**Notas de Django**

**¿Qué es Django?**

**Django** es un **framework** de desarrollo web de **código abierto**, escrito en **Python**, que respeta el patrón de diseño conocido como **Modelo–vista–template**. Fue desarrollado en origen para gestionar varias páginas orientadas a noticias de la World Company de Lawrence, Kansas, y fue liberada al público bajo una **licencia BSD** en julio de 2005; el framework fue nombrado en alusión al guitarrista de jazz gitano Django Reinhardt. En junio de 2008 fue anunciado que la recién formada Django Software Foundation se haría cargo de Django en el futuro.

La meta fundamental de Django es facilitar la creación de sitios web complejos. Django pone énfasis en el re-uso, la conectividad y extensibilidad de componentes, el desarrollo rápido y el principio **No te repitas** (DRY, del inglés Don't Repeat Yourself). Python es usado en todas las partes del framework, incluso en configuraciones, archivos, y en los modelos de datos.

**Arquitectura**

Aunque Django está fuertemente inspirado en la filosofía de desarrollo **Modelo Vista Controlador**, sus desarrolladores declaran públicamente que no se sienten especialmente atados a observar estrictamente ningún paradigma particular, y en cambio prefieren hacer "lo que les parece correcto". Como resultado, por ejemplo, lo que se llamaría "controlador" en un "verdadero" framework MVC se llama en Django "vista", y lo que se llamaría "vista" se llama "plantilla".

Gracias al poder de las capas mediator y foundation, Django permite que los desarrolladores se dediquen a construir los objetos Entity y la lógica de presentación y control para ellos.

**Presentación:**

Aquí se maneja la interacción entre el usuario y el computador. En Django, esta tarea la realizan el motor de plantillas y el cargador de plantillas que toman la información y la presentan al usuario (vía HTML, por ejemplo). El sistema de configuración de URLs es también parte de la capa de presentación.

**Control:**

En esta capa reside el programa o la lógica de aplicación en sí. En Django son representados por las vistas y los manipuladores. La capa de presentación depende de ésta y a su vez ésta depende de la capa de dominio.

**Mediator:**

Es el encargado de manejar la interacción entre el subsistema Entity y foundation. Aquí se realiza el mapeo objeto-relacional a cargo del motor de Django.

**Entity:**

El subsistema entity maneja los objetos de negocio. El mapeo objeto-relacional de Django permite escribir objetos de tipo entity de una forma fácil y estándar.

**Foundation:**

La principal tarea del subsistema foundation es la de manejar a bajo nivel el trabajo con la base de datos. Se provee soporte a nivel de foundation para varias bases de datos y otras están en etapa de prueba.

**Soporte de bases de datos**

Respecto a la base de datos, la recomendada es **PostgreSQL**, pero también son soportadas **MySQL** y **SQLite 3**. Se encuentra en desarrollo un adaptador para Microsoft SQL Server. Una vez creados los modelos de datos, Django proporciona una abstracción de la base de datos a través de su API que permite crear, recuperar, actualizar y borrar objetos. También es posible que el usuario ejecute sus propias consultas SQL directamente. En el modelo de datos de Django, una clase representa un registro de una tabla en la base de datos y las instancias de ésta serán las tuplas en la tabla.

**Soporte de servidores Web**

Como mencionamos en los requisitos, Django incluye un servidor web liviano para realizar pruebas y trabajar en la etapa de desarrollo. En la etapa de producción, sin embargo, se recomienda **Apache 2** con **mod\_python**. Aunque Django soporta la especificación **WSGI**, por lo que puede correr sobre una gran variedad de servidores como **FastCGI** o SCGI en Apache u otros servidores (particularmente **Lighttpd**).

***Fuente:***[*https://es.wikipedia.org/wiki/Django\_(framework)*](https://es.wikipedia.org/wiki/Django_(framework))

*Septiembre de 2018*

**Django de un vistazo**

Debido a que Django fue desarrollado en un ambiente de sala de redacción de ritmo rápido, fue diseñado para hacer tareas comunes de desarrollo web de forma rápida y fácil. Aquí se incluye una introducción informal de cómo escribir una aplicación web que utiliza una base de datos con Django.

El objetivo de este documento es darle suficiente información técnica para entender cómo funciona Django, pero no pretende ser un tutorial o referencia - ¡Aunque tenemos ambas cosas! Cuando esté listo para empezar un proyecto con Django, puede [comenzar con el tutorial](https://docs.djangoproject.com/es/2.0/intro/tutorial01/) o: doc:doc:ir directamente a la documentación más detallada </topics/index>.

**Diseñe su modelo**

Aunque es posible usar Django sin una base de datos, Django cuenta con un [mapeador objeto-relacional](https://en.wikipedia.org/wiki/Object-relational_mapping) en el que es posible definir la estructura de la base de datos utilizando código Python.

La [sintaxis de modelo de datos](https://docs.djangoproject.com/es/2.0/topics/db/models/) ofrece muchas formas de representar sus modelos – hasta ahora, ha resuelto una cantidad equivalente a años de problemas de esquemas de base de datos. Aquí hay un ejemplo rápido:

**mysite/news/models.py**

**from** **django.db** **import** models

**class** **Reporter**(models.Model)**:**

full\_name = models.CharField(max\_length=70)

**def** \_\_str\_\_(self)**:**

**return** self.full\_name

**class** **Article**(models.Model):

pub\_date = models.DateField()

headline = models.CharField(max\_length=200)

content = models.TextField()

reporter = models.ForeignKey(Reporter, on\_delete=models.CASCADE)

**def** \_\_str\_\_(self)**:**

**return** self.headline

**Instálelo**

A continuación ejecute la utilidad de la línea de comandos de Django para crear las tablas de base de datos de forma automática:

**$** python manage.py migrate

El comando **migrate** analiza todos sus modelos disponibles y crea cualquier tabla faltante en su base de datos, así como ofrece opcionalmente un [control de esquema mucho más rico](https://docs.djangoproject.com/es/2.0/topics/migrations/).

Disfrute de la API sin restricciones

Con esto, usted tiene una [API Python](https://docs.djangoproject.com/es/2.0/topics/db/queries/) rica y sin restricciones para acceder a sus datos. La API se crea sobre la marcha y no es necesario generar ningún código:

# Import the models we created from our "news" app

>>> **from** **news.models** **import** Article, Reporter

# No reporters are in the system yet.

>>> Reporter.objects.all()

<QuerySet []>

# Create a new Reporter.

>>> r = Reporter(full\_name='John Smith')

# Save the object into the database. You have to call save() explicitly.

>>> r.save()

# Now it has an ID.

>>> r.id

1

# Now the new reporter is in the database.

>>> Reporter.objects.all()

<QuerySet [<Reporter: John Smith>]>

# Fields are represented as attributes on the Python object.

>>> r.full\_name

'John Smith'

# Django provides a rich database lookup API.

>>> Reporter.objects.get(id=1)

<Reporter: John Smith>

>>> Reporter.objects.get(full\_name\_\_startswith='John')

<Reporter: John Smith>

>>> Reporter.objects.get(full\_name\_\_contains='mith')

<Reporter: John Smith>

>>> Reporter.objects.get(id=2)

Traceback (most recent call last):

...

DoesNotExist: Reporter matching query does **not** exist.

# Create an article.

>>> **from** **datetime** **import** date

>>> a = Article(pub\_date=date.today(), headline='Django is cool',

... content='Yeah.', reporter=r)

>>> a.save()

# Now the article is in the database.

>>> Article.objects.all()

<QuerySet [<Article: Django **is** cool>]>

# Article objects get API access to related Reporter objects.

>>> r = a.reporter

>>> r.full\_name

'John Smith'

# And vice versa: Reporter objects get API access to Article objects.

>>> r.article\_set.all()

<QuerySet [<Article: Django **is** cool>]>

# The API follows relationships as far as you need, performing efficient

# JOINs for you behind the scenes.

# This finds all articles by a reporter whose name starts with "John".

>>> Article.objects.filter(reporter\_\_full\_name\_\_startswith='John')

<QuerySet [<Article: Django **is** cool>]>

# Change an object by altering its attributes and calling save().

>>> r.full\_name = 'Billy Goat'

>>> r.save()

# Delete an object with delete().

>>> r.delete()

**Una interfaz administrativa dinámica: no es sólo andamios; es toda una casa**

Una vez definidos sus modelos, Django puede crear automáticamente una [interfaz administrativa](https://docs.djangoproject.com/es/2.0/ref/contrib/admin/) profesional y lista para la producción; un sitio Web que le permita a los usuarios autenticados añadir, modificar y eliminar objetos. Es tan fácil como registrar su modelo en el sitio administrativo:

**mysite/news/models.py**

**from** **django.db** **import** models

**class** **Article**(models.Model)**:**

pub\_date = models.DateField()

headline = models.CharField(max\_length=200)

content = models.TextField()

reporter = models.ForeignKey(Reporter, on\_delete=models.CASCADE)

**mysite/news/admin.py**

**from** **django.contrib** **import** admin

**from** **.** **import** models

admin.site.register(models.Article)

La filosofía aquí es que su sitio es editado por un equipo, un cliente o tal vez sólo por usted y usted no quiere tener que lidiar con la creación de interfaces de backends, sólo quiere administrar el contenido.

Un flujo de trabajo típico en la creación de aplicaciones Django es crear modelos y poner a funcionar los sitios administrativos lo más rápido posible para que su personal (o clientes) puedan empezar a ingresar los datos y luego, perfeccionar la forma como los datos se presentan al público.

**Diseñe sus URLs**

Un esquema de URL limpio y elegante es un detalle importante en una aplicación web de alta calidad. Django fomenta un diseño de URL hermoso y no añade ningún elemento innecesario en las URLs, como **.php** o **.asp**.

Para diseñar las URLs de una aplicación, usted debe crear un módulo de Python llamado URLconf. Esta es una tabla de contenidos para su aplicación que contiene un mapeo sencillo entre patrones URLs y funciones callback de Python. Las tablas URLconf también sirven para separar las URLs del código Python.

Así es como se vería una URLconf para el ejemplo **Reporter**/**Article** anterior:

**mysite/news/urls.py**

**from** **django.urls** **import** path

**from** **.** **import** views

urlpatterns = [

path('articles/<int:year>/', views.year\_archive),

path('articles/<int:year>/<int:month>/', views.month\_archive),

path('articles/<int:year>/<int:month>/<int:pk>/', views.article\_detail),

]

El código anterior mapea las URLs a funciones callback de Python («vistas»). Las cadena de caracteres de la ruta usa etiquetas para «capturar» los valores de las URLs. Cuando un usuario solicita una página Django pasa por cada patrón en orden y se detiene en el primero que coincide con la URL solicitada. (Si ninguno de ellos coincide Django llama una vista del caso especial 404). Esto es increíblemente rápido debido a que las rutas son compiladas como expresiones regulares en tiempo de carga.

Una vez que una de las expresiones regulares coincide, Django llama a la vista determinada, que es una función Python. Cada vista recibe un objeto de consulta – que contiene metadatos de consulta – y los valores capturados en la expresión regular.

Por ejemplo si un usuario solicitara la URL «/articles/2005/05/39323/» Django llamaría la función **news.views.article\_detail(request, year=2005, month=05, pk=39323)**.

**Escriba sus vistas**

Cada vista es responsable de hacer una de dos cosas: retornar un objeto HttpResponse que incluye el contenido de la página solicitada, o levantar una excepción como Http404. El resto depende de ti.

Generalmente, una vista recupera los datos de acuerdo con los parámetros, carga una plantilla y crea la plantilla con los datos recuperados. Aquí está una vista de ejemplo para el year\_archive de arriba:

**mysite/news/views.py**

**from** **django.**shortcuts **import** render

**from** **.models** **import** Article

**def** year\_archive(request, year):

a\_list = Article.objects.filter(pub\_date\_\_year=year)

context = {'year': year, 'article\_list': a\_list}

**return** render(request, 'news/year\_archive.html', context)

Este ejemplo utiliza el [sistema de plantillas de Django](https://docs.djangoproject.com/es/2.0/topics/templates/) que tiene varias características potentes, pero se esfuerza por mantenerse lo suficientemente simple como para que los que no son programadores lo puedan utilizar.