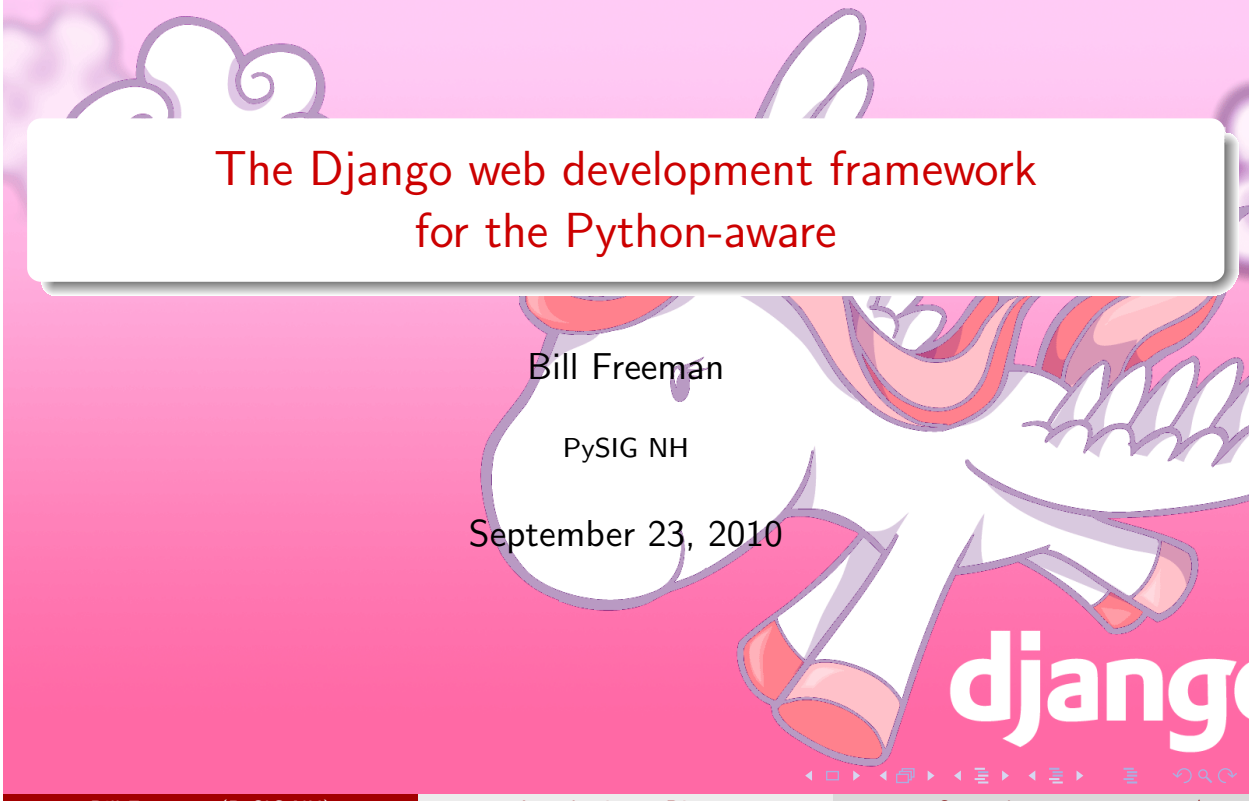
The Django web development framework for the Python-aware

Bill Freeman

PySIG NH

September 23, 2010



Introduction

Django is a web development framework. While most parts are optional, this means that it:

- Accepts requests (typically HTTP GET and POST)
- Garners information from URLs
- Interacts with a database (CRUD)
- Composes a response (templates, Python)
- Does session management
- Helps with pesky details, e.g.; CSRF

"Front-end" Server

- **Django**'s native client-facing interface is WSGI (Web Services Gateway Interface).
- **Django** includes a "development" server that is convenient for, well, development, connecting HTTP to WSGI (with a few extras).
- In deployment, however, one *should* use a server like Apache or NGiNX to meet the TCP/IP interface, do SSL, and to serve truly static files such as CSS, JS, images, PDFs, etc.
- Now that mod_python is dead, mod_wsgi is pretty much the universal answer for connecting the front-end server and **Django**.
- There are WSGI interfaces for other front ends, or connectors that hide WSGI under the hood.

Request Path Parsing

- **Django** parses the request URL's "path" using a "**urlconf**".
- A **urlconf** has a list (or equivalent) of "**urlpattern**" objects. It applies each **urlpattern** in turn to the path the **urlconf** was given against, stopping when one succeeds, returning its result. (If no **urlpattern** succeeds, the **urlconf** fails, see below.)
- A **urlpattern** has a regular expression (**regex**). If a the **regex** of a **urlpattern** doesn't match the path string, the **urlpattern** fails.
- If the **regex** matches then:
  - If the **urlpattern** has a **view function** then the **urlpattern** succeeds, returning the **view function** and maybe other stuff.
  - Otherwise the **urlpattern** has a sub-**urlconf**. The **urlpattern** removes the portion of the path that it matched and applies its **urlconf** to the remainder. The **urlpattern** succeeds if its **urlconf** does.
- If the top level **urlconf** fails, **Django** returns a 404 response (page not found).

Gathering View Function Arguments

A **urlpattern**'s **regex** may include "groups" which are used as arguments to the **view function**.

- Unnamed groups are used as positional arguments:

r'digest/month/(20\d\d)-(\d\d?)/'

would match 'digest/month/2010-9/' and would collect '2010', and '9' as positional arguments.

- Named groups are used as keyword arguments:

r'digest/month/(?P<year>20\d\d)-(?P<month>\d\d?)/'

would match the same string, but pass '2010' as the **year** argument, and '9' as **month**.

Each urlpattern may also have a dictionary of extra keyword arguments.

View functions

- A **view function** is an ordinary Python function.
- An "HttpRequest" object is always passed as the first positional argument.
- The function may accept additional arguments. Any that are not optional must be supplied by means of the **urlpattern**s involved in choosing the **view function**.
- The HttpRequest object has, among other things:
  - A "method" attribute, identifying the kind of HTTP request being processed.
  - A "GET" attribute, a mapping of query parameter names to values.
  - A "POST" attribute, a mapping of POST parameter names to values.
- A **view function** typically returns an HttpResponse object. It will be returned to the front end for transmission to the client. HttpRedirect objects are another possibility.

Templates, variable references, context variables

- **Django** templates are mostly HTML.
- Special constructs begin with {{, {% or {#, and end, respectively, with }}, %} or #} (variable references, tags, and comments). The beginning and ending sequences must be on the same line.
- Variable reference inserts something into the document:

{{ foo }}

Inserts unicode() of the value of the **foo** "context variable".

- Most "context variables" are provided by the code that wants the template rendered.
- The variable can be de-referenced using dot:

{{ foo.bar }}

Gets the bar attribute of foo, calling if it is a method, or gets foo['bar'] if foo is a mapping. If foo were a list, foo.1 would get its second element.

Templates — filters

- Variables can be "filter"ed:

{{ foo.bar|upper }}

Makes the result upper case. You can use multiple filters in a pipeline.

- Some filters can take an argument:

{{ foo.start_date|date:"D d M Y" }}

Foo's attribute, presumably a date or datetime, is formatted as you like.

- Normally variable references are escaped to prevent inadvertent insertion of HTML markup from the variable value:

{{ foo.bar|upper|safe }}

Some filters mark their result to avoid the escaping. **safe** is one such, and that is its only purpose.

- You can write your own filters in Python.

Templates — tags

- Tags *MAY* provide text output:

{% now "D d M Y" %}

Shows today's date.

- Tags *MAY* be paired with a closing tag:

{% for f in foo.fields %}{{ f.name }}: {{ f.value }}
{% endfor %}

Iterates foo.fields, as you might expect. Any previous value of the context variable **f** is restored after the endfor. You can access the loop counter in various ways using the "forloop" context variable. The boolean "forloop.last", plus the "if" tag, could be used to eliminate the
 HTML tag the last time through.

- Tags can take a variable number of arguments.
- You can write your own tags in Python. It is, however, harder than writing filters.

Templates using other templates

- The **include** tag does what it sounds like, inserting the rendered output of another template into the including template. (This actually is much less used that the extends tag, see below.)
- The **extends** tag and **block** tag implement "template inheritance".
- The **extends** tag must be first in a template, and can occur only once. The base template (the one mentioned in the **extends** tag) is rendered instead of the derived template (the one using **extends**).
- A base template can also use the **extends** tag, to any depth.
- The **block** tag takes a name argument, and, with the **endblock** tag, defines a "named block".
- The content of a named block in a derived template replaces the content of any block with the same name in its chain of bases.
- "base" templates can provide site-wide common features. Templates a level up can provide section specific differences. Top level templates implement the final page type details.

Form support

Django aids in the production of HTML forms.

- You create a python sub-class of **Django** "Form" class.
- Most class attributes are instances of **Django** "form.Field" classes.
- A class instance can render itself as INPUT and SELECT tags.
- Make the instance a template "context" variable, say **form**.
- The template says, for example, (within a FORM tag):
  {{ form.as_p }}
- A dictionary of initial field values can be provided at instantiation.
- When the form is submitted, the **view function** passes request.POST (or request.GET) to a new instantiation of the class, and then says:
  if(form.is_valid()):
- Each Field knows how to validate the submitted data. If a field is invalid, and the form instance is rendered again, an error string is shown near the INPUT.
- A valid form provides a dictionary of submitted values for the **view function** to use.

Models

- "Model" classes correspond to database tables.
- Model class instances correspond to database rows.
- Many model class attributes correspond to database columns.
- Corresponding model class instance attributes hold python objects representing database values.
- Model Field attributes, along with the **Django** database back end code, know how to convert between the python representation of values and the SQL representation of those values.
- Model Fields validate instance values before saving an instance to the database, when the "save" method of the Model instance is called.
- Model Fields include support for foreign key (many to one) and many to many relationships (the m2m join tables are handled by **Django**).
- Model Forms are Forms that use Model classes to define the form's fields, can be initialized from a Model instance, and know how to transfer form field data to Model instance fields.

Querysets

- A "queryset" remembers selection criteria for a given Model.
- No database query is made until a queryset is "evaluated".
- A common "evaluation" is to iterate over a queryset which generates a series of Model instances.
- A queryset can be refined, which really returns a new queryset with both the original queryset's selection criteria, and the "refinements".
- Selection criteria are specified using keyword arguments to the queryset methods "filter", "exclude", or "get".
- The name of the keyword argument provides the name of the field to be tested, and, optionally, what kind of test (if other than equality) to make against the value of the keyword argument.

```
ModelName.objects.filter(end_date__gte=datetime.datetime.now())
```

Selects rows, which, according to their end_date column, have already ended.

Admin

django.contrib.admin is a "plug-able app":

- Easy to generate a section of an admin interface from a model.
- Can create, delete, edit and search model instances/table rows without writing your own urlpattern, **view function**, forms, and templates.
- Easy to customize what is shown, what is editable, how it is grouped, and what is searchable.

There are many other plug-able apps, including CMS, blog, wiki, tagging, UI enhancing, etc. Some, like admin, are bundled with **Django**, but nearly all are available using pip or easy_install. You may still want to customize one, or write your own to get exactly the performance you want: this is Open Source, so have at it.

Middleware

- Settings specifies a "stack" (sequence, really) of middleware classes.
- The classes, in order, get a crack at the request before url parsing. They can modify the request object, gather information, return a response, including a redirect, or do nothing if this request isn't interesting.
- The classes get a second crack after url parsing, but before the **view function** is called. (There is middleware, for example, that checks whether the **view function** has the attribute applied by the "login_required" decorator.)
- When a **view function** returns a response, the middleware, in reverse order (popping the stack?) gets a crack at the response, before it is returned to the front end, on the way to the client.
- If, instead, the **view function** raises an exception, the middleware gets, in reverse order, a crack at the exception.

Template Context Processors

- A **view function** may choose to pass a "request context" to provide the values of template variables for a template rendering.
- RequestContext, which creates a request context object, is passed the request object, plus any explicit context variable definitions the **view function** wishes to provide.
- The RequestContext constructor offers the request object to each template context processor listed in settings.py.
- Any of the template context processors can return a dictionary of additional context variable definitions.

Not covered

- Template search path.
- Writing your own template tags and filters.
- Additional manage.py commands.
- Writing additional manage.py commands.
- Multiple databases.
- Non-SQL databases.
- Using legacy databases.
- Shortcuts.
- Generic views.

More information

- http://docs.djangoproject.com/
- http://groups.google.com/group/django-users
- http://pypi.python.org/pypi/virtualenv
- http://www.python.org/dev/peps/pep-0370/
- And, of course, http://python.org/