Giving your WebApp a pony!

An introduction to Django, the python powered webapp framework

Nick Burch

Senior Developer

Torchbox Ltd

Introduction

- Django - The web framework for perfectionists with deadlines
- Django makes it easier to build better web apps more quickly and with less code, and happens to be in Python
- Does most of what you want straight out of the box, but you can change anything you want later without much fuss

Django Pony from http://www.djangopony.com/

Features

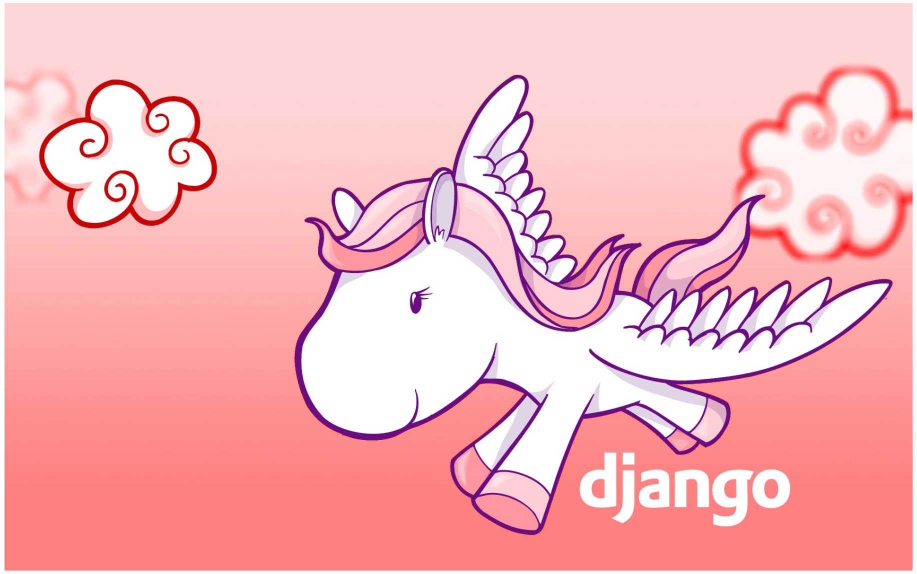
- Easy to get started with
- Still easy months later!
- Excellent documentation, both inline and on http://docs.djangoproject.com/
- Makes common web application tasks easy, but allows complicated things too
- Provides most things out of the box
- Lets you do your own thing if you want

Quick Start

- A project is a collection of applications, which make up your site
- The project is what you run, and handles the settings
- An application is where you code goes, eg database models, views
- Simple projects have one application, complex ones often want a few. You'll also use several standard apps too

Quick Start 2

- Create a new project
```
django-admin.py startproject aceu
```
- Configure up your project, by editing settings.py, and maybe including a local_settings.py too
- Start your first app
```
./manage startapp con
```


django

Quick Model Intro

- A model is used to power the ORM
- Models can be very simple, but you can do powerful things too
- A model has a number of fields, into which data is stored

```
class Speaker(models.Model):
    name = models.CharField(max_length=100, help_text="Full Name", 
unique=True)
    biography = models.TextField(help_text="A 200 word bio for the 
speaker")
```

More on the model

- Now we need to create the database objects for it - ./manage.py syncdb
- Add another model

```
class Talk(models.Model):
    name = models.CharField(max_length=100, unique=True)
    abstract = models.TextField(help_text="A 50 word abstract")
    start = models.DateField(help_text="Start date+time of the talk")
    end   = models.DateField(help_text="End date+time of the talk")
    speaker = models.ForeignKey(Speaker)
```

- Take a look in the database to see it!

The Admin Interface

- Django comes with a very nice admin interface out of the box
- Enable it in your settings, and run syncdb again to set things up
- Enable it in url.py, and go to /admin/
- A nice, simple, free admin interface awaits!

Enabling Admin for Us

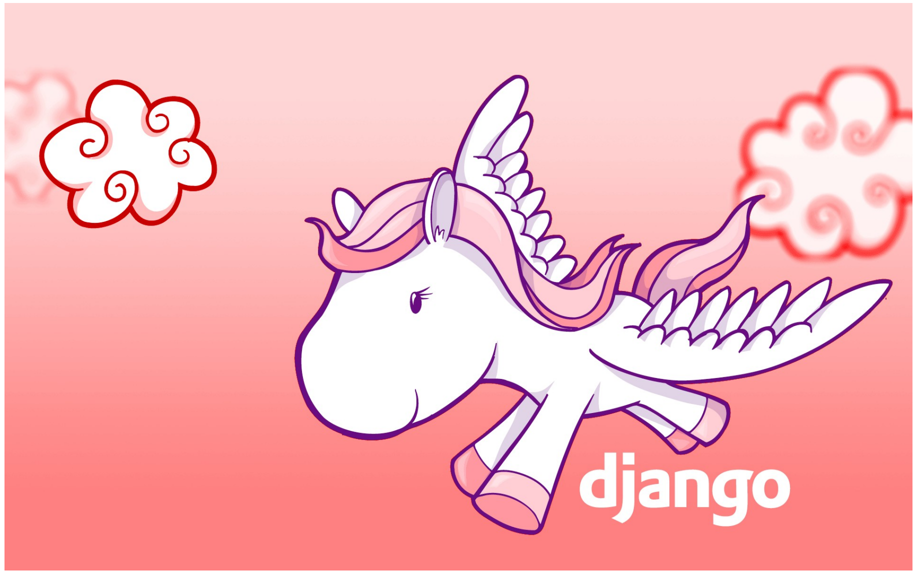
- You need to explicitly request the admin for a model, but it's very easy
- For basic admin, just register each model in turn
- For complex admin features, including customising the fields, define your own admin class and register that

Enabling admin cont.

```
from django.contrib import admin
from con.models import *

admin.site.register(Speaker)
admin.site.register(Talk)
```

- To control ordering etc, add a meta section

```
class Speaker(models.Model):
    .....
    class Meta:
        ordering = ['name']

class Talk(models.Model):
    .....
    class Meta:
        ordering = ['name', 'speaker__name']
```


django

Views

- The url mapping routes requests to views
- Views return content / redirects

```
from django.http import \
        HttpResponseRedirect, HttpResponseForbidden, HttpResponse

def welcome(request):
   return HttpResponseRedirect(“/start/”)

def start(request):
   return HttpResponse(“<html></html>”>
```

Handly view helpers

- Views work best when returning templates with variables, and django makes this easy

```
from django.http import HttpRequest, HttpResponseRedirect, Http404, \
    HttpResponseForbidden, HttpResponse
from django.shortcuts import render_to_response
from django.template import RequestContext

def render(request, template_name, data_dict=None):
    assert isinstance(request, HttpRequest)
    return render_to_response(
        template_name, data_dict or {},
        context_instance=RequestContext(request)
    )
def welcome(request):
    return render(request, “welcome.html”)
```

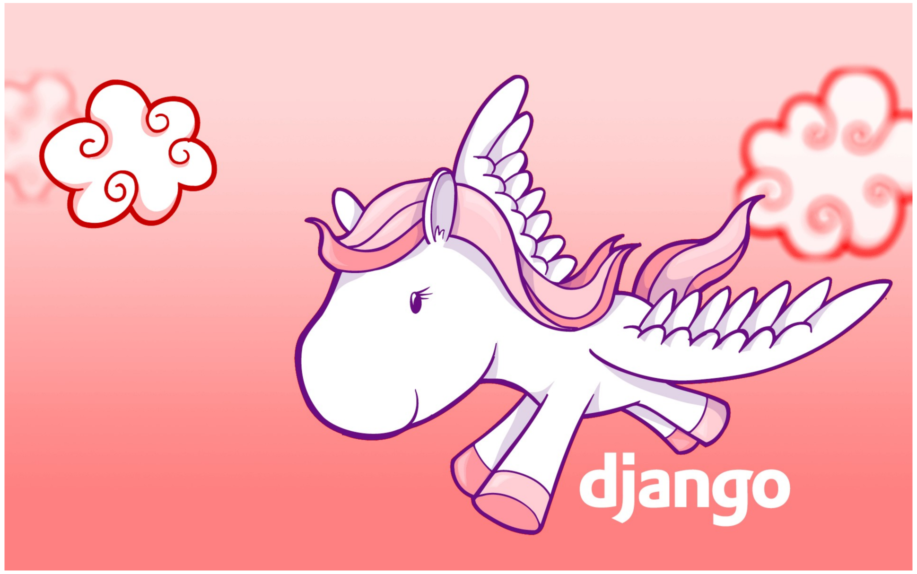
Variables and Content Types

- request.GET, request.POST and request.META are dictionaries
- eg request.GET[“format”]
- eg request.GET.get(“format”, “default”)

- HttpResponse can take an optional content type argument

Example

```
def display_talk(request, talk_id):
    talk = get_object_or_404(Talk, id=talk_id)

    if request.GET.get("format",None) == "json":
        return download_talk(request, talk)
    return render(request, 'talk.html', {"talk":talk})

def download_talk(request, talk):
    from django.core import serializers
    json = serializers.serialize("json", [talk])
    return HttpResponse(json, "text/javascript")
```


django

Model Forms

- Automatically build a form object from your model, and its constraints

```
from django import forms

from con.models import *

class SpeakerForm(forms.ModelForm):
    class Meta:
        model = Speaker
        #exclude = ("name")

class TalkForm(forms.ModelForm):
    class Meta:
        model = Talk
        exclude = ("start","end")
```

Form + View

- Forms handle validation
- Model forms also handle creating / updating model objects from data

```
def edit_speaker(request, speaker):
    if speaker_id == "new":
        speaker = None
    else:
        speaker = get_object_or_404(Speaker, id=speaker_id)

    # Create our form, using the existing speaker if given
    form = SpeakerForm(request.POST or None, instance=speaker)

    if request.POST and form.is_valid():
        speaker = form.save()
        return HttpResponseRedirect("/speakers/%s/" % speaker.id)
    return render(request, 'edit_speaker.html', {
        "speaker": speaker, "form": form,
    })
```

ApacheCon

FIG. 2.
PONY "MAGIC"

Leading the Wave
of Open Source

Django ORM

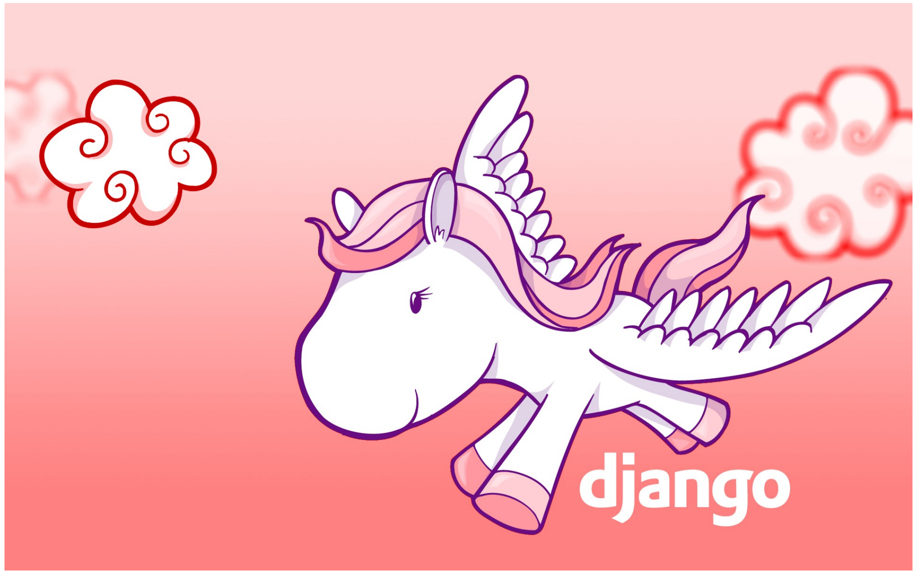
- Allows you to create, update, save, search and fetch
- When calling save, your object will be validated
- .get(filter) returns on object
- .filter() and .all() return multiple
- <model class>.objects is your access to the ORM mapper for that model

Example

```
s = Speaker()
s.save()
s.name = "AlsoNick"
s.biography = "Did stuff"
s.save()

s2 = Speaker(name="AlsoNick2", biography="More stuff")
s2.save()

s.name = "StillAlsoNick"
s.save()
s.name

s1 = Speaker.objects.get(id=1)
s1
s2 = Speaker.objects.get(name="AlsoNick2")
s2

Speaker.objects.all()
Speaker.objects.count()
Speaker.objects.filter(name__exact="Nick")
Speaker.objects.filter(name__contains="Nick")
            Talk.objects.filter(start__gt=datetime.date(2009,1,5))
```

Fixtures

- Fixtures are serialized database objects, for easy data population
- Can be loaded on demand, or automatically (initial_data.json)
- Normally in JSON, but note that the parser is strict – no comments
- ./manage.py provides two commands – loaddata and dumpdata


django

URL mapping - urls.py

- Powerful mapping between urls and view functions
- Uses regular expression to match, and pick arguments. Matches in order
- Includes allow per-app url patterns
- Can do powerful stuff with capture and functions
- Everything pointed to is a view, and should return a HttpResponse

Example

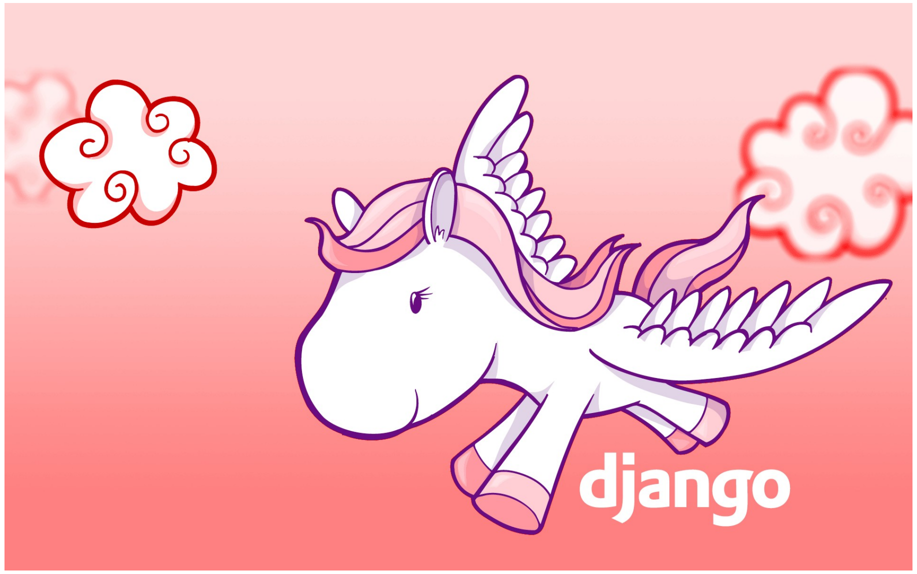
```
urlpatterns = patterns('aceu.con.views',
    (r'^talks/(\d+)/$', 'display_talk'),
    (r'^talks/$',       'pick_talk'),

    (r'^speakers/new/$', 'edit_speaker', {"speaker_id":"new"}),
    (r'^speakers/(\d+)/edit/$', 'edit_speaker'),
    (r'^speakers/(\d+)/new/$', 'edit_speaker'),
    (r'^speakers/(\d+)/$', 'display_speaker'),
    (r'^speakers/$',       'pick_speaker'),

    (r'^$', 'welcome'),
)


urlpatterns += patterns('',
    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),
)
```

Forms Validation

- The key method for your own forms is the clean(data) method
- This takes in the raw data, and returns either the cleaned form (eg dates, integers, stripped strings), or raises validation errors
- Model Forms do most of this for you already

Form Fields

- Similar to model fields in many way, but with extra validation support
- All the standard fields are listed at http://docs.djangoproject.com/en/dev/ref/forms/fields/
- Even with Model Forms, you can override definitions if you want, eg store as CharField, but validate as an email


django

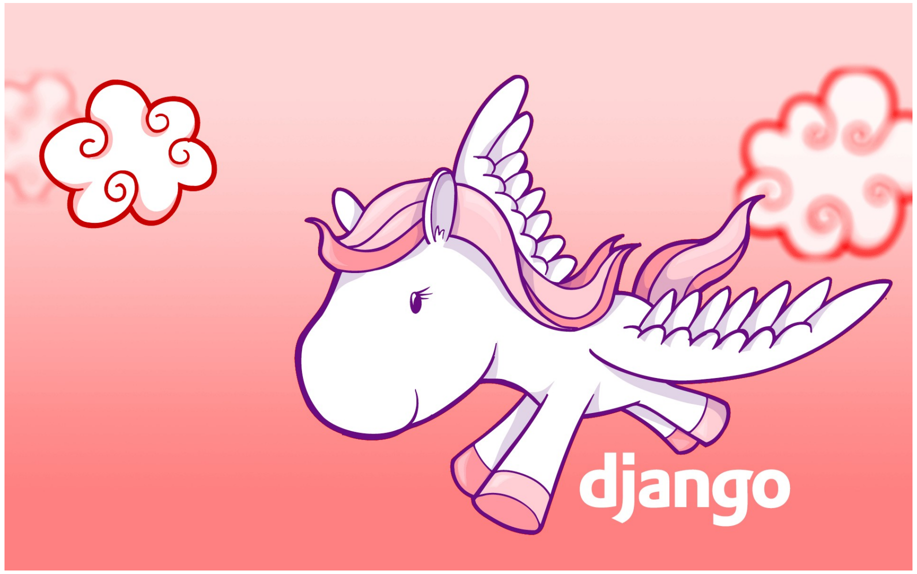
More on the Admin

- You can customise the form used if you'd like
- Objects linked by a foreign key can be edited inline
- You can have two different forms, one for adding, another for editing. The authentication system is one example

Admin Filtering, Permissions

- You can define filters, customise the ordering, and decide what fields to show in the admin
- Permissions can be set on all objects, on a per-user or per-group basis
- Superusers get access to everything
- You need to have the staff flag set to get into the admin though

ApacheCon



Authentication

- Django comes with a very nice authentication system, which the admin uses
- Enable django.contrib.auth as an installed app, and turn on the authentication middleware (normally done by default)
- Users can now be created

Authentication Models

- django.contrib.auth.models has these
- Users have names, email, username, password and staff/superuser
- Groups hold users
- Permissions apply per model, for adding, updating or deleting
- AUTH_PROFILE_MODULE allows for extra properties to be attached

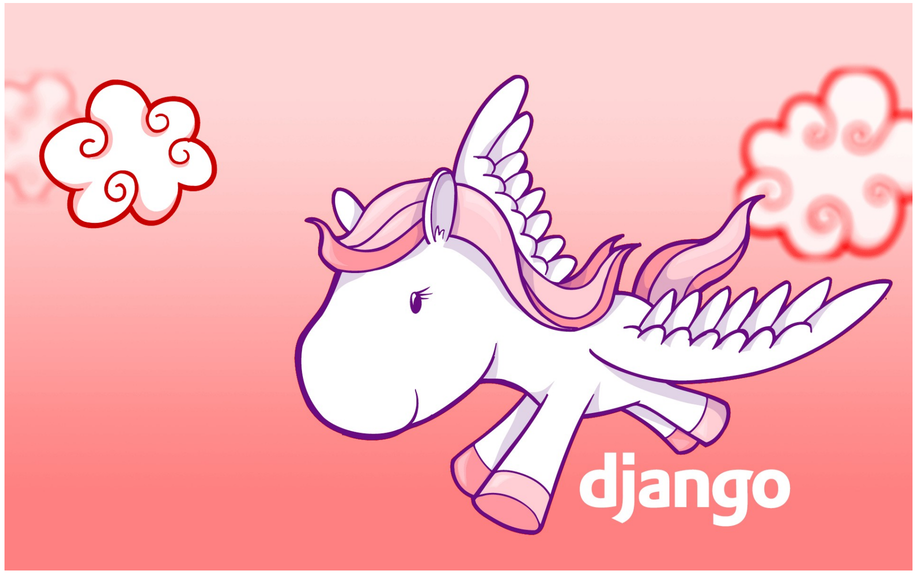
Middleware, Decorators

- The authentication middleware will provide request.user to all your views, containing either the logged in user, or an AnonymousUser object
- Using decorators, you can restrict views to logged in or staff users

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ....
```

Logging in and out

- django.contrib.auth provides ways to test the authentication of a user (authenticate)
- It lets you mark a user's session as logged in as a user – login
- It lets you log a user out – logout
- It provides handy help views and forms if you don't want to do it all yourself


django

Templates

- The template language is quite a bit simpler than many
- You can't do very complex if blocks for example, but it does mean your logic tends to stay out of your templates!
- For more complex things, you can parse variables through filters, or write your own tags

Template Basics

- Output variables with {{foo}}
- Do special things with variable output, using filters, eg <p>There are {{foo}} thing{{foo|pluralize}}</p>
- {% if foo %}<p>Foo!</p>{% else %}<p>Not foo!</p>{% endif %}
- {% for foo in bar %}<p>{{foo}}</p>{% endfor %}

Templates Inheritance, Paths

- The template can be broken up into blocks, which may or may not be overriden by child templates
- Generally allows for clean templates

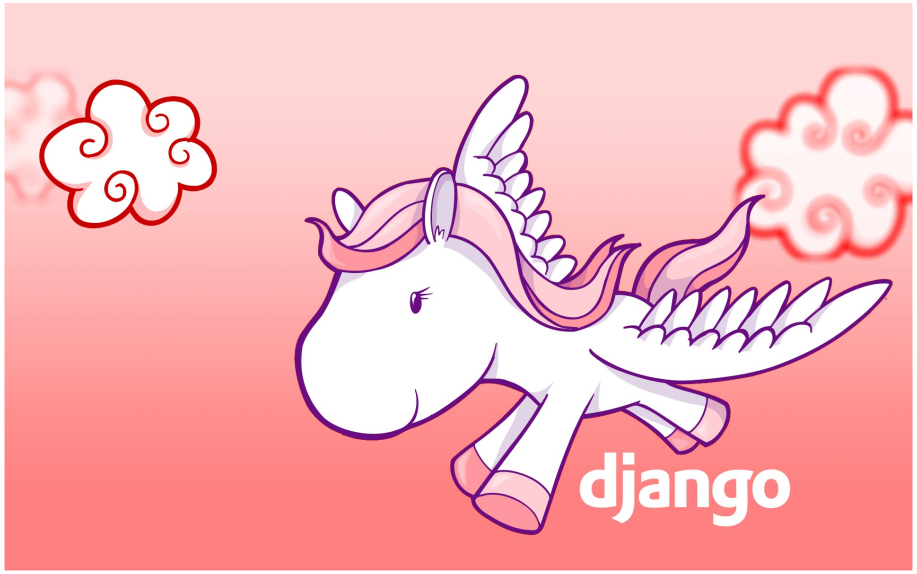
- Normally templates go in /templates/
- You need to list the directory in TEMPLATE_DIRS in settings though

Common Tags

- for x in ... / endfor
- for x,y in ... .items / endfor
- for x in ... / empty / endfor
- if x / else / endif
- ifequal x y / endif
- url path.to.view arg1,arg2

Common Filters

- {{foo|capfirst}}
- {{foo|floatformat}} {{foo|floatformat:3}
- {{foo|first}} {{foo|last}}
- {{foo|linebreaksbr}}
- car{{foo||pluralize}} cherr{{pluralize:”y,ies”}}
- {{foo|date:"D d M Y"}}


django

Database Migrations

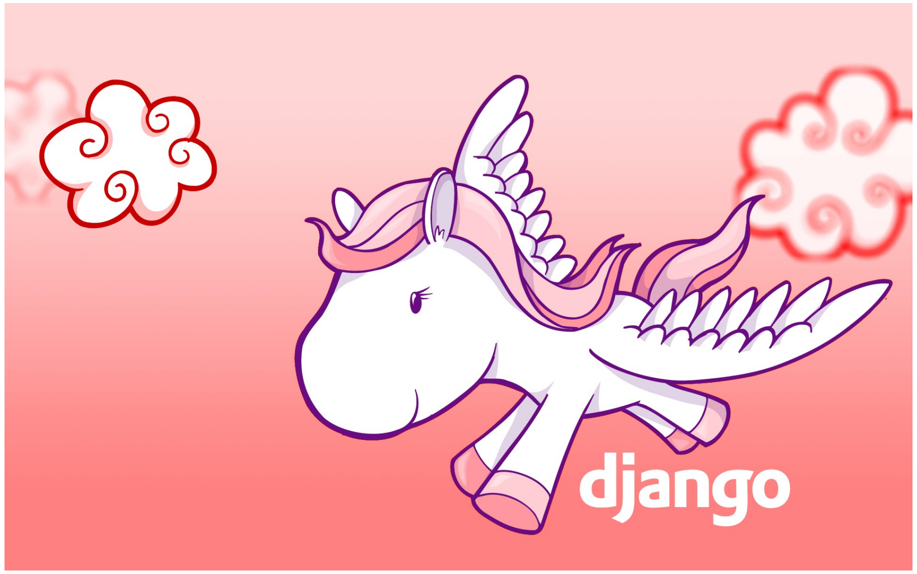
- Syncdb only handles adding new tables in
- South provides an easy to use, comprehensive database migrations framework
- Generating a migration for a new model is very easy
- Changing database structure is easy too

South

- South provides methods for changing your database structure, and manipulating the contents
- All of django is available too, as is raw SQL if you really want
- Handles running migrations out of order, missing migrations, tracks when migrations were applied etc
- Tries to learn from mistakes of rails

Examples

```
from django.db import models
from aceu.models import *

class Migration:
  def forwards(self):
    # Model 'JavascriptError'
    db.create_table('aceu_javascripterror', (
        ('id', models.AutoField(verbose_name='ID', primary_key=True,
auto_created=True)),
        ('when', models.DateTimeField()),
        ('message', models.TextField()),
        ('url', models.TextField()),
    ))
    db.send_create_signal('aceu', ['JavascriptError'])
  def backwards(self):
    db.delete_table('aceu_javascripterror')

class Migration:
  def forwards(self):
    db.add_column("aceu_tip", "icon", models.CharField(max_length=50,
blank=True, null=True))
  def backwards(self):
    db.drop_column("aceu_tip", "icon")
```


django

Caching

- Django supports a number of caching backends, including memcached, database, and filesystem
- The whole site can be cached
- Views can declare themselves to be cached
- Template fragments can declare themselves to be cached
- The caching api is available in code

Middleware

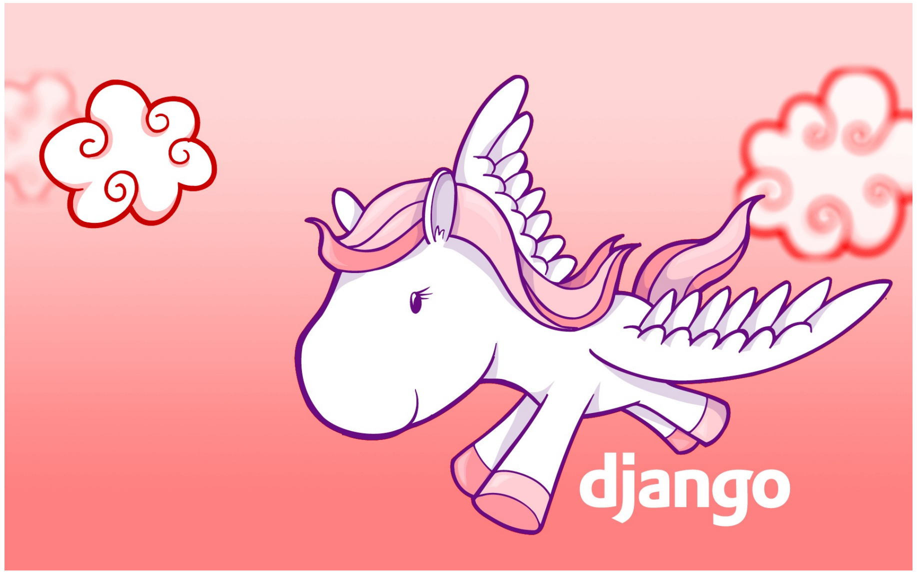
- Middleware runs on the input and output to your application, has a very lightweight interface, is run in order
- Can easily setup various objects, eg load objects based on session keys
- Can perform operations on outputs, eg compression on caching headers
- Lots of helpful middleware is provided by django

Examples

```
class FormatMiddleware(object):
    def process_request(self, request):
        request.format = request.REQUEST.get("format", "django")

class ApiUserMiddleware(object):
    def process_request(self, request):
        if request.REQUEST.get('_api_user', None):
            try:
                request.user = 
AltUser.objects.get(external_id=request.REQUEST['_api_user']).user
            except AltUser.DoesNotExist:
                pass

class ConsoleExceptionMiddleware:
    def process_exception(self, request, exception):
        try:
            errlog = open("/tmp/onzolog", "a")
            exc_info = sys.exc_info()
            print >> errlog, 
'\n'.join(traceback.format_exception(*(exc_info or sys.exc_info())))
            errlog.close()
        except:
            pass
```

Template Context Processors

- A bit like middleware, for templates
- Runs before the template is rendered
- Can inject extra variables into the template
- Default ones include injecting the authentication details in
- Allows for more lightweight views, and more commonality in templates


django

Serialisation

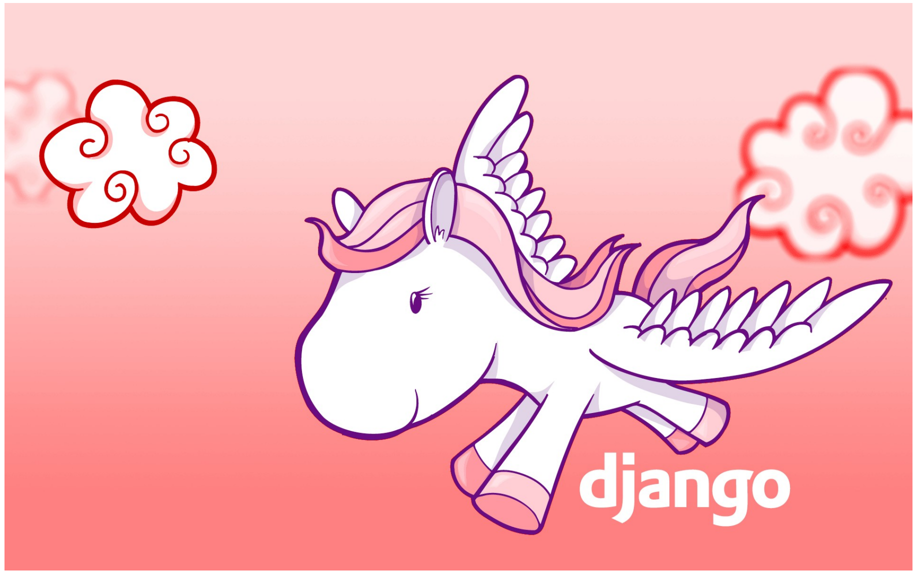
- Django provides easy ways to serialise your database objects to XML and JSON, and read them back in again
- Commonly used with fixtures, to populate the database
- However, makes writing api services quite easy
- Can easily restrict to just some fields

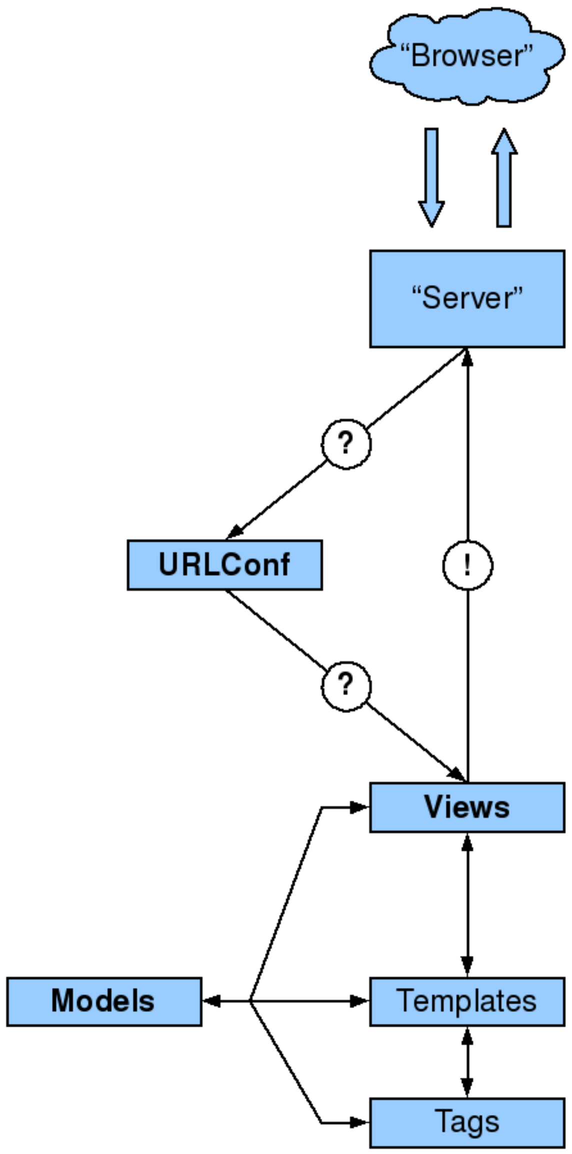
Testing

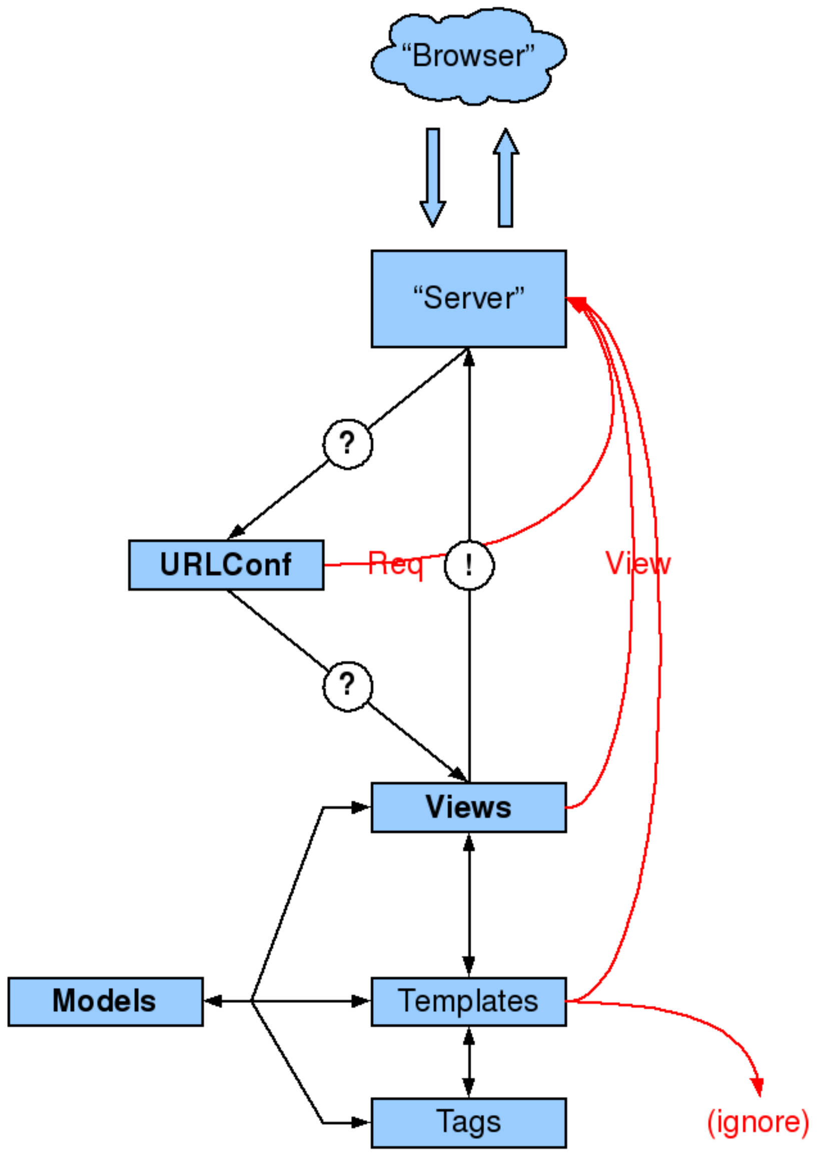
- Can use both doctest and unittest
- ./manage test   ./manage test aceu
- Creates a test database, populates it as needed, then discovers and runs your tests for you
- Django provides a test web client to make writing view tests easier
- This client can get at template details,context variables etc

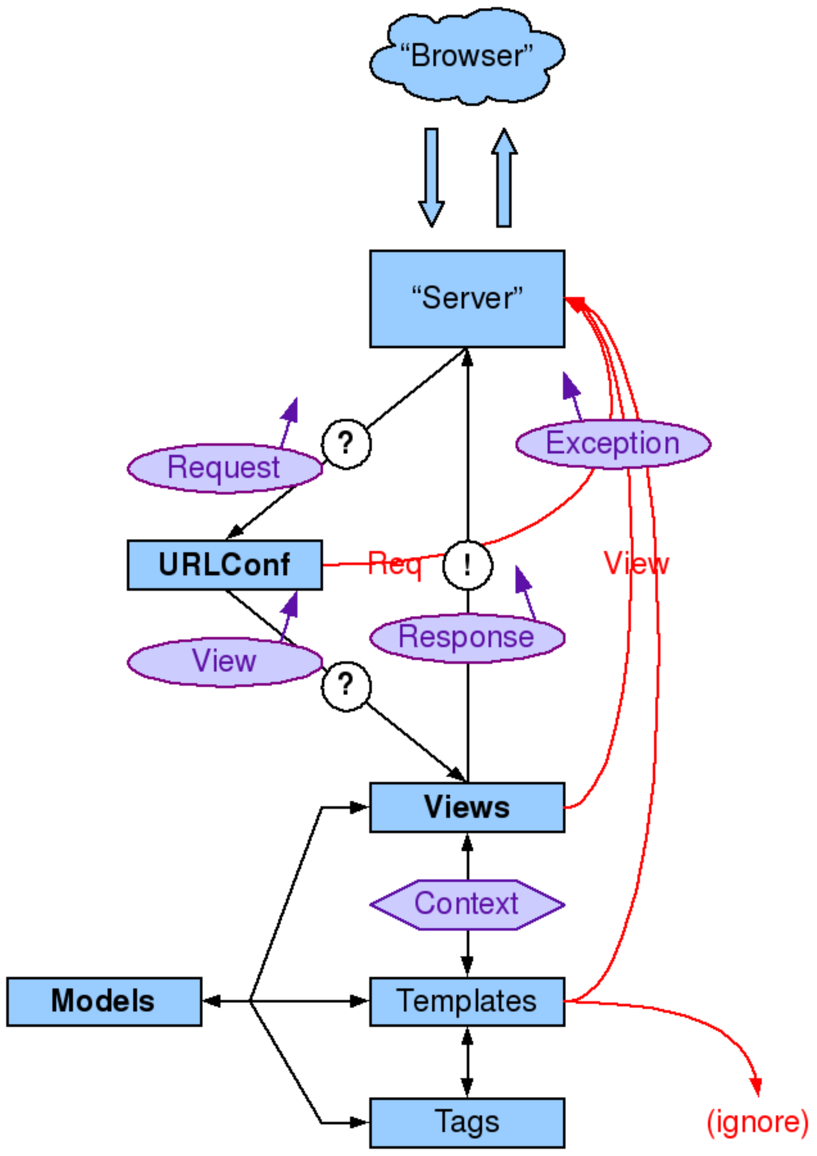
Using other people's code

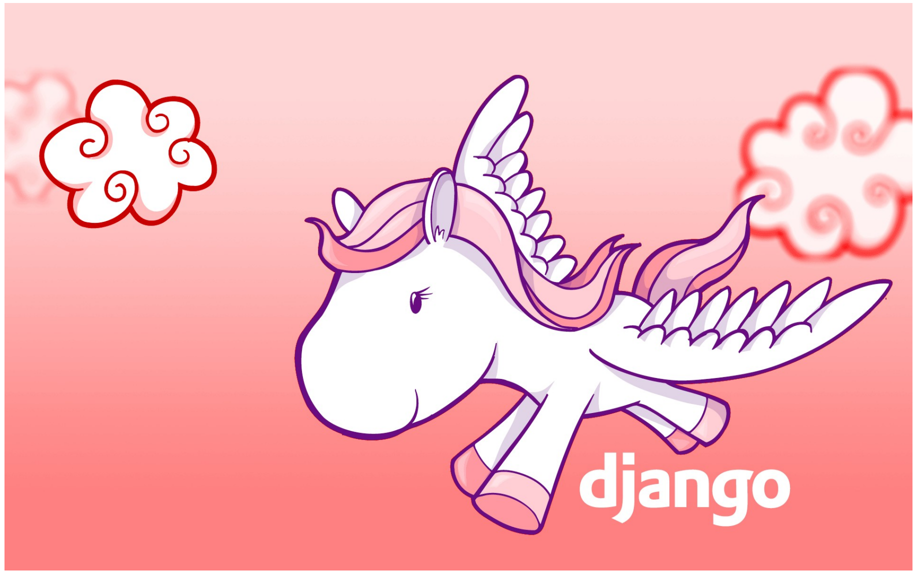
- You can easily drop in other applications to your project to do things
- Already made heavy use of many standard and contrib django apps
- Pinax - http://pinaxproject.com/ provides lots of handy apps for openid, gravatar, twitter etc
- You can easily find other open source django applications out there, and drop them into your project and use them


django


"Browser"
"Server"
?
URLConf
!
?
Views
Models
Templates
Tags


"Browser"
"Server"
?
URLConf
Req
!
View
?
Views
Models
Templates
Tags
(ignore)


"Browser"
"Server"
Request
?
Exception
URLConf
Req
!
View
Response
View
?
Views
Context
Models
Templates
Tags
(ignore)


django

Deploying with mod_python

- Works quite nicely
- Can make your apache threads quite heavy weight, consider not having too many, maybe have two httpd instances

```
<Location "/">
   SetHandler mod_python
   PythonPath "['/home/nick', '/home/nick/aceu'] + sys.path"
   PythonInterpreter aceu
   SetEnv DJANGO_SETTINGS_MODULE aceu.settings
   PythonHandler django.core.handlers.modpython
   #PythonDebug On
</Location>
Alias /admin-media/ /usr/lib/python2.4/site-
packages/django/contrib/admin/media/
Alias /media/ /home/nick/aceu/static/media/
```

Deploying as a war

- Using jython, you can compile your django app into a war, and deploy it however you'd like
- Grab jython 2.5, and django-jython
- Test with debugging webserver
- Add “doj” to your installed apps
- jython2.5 manage.py war --include-java-libs=../postgres-8.3-603.jdbc4.jar

ApacheCon

Questions?