

# The server side of Ajax: Django

## It's server-side time

- Almost always, you'll need server-side data
- Probably needs to be dynamic
- Hence, server-side code

## What does the server-side code do?

- Depends on your application
- Doesn't really matter
- But the **type of output** matters

## Output types

- Text/HTML
- JSON
- XML

## Text/HTML

- Just like any other dynamic Web page
- It's just smaller
- Let's back up a bit...

## Dynamic Web pages, the Django way

- Other approaches:
  - Filesystem
  - PHP / CGI
  - Calling method name from URL
- The Django approach
  - URLs decoupled from code

# A Django URLconf

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    ('^$', views.homepage),
    ('^/about/$', views.about),
    ('^/events/$', views.cal_archive),
    ('^/events/(200\d)/$', views.cal_year),
    ('^/events/(200\d)/(\d\d)/$', views.cal_month),
)
```

## URLconf design decisions

- URLs should be pretty
- Easy table of contents
- Decoupled from code
- Nestable
  - `( '^/foo/$', include('mysite.otherurls') ),`
- Allow for plugged-in apps

## How requests get processed

- Django finds matching URL
- Calls associated function
- Function is responsible for returning a response or raising an exception

## Example: A request to `/events/2006/`

```
urlpatterns = patterns('',
    ('^$', views.homepage),
    ('^/about/$', views.about),
    ('^/events/$', views.cal_archive),
    ('^/events/(200\d)/$', views.cal_year),
    ('^/events/(200\d)/(\d\d)/$', views.cal_month),
)
```

In `views.py`:

```
def cal_year(request, year):
    # This is the cal_year view.
    # ...
```

## A simple Django view

```
from django.http import HttpResponse

def cal_year(request, year):
    return HttpResponse('Hello world.')
```

Do whatever you want!

# A longer Django view

```
from django.http import HttpResponse
from mysite.models import Event

def cal_year(request, year):
    events =
    Event.objects.filter(day__year=year)
    html = ['<p>%s</p>' % e.name for e in
    events]
    html = ''.join(html)
    return HttpResponse(html)
```

# A longer Django view

```
from django.http import Http404, HttpResponse
from mysite.models import Event

def cal_year(request, year):
    events =
    Event.objects.filter(day__year=year)
    if not events:
        raise Http404
    html = ['<p>%s</p>' % e.name for e in
events]
    html = ''.join(html)
    return HttpResponse(html)
```

## Django models

```
from django.db import models

class Event(models.Model):
    name = models.CharField(maxlength=100)
    day = models.DateField()
    description = models.TextField()
```

# Django models: Adding methods

```
from django.db import models

class Event(models.Model):
    name = models.CharField(maxlength=100)
    day = models.DateField()
    description = models.TextField()

    def __str__(self):
        return self.name

    def is_today(self):
        import datetime
        return self.day == datetime.date.today()
```

## Model design decisions

- Explicit definition in code, no runtime introspection
  - Performance
  - Rich metadata
  - Database agnostic
- No assumptions based on field names – no black magic!

Once you've written the models...

- Generate the CREATE TABLE statements
- API for manipulating the data
- Utility can generate models from legacy DB

# Django database API: Basics

```
class Event(models.Model):
    name = models.CharField(maxlength=100)
    day = models.DateField()
    description = models.TextField()

>>> e = Event(name='Test',
               day=datetime.date.today(),
               description='Something happens.')
```

```
>>> e.save()
>>> e.name
'Test'
>>> e.delete()
```

# Django database API: QuerySets

```
class Event(models.Model):  
    name = models.CharField(maxlength=100)  
    day = models.DateField()  
    description = models.TextField()
```

```
>>> qs = Event.objects.all()
```

```
>>> print qs
```

```
[<Event: Foo>, <Event: Bar>]
```

```
>>> Event.objects.filter(name='Foo')
```

```
[<Event: Foo>]
```

```
>>> Event.objects.filter(name__contains='oo')
```

```
[<Event: Foo>]
```

## Django database API: QuerySets

```
Event.objects.filter(day__year=2006)
```

```
Event.objects.filter(day__month=11)
```

```
City.objects.filter(state__abbr__exact='MA')
```

```
Event.objects.count()
```

```
Event.objects.some_custom_method()
```

```
Event.future_objects.filter(...)
```

# QuerySets are lazy

```
qs = Event.objects.all()  
qs = qs.filter(day__year=2006)  
qs = qs.filter(name='Foo')
```

This is the same thing:

```
qs = Event.objects.filter(day__year=2006,  
                           name='Foo')
```

Nice for constructing queries dynamically:

```
qs = Event.objects.filter(name='Foo')  
if check_year:  
    qs = qs.filter(day__year=2006)
```

## Back to the view...

```
from django.http import HttpResponseRedirect
from mysite.models import Event

def cal_year(request, year):
    events =
    Event.objects.filter(day__year=year)
    html = ['<p>%s</p>' % e.name for e in
    events]
    html = ''.join(html)
    return HttpResponseRedirect(html)
```

Just one more thing: Hard-coding HTML is bad

# Use a template system

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from mysite.models import Event

def cal_year(request, year):
    events =
    Event.objects.filter(day__year=year)
    return render_to_response('cal_year.html',
        {'events': events})
```

# The Django template language

```
<html>
<head>
<title>My calendar</title>
</head>
<body>
<ul>

  {% for event in events %}
  <li>{{ event.title }} -- {{ event.day }}</li>
  {% endfor %}

</ul>
</body>
</html>
```

# Template inheritance

```
<html><head><title>My site</title></head>
<body>
<h1>My site</h1>
<p>Welcome to my home page.</p>
</body></html>
```

```
<html><head><title>My site</title></head>
<body>
<h1>My site</h1>
<ul>
{% for event in events %}
<li>{{ event.title }} -- {{ event.day }}</li>
{% endfor %}
</ul>
</body></html>
```

# Template inheritance

Parent template:

```
<html><head><title>My site</title></head>
<body><h1>My site</h1>
{% block content %}{% endblock %}
</body></html>
```

Child template:

```
{% extends "base.html" %}
{% block content %}
{% for event in events %}
<li>{{ event.title }} -- {{ event.day }}</li>
{% endfor %}
{% endblock %}
```

# Template filters

```
<table>
{% for event in events %}
<tr>
<td>{{ event.title|upper }}</td>
<td>{{ event.day|date:"F j, Y" }}</td>
</tr>
{% endfor %}
</table>
```

## Template system design decisions

- Don't invent a programming language
- Designer-friendly
- Text-based, NOT XML-based
  - Overhead
  - Sadistic
  - Limited to XML
- Safety/security
- Extensibility

## Back to Ajax

- Now we have the building blocks
- Text/HTML is just another view

## Example 1: Text/HTML snippet

In the URLconf:

```
('^ajax/today/$', views.ajax_today)
```

The view function:

```
def ajax_today(request):  
    import datetime  
    today = datetime.date.today()  
    count = Event.objects.count(day=today)  
    return HttpResponse('Events: %s.' % count)
```

# Example 1: Text/HTML snippet

In the HTML:

```
<script type="text/javascript">
var gid = document.getElementById;
var callback = {
  success: function(o) {
    gid('msg').innerHTML = o.responseText; }
  failure: function(o) {
    gid('msg').innerHTML = 'Could not retrieve.'; }
}
YAHOO.util.Connect.asyncRequest('GET',
  'http://example.com/ajax/today/', callback);
</script>

<div id="msg"></div>
```

## What about JSON?

- Say we want to return event data via JSON
- Dumb way, smart way

# JSON with Django: The “dumb” way

In the URLconf:

```
('^json/(\d\d\d\d-\d\d-\d\d)/$', views.json_day)
```

The view function:

```
def json_day(request, day):  
    events = Event.objects.filter(day=day)  
    json = ['{"name": "%s", "day": "%s",  
"description": "%s"}' % (e.name.replace('"',  
'\\"'), e.day.replace('"', '\\"'),  
e.description.replace('"', '\\"')) for e in  
events]  
    return HttpResponse('[%s]' % ','.join(json))
```

## JSON with Django: The “smart” way

- Utilities for converting database objects to JSON
- Serialization framework

## Example 2: SimpleJSON

```
>>> from django.utils import simplejson
>>> data = {'name': 'adrian'}
>>> simplejson.dumps(data)
'{"name": "adrian"}'
>>> data = [{'name': 'adrian'}, {'name':
            'alex'}, {'name': 'simon'}]
>>> simplejson.dumps(data)
'[{"name": "adrian"}, {"name": "alex"}, {"name":
 "simon"}]'
```

## Example 2: SimpleJSON

```
from django.utils import simplejson

def json_day(request, day):
    events = Event.objects.filter(day=day)
    json = simplejson.dumps(events)
    return HttpResponse(json,
                        mimetype='application/json')
```

```
TypeError: <Event: Foo> is not JSON serializable
```

## Example 3: Django JSON serialization

In the URLconf:

```
('^json/(\d\d\d\d-\d\d-\d\d)/$', views.json_day)
```

The view function:

```
from django.core.serializers import serialize

def json_day(request, day):
    events = Event.objects.filter(day=day)
    json = serialize('json', events)
    return HttpResponse(json,
                        mimetype='application/json')
```

# What does that look like?

```
[
  {
    "pk": "1",
    "model": "myapp.event",
    "fields": {
      "name": "Foo",
      "day": "2006-11-05",
      "description": "Lorum ipsum."
    }
  },
  {
    "pk": "2",
    "model": "myapp.event",
    "fields": {
      "name": "Bar",
      "day": "2006-11-06",
      "description": "Some description."
    }
  }
]
```

## Example 4: XML

In the URLconf:

```
('^xml/(\d\d\d\d-\d\d-\d\d)/$', views.xml_day)
```

The view function:

```
from django.core.serializers import serialize

def xml_day(request, day):
    events = Event.objects.filter(day=day)
    xml = serialize('xml', events)
    return HttpResponse(xml,
                        mimetype='application/xml')
```

# What does that look like?

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
  <object pk="1" model="myapp.event">
    <field type="CharField" name="name">Foo</field>
    <field type="DateField" name="day">2006-11-05</field>
    <field type="TextField" name="description">Lorum
    ipsum.</field>
  </object>
  <object pk="2" model="myapp.event">
    <field type="CharField" name="name">Bar</field>
    <field type="DateField" name="day">2006-11-06</field>
    <field type="TextField" name="description">Some
    description.</field>
  </object>
</django-objects>
```

# Factor out the commonalities

Old URLconf:

```
('^json/(\d\d\d\d-\d\d-\d\d)/$',  
 views.json_day),  
( '^xml/(\d\d\d\d-\d\d-\d\d)/$', views.xml_day)
```

New URLconf:

```
('^(json|xml)/(\d\d\d\d-\d\d-\d\d)/$',  
 views.api)
```



# Abstract further

URLconf:

```
('^(json|xml)/(\d\d\d\d-\d\d-\d\d)/$',  
views.api, {'model': event})
```

View:

```
from django.core.serializers import serialize  
  
def api(request, output, day, model):  
    events = model.objects.filter(day=day)  
    value = serialize(output, events)  
    return HttpResponse(value,  
        mimetype='application/%s' % output)
```

# Generic views

URLconf:

```
from django.views.generic import list_detail

# ...

('^/something/$', list_detail.object_list,
    {'queryset': Event.objects.all()})
```

No view code necessary!

## Avoiding Ajax server overload

- Lots of little, frequent requests add up
- If data doesn't change often, cache on server side

## Django cache system

- In config, choose cache type
  - Memcached
  - Database
  - Filesystem
  - Local memory
- Cache a particular view, cache a whole site, or use the low-level API

# The low-level cache API

```
>>> from django.core.cache import cache
>>> cache.set('my_key', 'hello', 30)
>>> cache.get('my_key')
'hello'
>>> cache.get('some_other_key')
None
```

```
# Wait 30 seconds for my_key to expire...
>>> cache.get('my_key')
None
```

```
>>> cache.delete('foo')
```

# The low-level cache API in a view

```
from django.core.cache import cache

CACHE_KEY = 'todays-event'
CACHE_TIME = 3600 # 1 hour

def ajax_today(request):
    count = cache.get(CACHE_KEY)
    if not count:
        import datetime
        today = datetime.date.today()
        count = Event.objects.count(day=today)
        cache.set(CACHE_KEY, count, CACHE_TIME)
    return HttpResponse('Events: %s.' % count)
```

# The per-view cache decorator

```
from django.views.decorators.cache import
    cache_page

def ajax_today(request):
    import datetime
    today = datetime.date.today()
    count = Event.objects.count(day=today)
    return HttpResponse('Events: %s.' % count)
ajax_today = cache_page(ajax_today, 60 * 60)
```

Automatically picks a cache key

## Auto-generating Ajax?

- Some server-side frameworks generate JavaScript automatically
- Django intentionally doesn't
  - Don't want to limit choices
  - Code generation can be evil
  - It's like generating a Web design
  - “It's like using a motorized wheelchair because you're too lazy to walk.”
  - <http://www.b-list.org/weblog/2006/07/02/django-and-ajax>

But if you really wanted to...

```
{% link_to_remote '/myurl/' mydiv msg='Loading' %}
```

```
<div id="mydiv"></div>
```

Thanks!

Adrian Holovaty

[holovaty@gmail.com](mailto:holovaty@gmail.com)

<http://www.holovaty.com/>