Cooking With Django

Chris Grubbs and Matt Bone

December 6, 2007

Contents

1 Introduction

phlombay is a collaborative cooking site running on Django, a Python-based web framework. It began its life as a semester project for Dr. Konstantin Laufer's server-side development class. The overarching goal of Dr. Laufer's assignment was to build a site that demonstrated a variety of user roles and content types, and would provide varying stages of functionality depending on the logged-in user's role. The idea of a cooking site seemed ideal because recipes can easily be expressed in terms of a database model, and would naturally lend themselves to a community-based system in which a variety of users interact.

Like most user-driven websites, phlombay employs a role-based authentication system. Anonymous users can freely browse the site's recipe repository, but they cannot make any changes. Registered users fall into one of two roles: line cooks and master chefs. New users start as line cooks, who are able to add their own recipes and modify them at any time. They can also subscribe to a recipe and receive email alerts whenever that recipe is modified.

Master chefs are able to edit any recipe on the site, regardless of its owner, and are also able to lend their "seal of approval" to a particularly good recipe, causing the recipe to show up in the featured box on the site's front page. Master chefs also get administrative privileges, allowing them to modify or delete user profiles, for example.

2 About Django

2.1 Overview

phlombay runs on the Django framework, which takes care of many of the obnoxious building blocks involved in creating a dynamic website. It provides an object-relational mapper, allowing us to express our underlying database model in terms of Python classes. It takes care of all the SQL and can handle a variety of database implementations. Additionally, Django provides a templating system, user authentication, a unit testing framework, and much more. We found development in Django to be a free-flowing and enjoyable process, albeit with some minor complications, which will be discussed later.

2.2 Templates

Django provides a fairly easy-to-use templating system. These templates hook into Django's view functions, so that we can pass variables (such as, say, whether a given user is able to edit a recipe or not) into the template renderer, which then renders the HTML (or PDF, XML, plaintext, etc.) accordingly. There is also a system of template inheritance, such that a base template can be created, and further page-specific templates can be built on top of that.

The template language also allows for Python-like code constructs, such as if statements and loops, which allowed us to further customize the template's interaction with the backend. To a certain extent, allowing code to exist in a template violates separation of concerns. Django leaves such architectural concerns to the developer - the framework itself allows but does not enforce separation of concerns.

Note that a consequence of allowing the renderer to produce formats other than HTML is that it does not provide any guarantee that what is emitted is valid XHTML.

2.3 Model-View-Controller in Django

The basic flow of execution in a Django app is this:

1. User requests an URL
2. Django refers to `urls.py`, which maps URLs to methods in `views.py`
3. The appropriate `views.py` method is called, which pulls data from the database and passes it to a template renderer

```
def detail(request, recipe_id):
    recipe = get_object_or_404(Recipe, pk=recipe_id)

    subscribed = False #user subscribed to recipe?
    can_edit = False
    if(request.user.is_authenticated()):
        subscribed = len(get_subscriptions(request.user, recipe))>0
        if str(request.user) == str(recipe.owner)
           or str(request.user.groups.all().get()) == 'masterchef':
            can_edit = True

    return render_to_response('recipes/recipe.html',
                              {'recipe':recipe,
                              'can_edit': can_edit,
                              'subscribed':subscribed},
                              context_instance=RequestContext(request))
```

Listing 1: Detail method in views.py.

4. Template renderer returns HTML to the user

It is worth noting that in Django, the `views.py` component is not analogous to the View in the Model-View-Controller sense. Rather, it is the aforementioned templating system that does the job of the View, while `views.py` can be likened to the Controller. Each method in `views.py` maps to an URL, and is responsible for a particular interaction between the view and the model.

For example, listing 1 shows the `detail()` method, which takes a recipe ID as a parameter (given in the URL in the form `/recipes/detail/n/`, where n is the ID.). It then grabs the recipe with that ID from the database (the Model) and checks to see if the user is subscribed and whether they can edit the recipe. Finally, it passes the template file, the recipe, and the other necessary information to the templating system (the View.) In this way, it acts as the Controller, mediating between the front and back ends.

2.4 Extras

2.4.1 Admin Interface

An automatically generated admin interface comes built in with Django. With a few minor modifications to our model classes, we can interact with our data model directly from this interface. We can add, modify, and delete instances of recipes and ingredients, as well as handle user administration. Obviously, this made implementing administrative permissions for master chefs much easier.

Incidentally, Django's admin interface would make it tremendously easy to create and administer a data driven site. After writing the data model classes, the developer would barely need to do any frontend work at all to get the site off the ground. Any content modification could be done directly from the admin interface.

2.4.2 Flatpages

Flatpages are simple database-driven pieces of "flat content" that don't fit into the object model elsewhere. Our "about", "help", and "contact" links are examples of flatpages. The content of these pages, rather than existing in a static file, lives in the database.

2.4.3 Markup

One of the problems with text fields in HTML forms is that things like linebreaks and ad-hoc lists do not translate well when they are rendered back into HTML. Also, forcing users to enter their own HTML

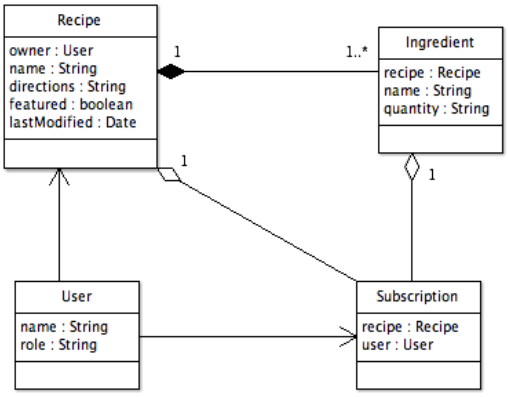

Figure 1: phlombay class diagram.

formatting is not only tedious but also insecure.

Django offers a markup module that allows users to enter textile, markdown or reStructured Text markup into text fields. Form submission is handled as usual, but the marked up text is sent through a filter prior to display. This filter converts the markup to the appropriate HTML. Currently, phlombay recipes make use of reStructured Text. reStructured Text is also used by the the Python docutils module for marking up inline documentation.

3 Object Model

3.1 Overview

The class diagram for phlombay is quite simple and is shown in figure 1. On the top left is the Recipe class (also see listing 2), which contains a variety of fields, including the owner's name, whether it is a featured recipe, and when it was last modified. Notably, it contains an aggregation of Ingredients (also see listing 3), which are classes with fields containing the ingredient's name, the quantity, and a reference to the parent Recipe.

On the bottom right is the Subscription class. This is a very simple class associating a Recipe with a subscribing User. The User class is derived from Django's authentication data model. It should be said that, in phlombay's actual implementation, the User class is derived from Django's authentication model and therefore contains additional fields and methods other than those shown. However, those aren't particularly relevant to a description of phlombay's basic data model.

3.2 Separating the model from Django

3.2.1 ActiveRecord and Django ORM

An Active Record is "an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data" [Fowler(2003), pg 160], and the Django approach to object-relational mapping uses this pattern. The first thing to notice about this pattern is that the classes are aware that they are being persisted in a database. That is, the logic that is responsible for persisting the instances of any class is found in the class itself. Business logic may also be mixed in with this persistence logic. At first this may sound messy, and for applications with complicated business logic, this is the case. However, these applications are not in the spirit of the Active Record pattern. Instead, Active Record is only for applications with little or no business logic; it is a pattern for applications mainly interested in create, read, update, and delete (CRUD) operations on the business objects.

```
class Recipe(models.Model):
    owner = models.ForeignKey(User)
    name = models.CharField(max_length=200)
    directions = models.TextField()
    featured = models.BooleanField(default=False)
    last_modified = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return self.name

    class Admin:
        pass
```

Listing 2: Recipe class.

```
class Ingredient(models.Model):
    recipe = models.ForeignKey(Recipe, edit_inline=models.TABULAR)
    quantity = models.CharField(max_length=100, core=True)
    name = models.CharField(max_length=200, core=True)

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

Listing 3: Ingredient class.

One of the interesting things to note is the way that object references are handled in this pattern. As shown in listing 3 the Ingredient class is aware of its foreign key relationship with a Recipe. The logic is somewhat backwards; in terms of an object tree, we would consider the Recipe to be the parent of an Ingredient, yet there is no mention of this in the Recipe class. While these one-to-many relationships are handled quite easily with foreign keys, many-to-many relationships are trickier. Remember, the Active Record pattern suggests a one-to-one correspondence between tables and classes, and we do not usually think of the join table as needing its own representation as a class. Yet our Subscription class (see listing 4) is just that. The subscription table, with two foreign keys, is nothing more than a join table, but because we are only interested in CRUD operations for Subscriptions, it is more natural to represent a subscription as an object rather than a many-to-many relationship on the User and Recipe classes.

3.2.2 Data Mapper

A Data Mapper "moves data between objects and a database while keeping them independent of each other"[Fowler(2003), pg 165], and Hibernate provides an implementation of this design pattern in Java. One of the interesting thing about Hibernate is that domain objects are not aware of any of the persistence code, and these objects are so decoupled that they may be used in other, non-persistent applications. Because

```
class Subscription(models.Model):
    recipe = models.ForeignKey(Recipe)
    user = models.ForeignKey(User)

    class Admin:
        pass
```

Listing 4: Subscription class.

```
self.assertEqual(len(mail.outbox), 1)
self.assertEqual(mail.outbox[0].subject,
                 'phlombay recipe update: %s' % (self.recipe_masterchef.name))
```

Listing 5: Testing using Django's mock email box.

of this decoupling, the Data Mapper is appropriate when "you want the database schema and the object model to evolve independently"[Fowler(2003), pg 170], and this usually means that both contain much more sophisticated business logic than is found in objects persisted with the Active Record pattern. A simple example of when this is useful is when the persistent objects are part of a complicated class inheritance tree. An implementation of the Data Mapper pattern could be used to persist these objects.

As we will see, implementations of the Data Mapper pattern in Python are possible, but we should first realize that we'll lose some type safety by working in a dynamic language. For example, consider a Java class with `lastModified` field of type `Date`. The compiler prevents programmers from inserting a string or an integer in this field, and thus when we map the field to date column, we do not need to concern ourselves with runtime type checking. However, this is not the case in Python. As all type information is delayed until runtime, we must deal with such checks at runtime and hope that the programmer has complied. While this problem with dynamic languages is not unique to persisting objects, this may be one instance where static type checking is particularly useful.

4 Development

4.1 Local Development

Django comes with an embedded web server and database for development purposes. One particularly useful feature of Django's development server is its transparent handling of updates to the code. If I update, say, the views method that handles the recipe editing page, Django detects my change and instantly restarts the web server. Next time I load that page, my changes will be there. It happens so quickly that for a few weeks we didn't even notice it happening.

For development, we took advantage of the Python implementation of `sqlite`, a flat file-based database system. This spared us the trouble of having to get a MySQL instance running. When the time comes to switch to a real database, Django makes the transition a matter of changing a configuration setting.

4.2 Testing

Django allows developers to use regular unit tests (using Python's `unittest` module) or Python doctests. For our purposes, regular unit tests were suitable. Django's testing framework creates a clean database for each test and allows simulation of login/logout actions and any GET or POST request. In addition to the basic `assertEquals` type of statements, Django also provides some specific assertions, such as `assertRedirects`, which verifies that a particular request returned a redirect HTTP status code, and checks where it redirected to.

One particularly impressive feature of Django's test framework is its handling of email. If a Django application needs to test its email functionality, sending actual email each time the tests are run is not an ideal situation. Therefore, Django provides a mock email box. Listing 5 shows an example of this. The first assertion verifies that one email has arrived in the box, and the second checks to see if the subject line is what we expect it to be. The code comes from our test of recipe subscription functionality.

5 When To Use Django

5.1 Weaknesses

What Django gains in usability it loses in flexibility. Each of its components - the model, view, and controller - is tightly connected to the others, so we do not have the decoupling we see in J2EE applications. This makes reusing components more difficult. Our above discussion of the Data Mapper pattern points toward the possibility of divorcing the data model from the Django architecture, but such an approach is not common among Django developers.

The template engine might also be considered a weakness. While it allows for Python-like code constructs, it is by no means pure Python, and at times its syntax can be cumbersome. As mentioned earlier, it also allows developers to put too much logic in their templates, violating the separation of concerns principle. However, it is possible to use other templating engines with Django, so some of these issues might be mitigated by the developer.

5.1.1 Authentication

Authentication proved to be tricky to implement the way we originally envisioned it. Django's authentication framework wants developers to provide a dedicated login page which would execute some login code and then redirect to the main portion of the site. However, we wanted a login panel to be available on the header for all pages, so we had to write some custom authentication functions. In the end, we were able to get things working, but because of this limitation, we could not simply drop in the authentication app the way we could with other Django components.

A positive consequence of the extra work was that we were able to use Python's decorator syntax. In our implementation, we can annotate methods in `views.py` with the `@require_linecook` decorator. What this does is pass the views function itself to the `require_linecook` function, which checks to see if the user has the correct role, executing the passed-in views function if they do and redirecting them to the front page if they don't. Such an annotation scheme might be likened to declarative authentication in J2EE.

5.1.2 Forms

Django provides a forms API which allows developers to programmatically generate HTML forms for data input and display. There are a variety of data types (character fields, text fields, IP addresses, date/time fields) that map to a particular HTML element or set of elements, and can be combined to create, say, a user profile form.

However, the API has some limitations that we discovered when building our recipe editor page. A recipe might have thirty different ingredients, but providing thirty blank boxes on the form would be ridiculous. Furthermore, providing a button that requests a new form with more boxes seems clunky. Unfortunately, Django's forms API does not provide an easy way to handle varying amounts of data on the client side. We ended up having to write a Javascript hack to deal with it. It works, but it's not pretty.

5.2 Strengths

Compared with the J2EE technologies discussed in Dr. Laufer's course, we found getting work done in Django to be a far more enjoyable experience. Once we understood the basic architecture and flow of execution, it wasn't difficult to add new functionality. As with Python programming in general, more often than not things seemed to "just work." There was a minimum of configuration - no wiring together components in XML. URL mapping in particular happened very intuitively, much better than the J2EE method of setting up forwards in external configuration files.

Additionally, the object-relational mapping system and database API were easy to work with. In Java, external technologies (i.e. Hibernate) must be brought in to provide such functionality; in Django it comes right out of the box.

Django also benefits from an active user base and a hard-working team of developers. It's a reasonably well-documented framework, and we were able to get most of our questions answered from the project website.

References

[Fowler(2003)] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.