From 0 to Website in 60 Minutes - with Django

Sean Boisen, <[my first name]@seanboisen.com>

LinuxFest Northwest 2010, April 24

Presentation URL:

http://semanticbible.org/other/talks/2010/linuxfestnw/Django

Outline

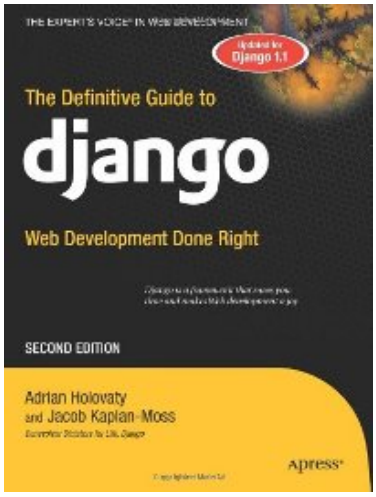
- Background: Django, goals, prerequisites
- The Food and Farm Finder (F3) Task
- Building the app
  - Views
  - Data and models
  - Templates
- Wrap-up

Goals

- Introduce the basics of Django, focusing on ...
- Interactively building a database-backed web site (during this talk!)
  - Handle a variety of HTTP requests in a general fashion
  - Create a data model and database implementation
  - Create alternate views of that data
  - Create a collection of HTML pages and responses
- Can I do all this in ~~60~~ 45 minutes?!?

Why Django?

- Django: the Web framework for perfectionists with deadlines
- Builds on Python:
  - Clean, elegant, object-oriented language
  - Dynamic typing and interactive evaluation are well suited to rapid prototype development
  - Rich set of library modules and an active open-source community



- Born in the high-pressure world of on-line publications
- Well-suited to database-backed tasks
- Popular, well-supported
- Automate the drudgery and focus on the fun!

Prerequisites

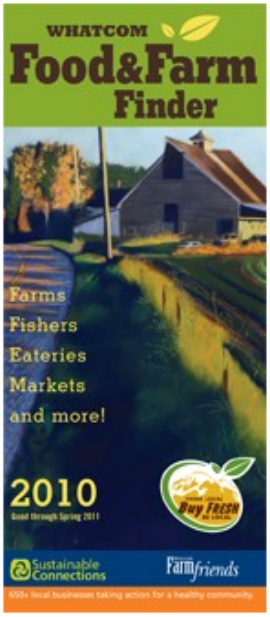
- 0 Django knowledge
- Python: basic syntax, classes and methods, regexps
- Basic understanding of web architecture and development
- Basic understanding of relational data models
- A little HTML
- Optional: track shoes

Shortcuts: What I'm Leaving Out

- Python installation
  - Download from python.org or ActiveState
  - I'll be using Python 2.5 today: 2.4 or 2.6 should work the same. Django is not yet compatible with Python 3.
- Django installation
  - Current official release is Django-1.1.1.tar.gz
  - You can also check for a third-party distribution
  - Check the installation guide for details
- Database installation
  - Python 2.5+ comes with SQLite, so no extra installation is required
  - I'm using MySQL today: this also requires the MySQLdb database connector. It can be hard to find Windows binaries for this.
- Database configuration for Django
  - Creating a new project: `django-admin.py startproject` *`mysite`*
  - Edit variables in `settings.py` for db engine, name, username, password
  - `python manage.py syncdb`
- Data: i've already modeled and organized it
- Hosting issues and deployment on an external web server
- These are all documented but tedious: let's focus on the fun parts!

The Application: F3

- Whatcom Food and Farm Finder from Sustainable Connections
  - Several dozen local farms, with detailed descriptions, contact information
  - A wide variety of foods and other products, with information about seasons
  - But ... Dead Tree Deficiencies!
- Wouldn't it be great if you could easily
  - *Search* this information
  - Plot the locations on a Google Map
  - Mash this data up with other resources
  - Support eating healthy and buying local



The Application: F3 (2)

- What it would take to make a Django app out of this?
  - Capture the data in a structured fashion
  - Create useful models and views of them
  - Put it up on the web for everybody
- **Disclaimer**: this is a rogue data operation!
  - No endorsement or approval from Sustainable Connections is expressed or implied.

The Source Data: Farms

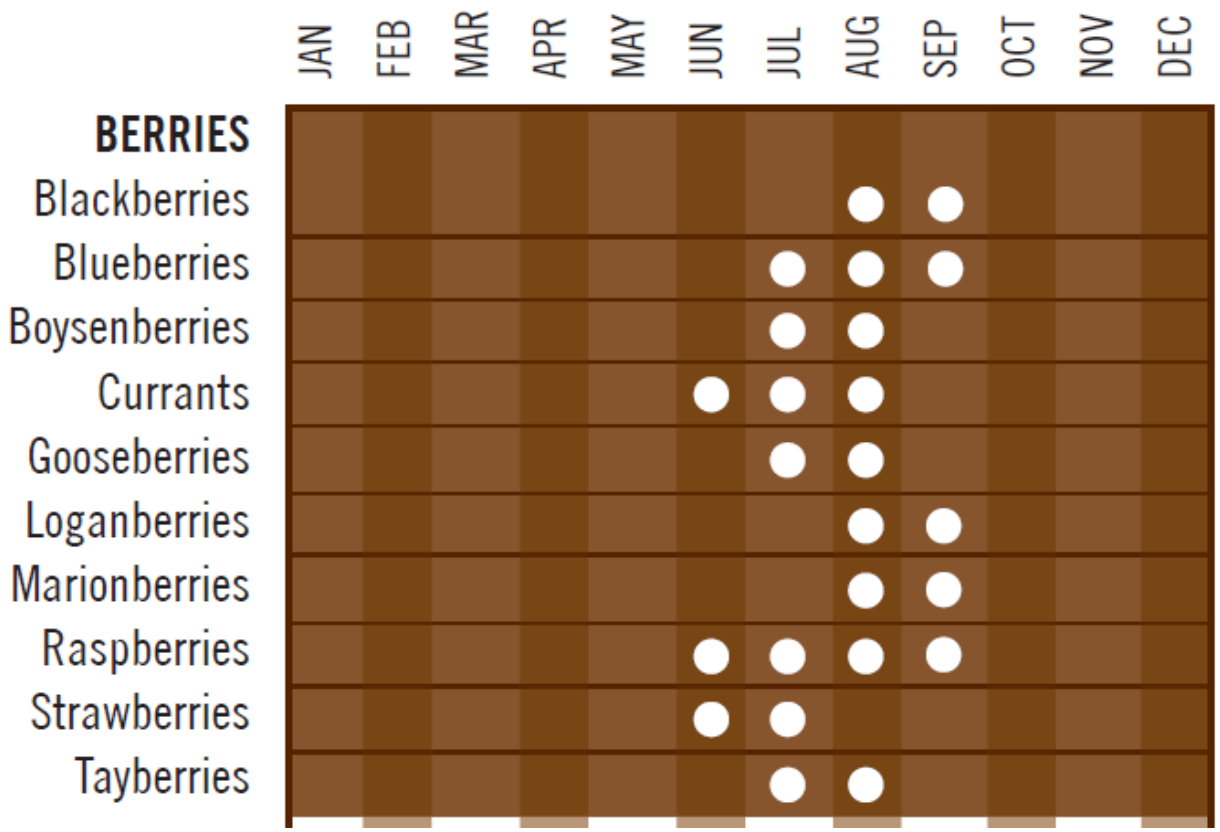
26 Boxx Berry Farm | Mike & Roger Boxx

6211 Northwest Rd, Ferndale | (360) 380-2699
www.boxxberryfarm.com

Farm Stand; U-Pick

Our Farm Market offers fresh strawberries, raspberries, blueberries, flowers, sweet corn and other homegrown vegetables, Eastern WA fruit and produce, pumpkins, as well as our own jams, syrups, homemade berry pies, and frozen berries. Get fresh strawberry shortcake, strawberry sundaes and ice cream cones at the Shortcake Shack. U-pick strawberries, raspberries, blueberries, pumpkins, and flowers. Open June-Oct, Mon-Sat 9-6, Sun 10-4 and weekends in December 10-4pm.

The Source Data: Berries



Data Elements of F3

- 70 instances of Food: name, type, month that it's available
  - Month: name, integer
  - FoodType (berries, plants, vegetables, etc.): name
- 64 instances of Farm: name, contact, address, etc.
  - ServiceOffering (farm stand, u-pick, etc.): name, description
  - SustainabilityIndicator (green power, zero waste, organic, etc.): name, description
  - PaymentMethod: name

Building the App: `startapp`

- A Django app is

  "a portable set of Django functionality, usually including models and views, that lives together in a single Python package" (p. 76)

- Create a new application skeleton

```
python manage.py startapp f3_0
```

- This makes a subdirectory and "placeholder" files: `views.py`, `models.py`, and `tests.py`

Building the App: Our First View

- We'll add a simple view to `f3_0.views.py`

```
from django.http import HttpResponse
def pingfn(request):
    return HttpResponse('Hello LinuxFest!')
```

- Then wire a URL to it in the project-level `urls.py`

```
import f3_0.views
urlpatterns = patterns('',
        (r'^ping/', f3_0.views.pingfn),
        )
```

- This will dispatch the URL `(appsite)/ping` to call `pingfn()`, supplying the HTTP request as an argument.

Building the App: Testing Our First View

- Start up Django's development server in a console

```
> python manage.py runserver
```

- Then visit `http://127.0.0.1:8000/ping` in a browser, and ...
- Wonder and be amazed!

Review of our First App

- We now have a complete initial system that can:
  - Receive a URL
  - Dispatch to a function
  - Generate a response
- We can test it interactively on our development server
- Pretty *un*impressive, but this demonstrates the basic HTTP interface

The Database Layer

- Django is designed for *database-driven* web sites
- The data layer includes Python-based object models, and a database API
- Model-View-Controller, Model-Template-View ... whatever
- Lots of acceleration thanks to Django models:
  - You get to work in Python instead of SQL
  - Django can generate table definitions, manage indexes, provide auto-incremented primary keys, and more
  - Your app can be database-agnostic

Defining Models: FoodType, Month

```
import django.db.models as m

class FoodType(m.Model):
    name = m.CharField(max_length=30, unique=True)

    def __unicode__(self):
        return self.name

class Month(m.Model):
    name = m.CharField(max_length=10, unique=True,
                       help_text="the month")
    order = m.IntegerField(unique=True)

    def __unicode__(self):
        return self.name
```

Defining Models: Food

```
class Food(m.Model):
    name = m.CharField(max_length=50, unique=True,
                       help_text="the name of the food")
    type = m.ForeignKey('FoodType')
    months = m.ManyToManyField('Month')

    def __unicode__(self):
        return self.name
```

- Many `Food` instances can have the same `type`
- The relationship between `Food` instances and `Month` instances is many-to-many

Defining Models: Farm (Abbreviated)

```
class Farm(m.Model):
    name = m.CharField(max_length=100)
    contact_person = m.CharField(max_length=100)
    address = m.TextField()
    phone_number = m.CharField(max_length=30)
    website = m.URLField()
    email_address = m.EmailField()
    description = m.TextField()
    hours_of_operation = m.CharField(max_length=100)
    seasonal_operation = m.CharField(max_length=100)
    service_offerings = m.ManyToManyField('ServiceOffering')
    payment_methods = m.ManyToManyField('PaymentMethod')
    sustainability_indicators = m.ManyToManyField('SustainabilityIndicator')
    categories = m.ManyToManyField('FarmCategory')
    confirmed_foods = m.ManyToManyField('Food')
```

Loading Data

```
import django.core.serializers as ser

for obj in ser.deserialize("json", open('f3_final.json', 'r')):
    obj.save()
```

- Django has built-in serialization and deserialization
- Our source data is formatted as JSON

A Snippet of Data in JSON


```
{ "pk": 49,
  "model": "f3_final.farm",
  "fields": {
    "phone_number": "360.380.2699",
    "website": "http://www.boxxberryfarm.com",
    "name": "Boxx Berry Farm",
    "confirmed_foods": [9, 8, 2, 13, 46, 60],
    "service_offerings": [1, 5, 6],
    "hours_of_operation": "",
    "seasonal_operation": "",
    "payment_methods": [],
    "address": "6211 Northwest Road\nFerndale",
    "sustainability_indicators": [1],
    "contact_person": "Mike & Roger Boxx",
    "email_address": "",
    "categories": [],
    "description": ""
  }},
```

Database API: Simple Query

```
>>> from f3_final.models import *
>>> bbf = Farm.objects.get(id=49)
>>> bbf
<Farm: Boxx Berry Farm>
>>> bbf.website
u'http://www.boxxberryfarm.com'
>>> FoodType.objects.all()
[<FoodType: Berries>, <FoodType: Dairy Products>,
<FoodType: Eggs> ... ]
```

- Basic equivalents of SQL SELECT

Database API: Other Queries

```
>>> Month.objects.filter(name__contains='r')
[<Month: January>, <Month: February>,
<Month: March>, <Month: April>, <Month: September>,
<Month: October>, <Month: November>, <Month: December>]
>>> Food.objects.filter(type__name='Berries')
[<Food: Blackberries>, <Food: Blueberries>,
<Food: Boysenberries> ... ]
```

- You can chain through data relationships
- Filters can be pipelined

More Data Layer Functionality

- Create and save new records
- Modify values
- Delete records
- `QuerySets` provide lazy evaluation and caching.
- Manager methods provide table-level functionality

Template Basics

- Django separates views from the HTML that displays them:
  - You should be able to change page design without touching code
  - Different people have different gifts
  - Each can be developed separately
- The template system provides basic HTML plus variables, control tags, and filters
- You don't *have* to use it
- You can generate more than just HTML

My First Template

```
<html>
<head><title>My First Template</title></head>
<body>
    <h1></h1>
        {% if object_list %}
        <table>
          <thead><tr><th>ID</th><th>Name</th></tr></thead>
          <tbody>
                {% for object in object_list %}
                <tr>
                  <td>{{ object.id }}</td>
                  <td>{{ object.name }}</td>
                </tr>
                {% endfor %}
          </tbody>
        </table>
        {% endif %}
</body>
</html>
```

Rendering Views with Templates

- Retrieve data
- Define the context information the template will need:
  - Contexts let you further isolate template design from data details
- Hand off to a template to render it
  - The template system uses `.` to access dictionary keys, attributes, methods (without argument), and list indices
- Non-existent variables are rendered as empty strings and fail silently

Putting It All Together: `f3_final`

- Tell the project-level `urls.py` to dispatch to app-specific URLs

```
import f3_0.views
urlpatterns = patterns('',
        (r'^f3/', include('lfnw.f3_final.urls')))
```

- Define a pattern in `f3_final/urls.py` to show a list of farms

```
import views
urlpatterns = patterns('',
        (r'^farms/$', views.farms_list))
```

Putting It All Together: `f3_final` (2)

- Write the view

```
from django.shortcuts import render_to_response
import f3_final.models as models
def farms_list(request):
    return render_to_response('myfirsttemplate.html',
                {'object_list': models.Farm.objects.all(),})
```

(`render_to_response` instantiates a template object with the given context and generates an HTTP response)

- Try it out: http://127.0.0.1:8000/f3/farms

Admin Interface

- More batteries included: a powerful admin interface for free!
- Includes display customization, entry validation, drop-downs, filtering, ...
- To enable it:
    - Add it to your `INSTALLED_APPS`
    - Run `python manage.py syncdb`
    - Uncomment the relevant lines in `urls.py`
- To use it:
    - Add an `admin.py` file to the app
    - Register models you want an admin interface for

Review: The Django Development Cycle

- Define the resources your app will expose
- Model those resources in the data layer
- Design the URLs for those resources
- Define views that query the models and provide operational logic
- Design templates to display the views
- Lather, rinse, repeat, have fun ...

What *Else* Can Django Do?

- Errors pages for debugging
- 404 handling, custom 404 responses
- Template inclusion to provide standard styles
- ... and much, much more

Best Practices for Speedy Django Development

- Leverage the power of interactive, incremental development
- Remember it's all "just Python"
- Keep things decoupled
- Cut *with* the grain, not against it:
  - Let Django manage the database interactions
  - Use the template system
- The Django community is very helpful: use it!

And Now, a Word from our Sponsors ...

- 

- My website: http://www.SemanticBible.com/
- My blog: http://www.SemanticBible.com/Blogos/
- Slidy, a standards- and browser-based XHTML presentation framework
- F3 data and models courtesy of the Whatcom Python Users Group
- And of course ... 

Resources

- Bitbucket repository for Django code and data
- If you like Django, check out James Tauber's Pinax: layers social networking and other features on top of Django