en los formularios o la interfaz de administración.

"Cc".

Es posible añadir el nombre de usuario o la dirección de correo electrónico a este campo para ser notificado cuando se realizan nuevas aportaciones al ticket.

"Keywords".

Con este campo puede marcarse un ticket con múltiples palabras clave. Esto puede ser útil, por ejemplo, para agrupar varias entradas de un mismo tema. Las palabras clave pueden ser coma o un espacio separado.

5.1.1.4. Cerrado de Tickets

Cuando un ticket ha completado su ciclo de vida útil, es hora de que se cierre. El cierre de un ticket es de una gran responsabilidad. Hay que estar seguro de que el problema este realmente resuelto.

Si se cierra un ticket, siempre se ha de asegurar de lo siguiente:

- Asegurarse, de nuevo, de que el problema se ha resuelto.
- Dejar un comentario explicando la decisión de cerrar el ticket.
- Si hay una manera de que se pueda mejorar el ticket para reabrirlo, hay que hacerlo saber.
- Si el ticket es un duplicado, hay que hacer referencia al ticket original.
- Hay que ser cortés. A nadie le gusta que le cierren su ticket por que sí.

Un ticket puede ser resuelto de las siguientes maneras:

- ***Arreglado:*** utilizado por los desarrolladores principales de Django, los cuales indican que se ha resuelto el problema.
- ***Invalido:*** se usa si el ticket se encuentra incorrecto. Esto significa que el problema en el ticket es en realidad el resultado de un error de usuario, o describe un problema con algo que no sea Django, o no es un informe de error o solicitud de funcionalidad como tal.
- ***"Wontfix":*** se utiliza cuando un desarrollador del núcleo decide que esta solicitud no es apropiada para su consideración en Django. Este suele ser elegido después de una discusión en la lista de correo de desarrolladores de Django.
- ***Duplicado:*** usado cuando otro ticket cubre el mismo problema. Cerrando tickets duplicados, se mantiene la discusión en un solo lugar, lo que ayuda a todos.
- ***"Worksforme":*** se usa cuando el ticket no contiene detalles suficientes para replicar el error inicial.
- ***Necesita información:*** usado cuando el ticket no contiene información suficiente para reproducir el problema reportado, pero es potencialmente valido. Este ticket debe reabrirse cuando se suministre más información.

Si se considera que un ticket ha sido cerrado por error, hay que reabrir el ticket informado del motivo de la nueva apertura.

5.1.1.5. Colaborar con triagers

El proceso de "*triage"* es impulsado principalmente por los miembros de la comunidad. En realidad, cualquiera puede ayudar.

Los desarrolladores del núcleo pueden proporcionar información sobre cuestiones con las que estén familiarizados, o tomar decisiones sobre temas polémicos, pero no son responsables del triaging de tickets en general.

Para participar, hay que comenzar por crearse una cuenta en el Trac.

Entonces, ya se puede ayudar a con:

- Cierre de tickets "sin revisar", "inválidos", "worksforme" o "duplicados".
- Cierre de tickets "sin revisar" como "necesitan información", cuando la descripción es escasa para ser aceptado.
- Corrección de las "necesitan pruebas", "necesitan documentación" o "tiene parche" para tickets que tiene dichas etiquetas marcadas incorrectamente.

- Ajuste de la bandera de “easy picking” para tickets que son pequeños y relativamente fáciles.
- Ajuste del tipo de entradas que siguen siendo “*Uncategorized”*.
- Comprobar que los tickets viejos siguen siendo válidos.
- Identificar tendencias y temas en los tickets.
- Verificar si los parches enviados por otros usuarios son correctos.

Sin embargo, se pide el seguimiento de todos los miembros de la comunidad en general para que trabajen en la base de datos de tickets:

- No cerrar las entradas como “*wontfix”*. Los desarrolladores principales harán la determinación final de la suerte de un ticket, por lo general, después de consultar con la comunidad.
- No promover tickets propios para realizarle el “checkin”. Se puede marcar otras entradas de la gente que se han revisado como listas para hacer el checkin.
- No revocar una decisión que ha sido hecha por un desarrollador del núcleo. Si se está en desacuerdo con una decisión que se ha hecho, se debería enviar un mensaje a django-developers.
- Si no se está seguro de hacer un cambio, no hacerlo.

5.2. Reportar bugs de la interfaz de usuario y funcionalidades

Si el bug o funcionalidad recae sobre un aspecto visual, se propone unas pautas a seguir en dicho caso:

- Incluir capturas de pantalla en el ticket, que son el equivalente visual de un caso de prueba. Se debe reportar exclusivamente el problema nada más.
- Si el problema es difícil de mostrar mediante el uso de una imagen fija, se debe considerar el uso de otras técnicas como el de una breve screencast, para ello usar un software especializado en este tipo de tareas.
- Si se está ofreciendo un parche que cambia el aspecto o comportamiento de la interfaz de usuario de Django, se debe adjuntar el antes y el después de la escena o screencasts. Los tickets que carezcan de este tipo de adjuntos serán difícilmente gestionados por los desarrolladores para poder así asesorar la correcta solución a dicho problema.
- El uso de capturas de pantalla no eximen al usuario de otras buenas prácticas de presentación de informes. Hay que asegurar se de incluir URLs, fragmentos de código e instrucciones paso a paso sobre como reproducir el comportamiento que se aprecia en dichas capturas adjuntadas.

- Hay que asegurarse también de establecer el indicador *"UI/UX"* en el ticket, ya que de esta manera cualquier usuario interesado en su bug podrá encontrarlo fácilmente.

5.3. Solicitar funcionalidades

Desde la misma web se hace hincapié en que siempre se está tratando de hacer Django mejor, y las peticiones de los usuarios sobre funcionalidades son una parte clave de estas mejoras. A continuación se exponen algunos consejos sobre cómo hacer una petición con más eficacia:

- Se debe asegurar de que la funcionalidad requiere cambios en el núcleo de Django. Si la idea puede desarrollarse como una aplicación independiente o un módulo aparte, por ejemplo, se desea apoyar a otro motor de base de datos es probable que se le recomiende a ese usuario que lo desarrolle por el mismo de manera independiente. Por otra parte, si el proyecto reúne el apoyo comunitario suficiente, se podría considerar dicha funcionalidad para que se incluya en Django.
- Primero hay que solicitar la funcionalidad en la lista de django-developers y no en el sistema de ticket tracker. Con esto se conseguirá que su funcionalidad se lea por más miembros de la comunidad.
- Describir de manera clara y concisa las características de la funcionalidad que falta y que aporta el usuario y también de cómo se desea implementar. Se debería incluir algún código de ejemplo para tener como base.
- Explicar el motivo de porque se desea la funcionalidad. En algunos casos es obvio, pero como Django está diseñado para ayudar a los desarrolladores reales que consiguen trabajos reales, se tendrá que explicar, si no es tan obvio, porqué la funcionalidad sería útil.

Si los desarrolladores principales, los del núcleo, están de acuerdo en la funcionalidad, entonces es apropiada la creación de un ticket. Hay que incluir un enlace al debate sobre django-developers en la descripción del ticket.

Al igual que con la mayoría de los proyectos de código abierto, charlas de código… si se está dispuesto a escribir el código para la función de uno mismo, o mejor aún, si ya se ha escroto, es mucho más probable que sea aceptada. Para ello solo hay que hacer un fork sobre Django en GitHub, crear una futura branch y mostrar la manera en la que se trabaja sobre dicha funcionalidad.

5.3.1. Documentar nuevas funcionalidades

La política que se sigue en Django para la creación de nuevas funcionalidades es:

- Toda la documentación de nuevas funcionalidades debe estar escrita de una manera que designen claramente las funcionalidades que solo están disponibles en la versión de desarrollo de Django.

Hay que asumir que los lectores de documentación usan la última versión y no la de desarrollo.

- La manera preferida para el marcado de las nuevas funcionalidades es anteponiendo la documentación con: *""..versionadded:: X.Y"*, seguida de una línea en blanco y un contenido (identado) opcional.
- Mejoras generales, u otros cambios para las APIs que conviene destacar deberían utilizar la directiva *"..versionchanged:: X.Y"*(con el mismo formato mencionado anteriormente).

A continuación se muestra un ejemplo de la propia web de Django:

- Primero, el documento*"ref/setting.txt"* podría tener el siguiente diseño general:

```
- ========
- Settings
- ========
- ...
- .. _available-settings:
- Available settings
- ==================
- ...
- .. _deprecated-settings:
- Deprecated settings
- ====================
- ...
```

(Captura obtenida de https://docs.djangoproject.com/en/dev/internals/contributing/writing-documentation/#documenting-new-features)

- El documento *"topics/settings.txt"* podría contener algo como lo siguiente:

```
You can access a :ref:`listing of all available settings

<available-settings>`. For a list of deprecated settings see

:ref:`deprecated-settings`.

You can find both in the :doc:`settings reference document

</ref/settings>`.
```

Se usa el elemento de referencia cruzada *"Sphinx"* cuando se quiere enlazar con otro documento en su conjunto y el elemento "*ref*" cuando se desea vincular a una ubicación arbitraria en un documento.

- A continuación, obsérvese cómo se anotan los siguientes valores:

```
- .. setting:: ADMINS

- ADMINS

- ------

- Default: ``()`` (Empty tuple)

- A tuple that lists people who get code error notifications. When

- ``DEBUG=False`` and a view raises an exception, Django will email these people

- with the full exception information. Each member of the tuple should be a tuple

- of (Full name, email address). Example::

-     (('John', 'john@example.com'), ('Mary', 'mary@example.com'))

- Note that Django will email *all* of these people whenever an error happens.

- See :doc:`/howto/error-reporting` for more information.
```

(Captura obtenida de https://docs.djangoproject.com/en/dev/internals/contributing/writing-documentation/#documenting-new-features)

Esto marca el siguiente encabbezado como el objetivo "*canónico*" de los administradores de ajustes. Esto significa que para referirse a los administradores se debe usar el comodando "*: seting: ADMINS*".

Así es cómo básicamente todo encaja.

5.3.2. Mejorar la documentación

Algunas pequeñas mejoras se pueden realizar para hacer que la documentación se lea y vea mejor:

- La mayor parte de los documentos *"index.txt"* tiene muy poco o nada de texto de introducción. Cada uno de estos documentos necesita una buena introducción al contenido que se vaya a tratar.
- El glosario es muy superficial.
- Añadir mas objetivos de matadatos. Hay muchos lugares que se ven como:

- ``File.close()``
- ~~~~~~~~~~~~~~~~

... estos deben ser:

- .. method:: File.close()

(Captura obtenida de https://docs.djangoproject.com/en/dev/internals/contributing/writing-documentation/#documenting-new-features)

Esto es, usar meta datos en lugar de títulos.

- Añadir más enlaces, casi todo lo que es un código en line en este momento, probablemente se pueda convertir en una referencia externa.
- Agregar campos de información siempre y cuando sea necesario en el lugar oportuno.
- Siempre que sea posible, usar enlaces. En lugar de "*''ADMINS''*", usar "*setting:'ADMINS'*".
- Usar directivas donde sea apropiado.
- Añadir *".. code-block:: <lang>"* a bloques litereales para que se destaquen.
- Cuando se refiriera e a clases/funciones/módulos, etc., se debería usar el nombre totalmente calificado (fully-qualified) del objetivo: *"(:class:`django.contrib.contenttypes.models.ContentType`)."*

5.4. Cómo se toman las decisiones

Siempre que sea posible, se intenta alcanzar un consenso preliminar. Para ello, a menudo se tienen votos informales en la lista django-developers sobre una funcionalidad. En estas votaciones se sigue el estilo de votación inventado por Apache y se utiliza sobre sí misma en Python, donde los votos se dan como 1, 0, -0, o, -1. Aproximadamente estos votos se refieren a:

- 1: "Me encanta la idea y estoy firmemente comprometido con ella."
- 0: "Suena bien para mí."
- -0: "No estoy muy emocionado, no estaré en contra."
- -1: "Estoy en total desacuerdo y no quiero que la idea pase a mayores."

Aunque estos votos en django-developers son informales, van a ser tomados muy enserio. Después de un periodo de votación, si surge un consenso evidente se seguirán los votos.

Sin embargo, el consenso no siempre es posible. Si no se alcanza un consenso, o si la discusión se esfuma sin una decisión concreta, se usará un proceso más formal.

Cualquier commiter del núcleo puede pedir una votación formal, utilizando el mismo mecanismo de votación anterior. Una propuesta se considerará realizada por el equipo central si:

- Hay tres votos de "1" entre los miembros del equipo central.
- N hay un voto de "-1" entre cualquier miembro del equipo central.
- No se ha ejecutado un voto negativo.

Al llamar a una votación, el creador de la misma debe especificar un plazo para recibir los votos. Una semana es lo típico que se suele sugerir como unidad mínima de tiempo.

Dado que este proceso permite a cualquier commiter del núcleo de vetar una propuesta, cualquier "-1" de voto debe ir acompañado de una explicación que lo aclare y que necesitaría ese voto para convertirse en al menos un "0".

Siempre que sea posible, estos votos formales deben anunciarse y ser públicos en las listas de correo de django-developers. Sin embargo, las cuestiones excesivamente sensibles o polémicas, podrán celebrarse en privado.

MUY interesante y elaborada esta sección. Hay cosas que se podrían explicar y poner algo mejor pero en general está muy bien elaborada

6. Gestión de la variabilidad

El interés por el estudio de la variabilidad en el desarrollo del software ha aumentado significativamente durante los últimos años. Esto se debe a su interés en diversos campos, desde la personalización del software a las líneas de productos. La forma más común de gestionar la variabilidad en líneas de productos software es mediante modelos de características o "*features"* que permiten además seleccionar la configuración de cada aplicación concreta dentro de una línea de productos.

Durante los últimos años los sistemas software cada vez tienden a soportar una mayor variabilidad, entendida como la habilidad de cambio o de personalización de un sistema. Esta variabilidad se debe a las demandas de los usuarios de sistemas más adaptables a sus necesidades (sistemas personalizables) y, sobre todo, a la presión del mercado, que hace más rentable fabricar líneas de productos software para reutilizar la mayor parte del esfuerzo de desarrollo.

En el caso de Django, podemos afirmar que el concepto de variabilidad es el eje central del proyecto, cuyo único fin es, precisamente, utilizar las herramientas que se nos proporcionan para desarrollar nuestra propia aplicación web.

Por tanto, Django no posee una gestión de la variabilidad como tal, sino que forma parte del mismo fin del proyecto. corresponde a cada usuario del framework, a cada persona que vaya a desarrollar con él, adaptarlo a sus necesidades de negocio.

Ejercicio: Se podría trabajar algo más sobre esto, por ejemplo, estudiando cómo los "frameworks" son elementos que estudian la variabilidad.

Cree una aplicación Django desde cero y llámela "Recetario", arranque el servidor por defecto de Django y visualice el proyecto recién creado en un navegado.

Solución:

Para crear nuestro primer proyecto, abrimos un terminal, nos ubicamos en la carpeta donde queramos crear nuestro proyecto y escribimos:

django-admin.py startproject recetario

Esta instrucción creará dos directorios con el nombre del proyecto (en este caso: recetario) y 5 archivos distribuidos de la siguiente manera:

- manage.py
- recetario
  - __init__.py
  - settings.py
  - urls.py
  - wsgi.py

Para arrancar el servidor, escribimos en el terminal lo siguiente:

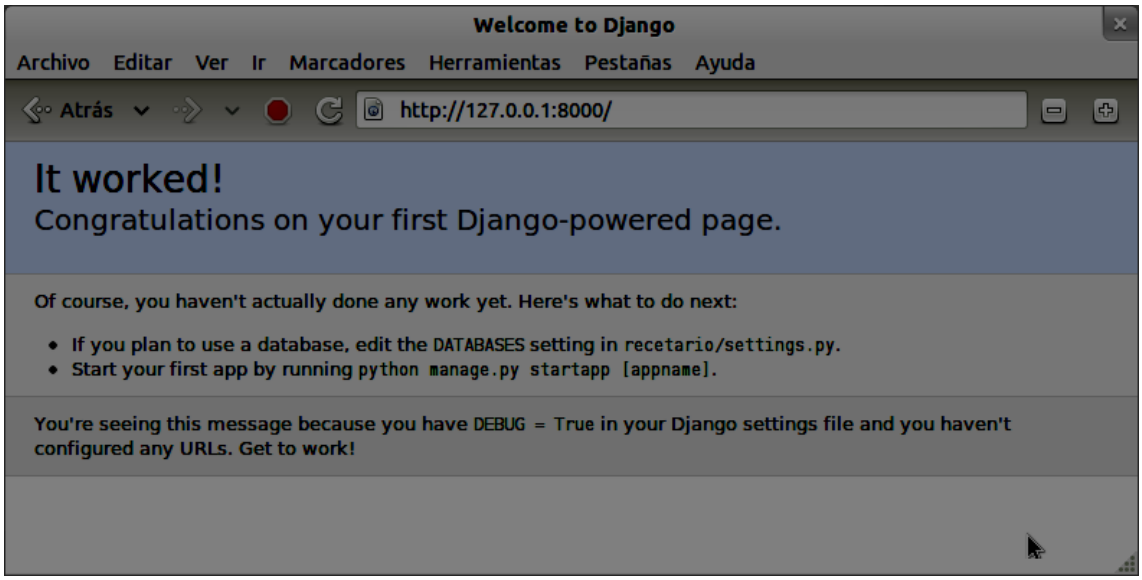
python manage.py runserver

Al ejecutar esa instrucción debemos visualizar un resultado como el siguiente:

```
Validating models...

0 errors found
Django version 1.4, using settings 'recetario.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Abrimos en el navegador web la dirección http://127.0.0.1:8000/ y deberemos ver lo siguiente:



7. Integración / Despliegue continuo

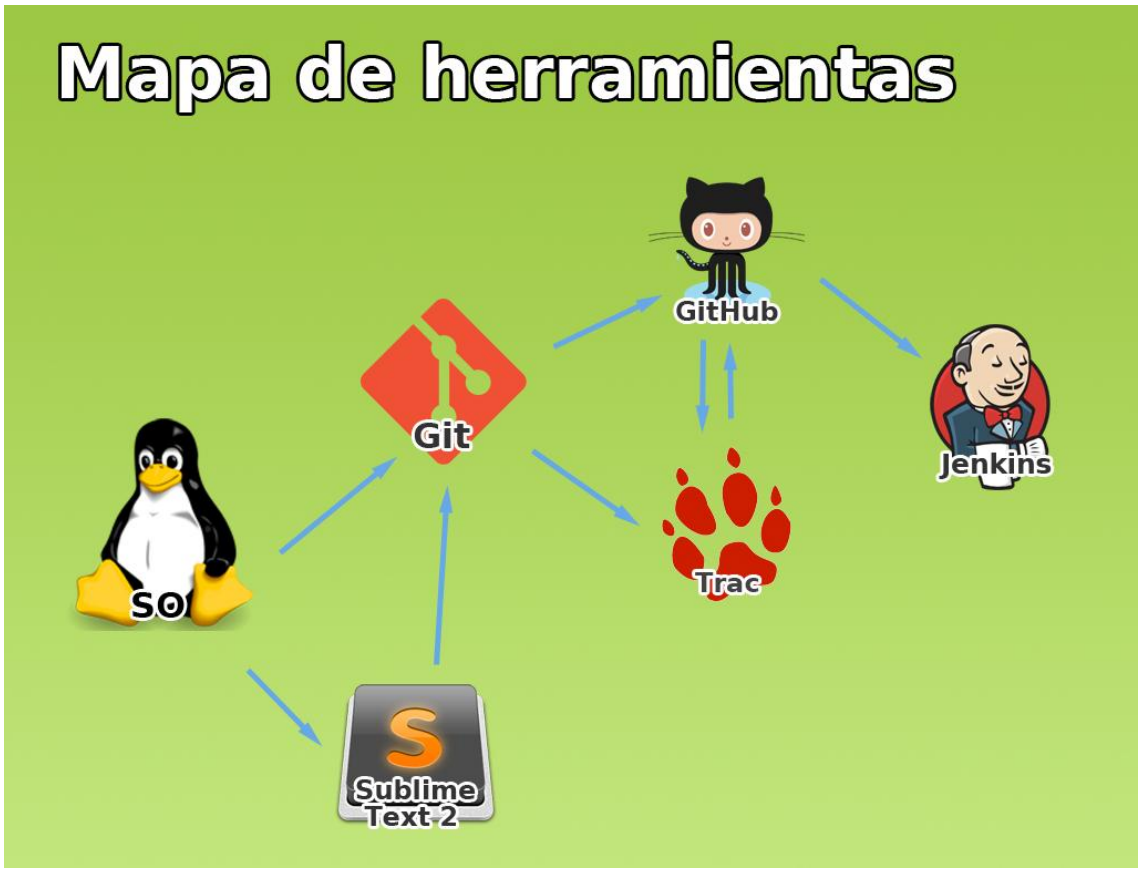
El equipo de Django utiliza Jenkins como plataforma de integración continua, que puede encontrarse en la siguiente url: http://ci.djangoproject.com/

En realidad, Jenkins está enfocado principalmente para el desarrollo en Java, pero gracias a su fácil extensibilidad a través de plugins, es capaz de operar con más lenguajes de programación. En el caso de Django, y dada la nula información que disponemos a este respecto en la documentación oficial, lo único que hemos podido constatar es que se realiza una batería de tests trabajando sobre distintos motores de bases de datos (mysql, sqlite, postgres) cada vez que se hace un commit.

En cuanto al despliegue continuo, ya hemos comentado anteriormente que el proyecto Django como tal no necesita despliegue alguno al tratarse de una mera herramienta de trabajo y no de un producto final destinado a ofrecer un servicio web. Por tanto, ni tiene mecanismos de despliegue continuo ni se requieren para un proyecto de estas características.

Este apartado se podría trabajar bastante más. No basta con decir que se usa Jenkins sin mucho más. Sería bueno consultar la literatura al respecto.

8. Mapa de herramientas



Esto hay que explicarlo, no vale poner sólo el mapa, es necesario explicar más.

9. Conclusiones

Como punto final a este documento, podemos concluir que Django es una poderosa herramienta, bien estructurada y con un gran equipo de desarrolladores muy activo detrás que permite construir de manera rápida, eficiente y limpia una aplicación web.

Siguiendo el principio DRY (Don't Repeat Yourself) (http://c2.com/cgi/wiki?DontRepeatYourself) podemos hacer uso de su sencilla interfaz de mapeo objeto-relación para construir rápidamente una base de datos abstrayéndonos de la tecnología utilizada (mysql, sqlite...), a la vez que su particular visión del Modelo Vista Controlador (MVC), el llamado Model Plantilla Vista (MTV) nos permite desarrollar de una manera estandarizada cualquier cosa que deseemos.

La gestión del proyecto es prácticamente impecable, con un ciclo de lanzamientos bastante continuado y con unas políticas de orden, claridad y limpieza del código muy estrictas para que la facilidad de mantenimiento del proyecto esté siempre presente.

En la página principal de Django podemos encontrar enlaces a sitios web de proyectos de éxito desarrollados con este framework, tales como Disqus, Instagram o Mozilla.

En resumen, un gran proyecto, con una gran comunidad, un buen soporte, una documentación excelente y una herramienta muy poderosa para el uso diario en el desarrollo.