Building Web Applications with Python

A whistlestop tour

Joseph Redfern

November 29, 2017

Web Frameworks

What is a framework?

Web Frameworks provide implementations of common web application functionality. They can make development easier, faster, and more secure – especially when working on large projects with multiple developers.

PHP

- Laravel
- Symphony
- CakePHP
- ...

Python

- Django
- Flask
- CherryPy
- ...

Java

- Spring MVC
- Struts
- Spring Boot
- ...

Typical Framework Functionality

Functionality varies from framework to framework, but a typical features include:

- URL Routing
  - Mapping the path in a URL to a page/specific functionality
- Template Rendering
  - Easy separation of application data/logic from HTML
- Unit test helpers
  - Built-in support for unit tests, with helper functions and test runners

It is common for frameworks to have plugins that extend their functionality.

Why use a framework?

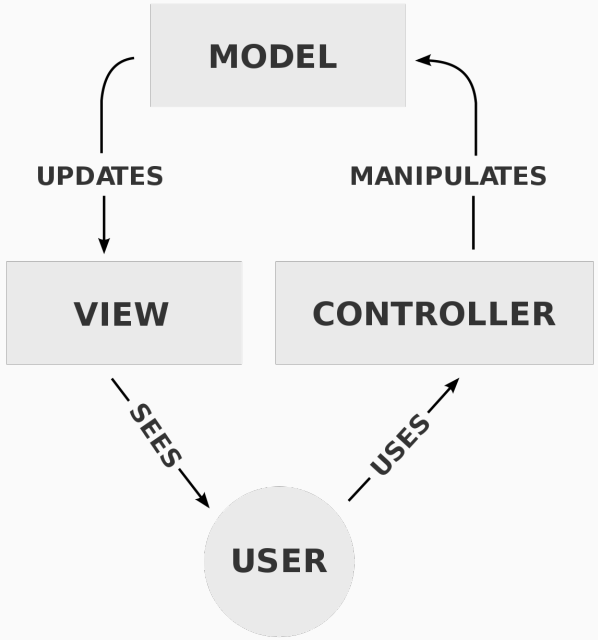
Framework-less development **can** get very messy.

```php
<?php
if(isset($_GET['id'])){
    $id = $conn->real_escape_string($_GET['id']);
    $res = $conn->query('SELECT * FROM people WHERE id=$id');
    if($res->num_rows > 0){
        $row = $res->fetch_assoc();
        echo 'Hello '.$row['name'].', how are you?';
    }
}else{
?>
<form method='get'>
Person ID: <input type='text' name='id'></input>
</form>
<?php
}
?>
```

Why use a framework?

There are a number of reasons to choose to use a web application framework:

- Decouple database, application logic and presentation
- Avoid re-inventing wheel – tested, more-secure and better documented
- Community
  - People have probably already solved your problem!
  - Plugins
  - Tutorials



Django

What is Django

Django is a Web Application framework for Python. It is a very complete framework and has an extensive feature set.

Features:

- Built in ORM (more on that later...)
- User Management w/ permissions system
- Automatic form generation/validation
- Admin Interface

Compared to other Python web frameworks (i.e. Flask, CherryPy) it has a relatively steep learning curve and **may** be overkill for some needs, but once the basics are mastered it can be extremely powerful.

Getting Started

First thing that needs to be done is to install Django:

```
pip install django
```

Having installed Django, we now have a program called `django-admin.py` on our path. We can use this tool to create a new Django project:

```
django-admin.py startproject MinceMaster3000
```

The `startproject` command will create a new folder (in this instance named `MinceMaster3000`) containing the bare-bones structure and files for a Django project.

Creating an App

Django projects consist of one or more apps. Apps are self-contained units of functionality. Our mince-pie rating system will consist of only one app, which we will call `pierate`.

We can then create an app using the new `manage.py` script created for us when we started the project:

```
python manage.py startapp pierate
```

Like the `startproject` command, `startapp` will create the boilerplate needed for a Django app.

Project Structure

```
~> tree
├── manage.py
├── MinceMaster3000
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── pierate
    ├── admin.py
    ├── apps.py
    ├── __init__.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
```

Models

Django comes with its own ORM (Object-relational mapping) functionality. An ORM marshalls between rows in a database and objects in your programming language.

This means that we can query the database using Python alone. To do this, we need to define “models”, which are classes that get mapped to/from the database by Django.

To do this, we extend a class called `Model`. Django will pick up on these classes, and automatically tables that represent those models for us.

models.py

```
from django.db import models
from django.db.models import Avg
from django.contrib.auth.models import User
from django.core.validators import MaxValueValidator

class Pie(models.Model):
    name = models.CharField(max_length=255)
    date_added = models.DateTimeField(auto_now_add=True)
    price = models.DecimalField(max_digits=4, decimal_places=2)

    def get_average_rating(self):
        ratings =  Rating.objects.filter(pie=self)
        return ratings.aggregate(avg=Avg('pastry') + Avg('filling'))['avg

class Rating(models.Model):
    date_added = models.DateTimeField(auto_now_add=True)
    creator = models.ForeignKey(User)
    pie = models.ForeignKey(Pie)
    pastry = models.DecimalField(validators=[MaxValueValidator(5)],
                                 max_digits=4, decimal_places=2)
    filling = models.DecimalField(validators=[MaxValueValidator(5)],
                                 max_digits=4, decimal_places=2)
    comments = models.TextField()
```

Forms

Django makes creating and validating forms easy.

By extending the `ModelForm` class, you can easily automatically generate and validate forms based on a model you have already defined:

```
from django.forms import ModelForm
from .models import Pie, Rating

class RateForm(ModelForm):
    class Meta:
        model = Rating
        fields = ['pie', 'pastry', 'filling', 'comments']
```

This code defines a form that is bound to the Rating model, and exposes the pie, pastry, filling and comments fields.

Views

In Django, views are responsible for processing requests, and returning responses.

Our system will views for:

- Listing all mince pies
- Viewing details about a specific mince pie
- Rating a pie

The easiest way to write a Django view is by defining a function. The first argument of the function will be the request that was sent to Django – further arguments are possible, but not required.

views.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from .models import Pie
from .forms import RateForm

def details(request, pie_id):
    pie = Pie.objects.filter(id=pie_id).first()
    return render(request, 'details.html', {'pie': pie})

def list_all(request):
    pies = Pie.objects.all()
    return render(request, 'list_all.html', {'pies': pies})

def rate(request):
    if request.method == 'POST':
        form = RateForm(request.POST)

        if form.is_valid():
            rating = form.save(commit=False)
            rating.creator = request.user
            rating.save()
            return HttpResponseRedirect('/p/{}'.format(rating.pie.id))
    else:
        form = RateForm()
        return render(request, 'rate.html', {'form': form})
```

rate.html

```
<h1>Rate a Pie</h1>

<p>Fill in the form below to rate a pie.</p>

<form method="POST">
    {% csrf_token %}
    <table>
    {{ form }}
    </table>
<input type="submit"/ >
</form>

<a href="/">Back</a>
```

list_all.html

```
<h1>MinceMaster 3000</h1>

<p><a href='/rate'>Click here to add a rating</a></p>

<table>
    <tr>
        <th>Pie Name</th>
        <th>Pie Price</th>
        <th>Average Rating</th>
    </tr>
{% for pie in pies %}
    <tr>
        <td><a href="/p/{{pie.id}}">{{pie.name}}</a></td>
        <td>£{{pie.price}}</td>
        <td>{{pie.get_average_rating|floatformat}}</td>
    </tr>
{% endfor %}

</table>
```

details.html

```
<h1>{{pie.name}}</h1>

<b>Price:</b>£{{pie.price}}<br>
<b>Added:</b>{{pie.date_added}}<br>
<b>Average Rating: </b>{{pie.get_average_rating}}<br>
<hr/>

{% for rating in pie.rating_set.all %}
<h3>{{rating.creator}}</h3>
<i>Added {{rating.date_added}}</i><br>
<b>Filling Rating: </b> {{rating.filling}}/5 <br>
<b>Pastry Rating: </b> {{rating.pastry}}/5 <br>
<b>Comments:</b>
<p>{{rating.comments}}</p>
<hr/>
{% endfor %}

<a href="/">Back</a>
```

Registering the App

There a few things that we have to do before we can run our application.

Firstly, we need to register our app with the Django project – this can be done by editing `settings.py` and adding `pierate` to the `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pierate'
]
```

URLs & Routes

At the moment, Django doesn't expose any views – there's no way of interacting with our application.

In order to fix this, we need to define some routes. Routes map from a URL to a Django View. For instance, `http://mince.pie/p/1` might return a page detailing Pie with ID 1.

To create these mappings, we need to edit `urls.py`:

```
    from pierate.views import list_all, rate, details

urlpatterns = [
    url(r'^$', list_all, name='list_all'),
    url(r'^p/(?P<pie_id>[0-9]+)/$', details, name='details'),
    url(r'^rate/$', rate, name='rate'),
]
```

Django uses regular expressions to match URLs.

Database & Users

We then need to instruct Django to create database tables from our model definitions. This can be done with:

```
python manage.py makemigrations
python manage.py migrate
```

Built-in suport for migrations means that you can update your models on a live-database without loosing any data – Django will handle the changes for you.

By default, Django uses a sqlite backend, which is useful for development purposes or smaller applications. It is easy to switch – this can be done by editing `settings.py`.

Enabling Django Admin

Django has a built-in admin interface that lets you create/read/update/delete instances of your model. All that needs to be done to enable it is to register your models.

This can be done by editing `admin.py`:

```
from django.contrib import admin
from .models import Pie, Rating

admin.site.register(Pie)
admin.site.register(Rating)
```

Adding Users

In order to add an initial user to the Django user management system, we can again fall back to our old friend `manage.py`:

```
python manage.py createsuperuser
```

You will be prompted for a username, email address and password. Django enforces a fairly sensible password policy by default – it cannot be a common word, and must be at least 8 characters long.

Superusers have full access to all site functionality, including Django Admin.

Running the Application

Django comes with a built-in web server that can be used for development/debugging purposes. To start your application, run:

```
python manage.py runserver
```

Django will then start a HTTP server on `localhost:8000`. By default, debugging is enabled – so if an exception occurs, a full stack trace will be printed. You can disable this by chanving the value of `DEBUG` in `settings.py`.

APIs with Django

Django can also be used to provide an API for consumption by a mobile application or client-side framework.

There's nothing to stop you from returning JSON directly from your views, but there are (potentially better) alternatives.

Django REST Framework a popular choice – will automagically create API endpoints for your models, helps with oAuth, and provides nice UI for interacting with your API during development.

Alternatives

In some instances, Django **may** not be the right tool for the job. If you don't need the ORM, a User Management system, forms generation etc then it might be worth considering other options.

Flask, Bottle and CherryPy are lighter-weight Python-based web frameworks. They are easier to get started with, are more "bare-bones" (leaving more down to the developer). Depending on the project (and your discipline as a developer) this could be a blessing or a curse.

Questions?

https://github.com/JosephRedfern/DjangoMincePieTalk