Django Architecture Approaches

Adrian Kosmaczewski

2008-04-04

I've just had a very interesting conversation with my colleague Marco about different approaches to the organization of code inside a Django application.

As you might know (and if you don't I'll tell you anyway), Django's views (somehow occupying the "Controller" level in an MVC architecture) must take (at least) an HttpRequest instance as a parameter and must return an HttpResponse instance. That's how it goes in Django, this is the law. This means that you must be sure that the last instruction in your request processing code (in whichever way you've organized it) must return an HttpResponse instance, usually calling the HttpResponse() constructor (or of any of its useful subclasses), or by calling the `django.shortcuts.render_to_response()` function, or something similar.

This has, in my opinion, a major drawback: it might limit code reuse and it increases the coupling in the code. Everything's not lost, however.

Before you start the flame wars, let me explain, using an example coming from the Django website; this represents a basic Django view function, returning some response containing data fetched from the database:

```python
from django.shortcuts import render_to_response, get_object_or_404
# ...
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p})
```

Let's say now that I want to reuse that particular data (the 'p' variable) in another view: given that the return value is always an HttpResponse instance, you are screwed; sometimes you just need the data, to find something, or simply to render it in another format like JSON or XML (RESTful architectures, anyone?). This goes pretty much against the DRY principles, and if you don't go deeper than the Django tutorials, your whole application might feature lots of repeated code.

Even worse, you have a direct reference to a template ("polls/detail.html"), and this kind of coupling does not scale well. It can become a real problem in big projects.

There are, however, strategies to avoid this: the first, the most common, is to refactor your code and to create a "layer" of data-specific functions, which will return instances (or arrays thereof) that you can reuse here and there. Doing this in a big project already started requires a good deal of unit testing first, to ensure that your refactoring is not breaking something elsewhere, but that's another problem (because you DO unit test, right??). This approach might not scale well in complex projects, and thus you would like to organize your code in other ways.

I learnt about organizing views using callable objects instead of functions while studying the code in the Django REST Interface project. In this case, you create code like this:

```
class Resource(ResourceBase):
    """
    Generic resource class that can be used for
    resources that are not based on Django models.
    """

# ... snip ...

def __call__(self, request, *args, **kwargs):
    """
    Redirects to one of the CRUD methods depending
    on the HTTP method of the request. Checks whether
    the requested method is allowed for this resource.
    """
    # Check permission
    if not self.authentication.is_authenticated(request):
        response = HttpResponse(_('Authorization Required'), mimetype=self.mimetype)
        challenge_headers = self.authentication.challenge_headers()
        response._headers.update(challenge_headers)
        response.status_code = 401
        return response

    try:
        return self.dispatch(request, self, *args, **kwargs)
    except HttpMethodNotAllowed:
        response = HttpResponseNotAllowed(self.permitted_methods)
        response.mimetype = self.mimetype
        return response
```

The important bit here is the "**call**" method, which allows an instance to be called as a function, without specifying any particular method. This makes me remember of the dreadful VB default methods but in Python it's not that bad, actually (VB is horrible by default anyway), and allows you to use a cool syntax to do complex tricks ("command pattern" way of doing things, without the method call overload). And of course, since you are using an object-oriented approach, you can use polymorphism and inheritance to organize and reuse code as much as you can (or want).

Finally, Marco told me that his team uses another cool approach: they avoid returning HttpResponse instances from the views, and instead use Python decorators to generate those. This way, you can achieve another neat separation of concerns, and you can reuse code simply and effectively.

I understand that the Python philosophy cares about explicitness, but the "easy" way of processing requests in Django leads to trouble in big applications: increased coupling, reduced DRY, more headaches. I think you should use some code-reuse strategy in your Django code, but this, of course, is more an architectural problem than a Django problem.