Will Hardy, will@hardysoftware.de 24 May 2010

Free the developers

Introduction

This talk isn't for everyone. Most people won't need to do anything I'm going to talk about here. There are probably two groups of people who would be interested in this talk: People (like me) who do a number of similar projects and want to maintain them simultaneously (a single code base) without giving up the ability to customise everything. Or, for people who want to create a framework that is useful for as many other developers as possible.

There are four parts to this talk: Firstly, I'll explain what on earth I'm talking about, secondly I'll look at some design approaches to frameworks. Thirdly I will talk about low level technical approaches and finally some completely nontechnical approaches.

What am I talking about?

The main point of this talk is to promote the idea of writing domain specific frameworks for other django developers. So what do I mean by “domain specific framework” ? Well it's not a reusable app.

Reusable/pluggable apps

- Are great, can work out of the box. I can understand why we have so many, and I think it's great to see people open source their apps.
- ... but they seem to be more useful for end users/admins, or developers who want to build the non-core aspects of their sites quickly
- they also constrain “end developers” (more on them later), forcing them to accept the original developer's way of doing things
- pluggable apps suffer from an inability to share context. See pluggable apps talk at DjangoCon 09 (PBS guys), for a discussion on limitations of eg `django.contrib.comments` app.

Domain specific frameworks

- a framework helps you do 90% of what you need to do more quickly, reliably.
- contrast with an app, which helps on a higher level, doing your work for you.
- Domain specific frameworks have a wider application, that is hopefully useful for an entire domain
- The wider the domain is, the more helpful your framework will be in terms of the number human-hours saved
- Share your domain knowledge! If you are an expert in a domain, share that knowledge and encapsulate it in your framework. (eg CSRF, password refresh hash discussion on google groups) These things are best implemented in frameworks where they can be reused widely as opposed to apps, where only the app's users benefit.
- There are of course downsides: firstly it goes against the agile philosophy of code for today, more importantly it takes a lot more time and effort, which you're probably not being paid directly for.

End developers

- By “end developer”, I mean the end user of your framework, who is a django developer. I figured I'd make up a term to describe this person.
- But who is this person? My talk here is about improving the experience of the end developer, so we need to know what this developer wants.
- Not just any end developer, not a java end developer or a ruby end developer, but a django end developer.
- A Django developer is perfectionist, agile, expressive, elegant.
- I personally use django because I want to provide customised sites, I define the site from the ground up the way the client needs it. Django lets me do this, and quickly.
- Reusable apps step on the feet of a django developer. Evidence for this? They have a known tendency to roll their own.
- One solution is simply to help them roll their own!

And that is what I'm going to talk about. I will look at ways of sharing your domain knowledge by creating reusable frameworks which are most helpful for your typical django developer.

I'll try to demonstrate each of the approaches with an example in a domain I know well (ecommerce apps).

Design Approaches

This is the trickiest part, there are no rules and it really depends on your domain.

Your framework's role

It's useful to identify the exact role your framework will play.

- Doesn't have to do everything. It really, really doesn't.
- Think about the reasons why frameworks are useful: eliminate repetitive code, contain (and test) complicated code.
- You'll probably have to code 3-4 different sites before you can recognise where the developer needs freedom and where they need development short cuts.
- Thinking about the clear role will help avoid your framework from overreaching - leads to feature creep

Example 1: framework role for ecommerce site

What I needed from a framework was to make ecommerce site development fast and reliable, whilst containing the complicated login.

So what are the core features of all ecommerce sites? Categorisation, highlighting, accounting, payment, CRM etc Best to identify these, and keep them as separate (substitutable) as possible: each customised site may want a different way to categorise products etc.

Given that there are other tools for categorisation and highlighting, I decided to limit my framework to 1) calculating totals 2) promote modular design 3) prevent unnecessary recalculations. By limiting the scope and keeping it broad, the framework can be very lightweight. Lightweight means you can test it better :-)

Assumptions and opinions

Another useful approach is to identify your opinions and assumptions. Keeping these out of your framework means the end developer will be more comfortable with your framework (if they have another opinion!) and it will be flexible in ways you might not imagine.

- Everything has a catalog ID? an image? a title? not really.
- Everything has a price? YES!
- In order to avoid assumptions on what models should look like, I divided elements of a financial summary (like an order, or a cart that needs to be totalled) into three types: lists of items, extras, and totals.
- Doing it this way meant that my framework was surprisingly flexible. It worked with a online store, auction site, charity site (donations), freelance consulting invoicing. This was unexpected, but obvious in retrospect: the framework is useful beyond products, images and catalog IDs.

The design is really up to you, as you know your domain best. Be careful to limit the role of your framework appropriately, by concentrating on what is important for a framework. This will then make your design clearer.

The less assumptions and opinions you embed in the framework, the more relevant it will be to a wider developer base - a wider range or projects.

Technical approaches

Moving on to a lower level, I'll look at some technical ways to develop frameworks for freedom loving django perfectionists

Django's core component is the data model. This is the bit I really like, it makes heavily customised data models very easy to implement. But most "frameworks" provide their own models, forcing the end developer to give up control of their data model in order to take advantage of the framework.

The most well known example of this is the `User` model in `django.contrib.auth`.

Providing models means that the pluggable app works out of the box, so it seems to be walking the fine line between flexibility and ease of use / small and large users (what Jacob was talking about earlier). In that case there is a tension, but in many ways you can have both.

There are everyday techniques that you're no doubt familiar with, which can be used to avoid restricting the developer in this way. The ones I'll quickly look at today are: generic relations, model subclassing, managing class/function, signals and template methods.

Generic relations

- Lets users connect their project to your framework without you knowing about it before hand.
- We all know GenericForeignKey from contenttypes (eg `django.contrib.comments`)
- You can also just use a simple backwards relationship, letting the user register their connection (eg user profiles)

Model subclassing

- Lets you define models, which can be extended customised, leaving the required fields intact.
- ... but they can only be customised to a certain extent the end developer will only be able to add to it - they cannot substitute or remove fields.

Example ecommerce:

```
class BaseProduct(models.Model):
    slug    = models.SlugField()
    sku     = models.Charfield(max_length=20)
    title   = models.Charfield(max_length=255)
    price   = models.DecimalField(decimal_places=2, max_digits=10)


class Parrot(BaseProduct):
    plumage = models.CharField(max_length=20)
```

- The `django.contrib.comments` app does this, allowing you to customise more freely
- I personally don't like splitting my model definition between invisible, black-box framework and end developer's project If a developer has control over some of the model, they should have control over the full model.

Signals

- This is a little more crazy, Satchmo does this for searching.
- It seems to be a good way to keep both apps completely separate
- Allows custom work on a relevant instance to take place, very useful

Example ecommerce:
A signal could be sent to allow external apps to contribute to the calculation of a total, eg shipping in a decentralised manner. The downside of this is that the calculation of totals is probably better suited to a centralised design:

```
def delivery(sender, summary, **kwargs):
    summary['delivery'] = Decimal('9.99')

summary_signal.connect(delivery)


>>> summary_signal.send(Cart, instance=cart, summary={})

>>> summary
{'delivery': Decimal('9.99'), 'parrots': Decimal('34.20') }

>>> sum(summary.values())
Decimal('44.19')
```

Template methods

- User can define their own models, but they need to provide certain attributes/ methods
- This is very duck-typey, and allows flexibility and convenience
- Django promotes using the `__unicode__` method on models in this way

Example ecommerce:
The developer's custom model will need to provide a method eg 'get_price' which will be used in calcuations

```
class Parrot(models.Model):
    name   = models.Charfield(max_length=20)
    price  = models.DecimalField(decimal_places=2, max_digits=4)

    def get_price(self):
        return self.price

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

- Downside: the methods that connect the framework to the project are scattered throughout the project. As mentioned earlier, for this domain, a centralised design is probably better suited.
- Downside: the API needs to be obvious, you might want to swallow exceptions if the method is missing, if a default is appropriate

Managing class/function

- Most of the time, a function is enough (it's very simple). This function would take the model instance in question with some arguments that describe the relevant aspects of the instance.

Example ecommerce:
A commerce app might provide a function, which takes a product as an argument and some parameters which describe how to use that product (eg which field/attribute stores the price)

```
total = get_total(parrots, price_field="price")
```

- If you need more intricate customisation (many parameters, functions etc) you can use a managing class with declarative syntax. This can be tricky, so try to avoid using meta classes if you don't have to.

Example ecommerce:
Despite the advice I just gave, this is the approach I eventually took with my ecommerce application. It's familiar to Django developers and allows custom methods to be written nearby.

```
class CartSummary(Summary):

    parrots   = Items(item_amount_from="price")
    returns   = Items(item_amount_from="credit")

    delivery  = Extra()
    discount  = Extra()
    tax       = Extra()

    subtotal  = Total('parrots', 'delivery', 'tax')
    pretax    = Total('parrots', 'returns', 'delivery', 'discount')
    total     = Total()


>>> summary = CartSummary(my_cart)

>>> summary.total
Decimal('44.19')
```

Nontechnical approaches

Sometimes nontechnical solutions are a better approach.

Documentation

- Another way to share your domain knowledge while letting the developers have the freedom they need is to simply document it
- Not everything has to be provided as a utility or function, especially if your framework would allow for an easy implementation
- A framework can and probably should document ways to use the software effectively, empowering the end developer to do it themselves. Or not. They're free to choose.

Example ecommerce:
Invoice IDs can be an art if you care enough. You can use base 62 to keep the length short, randomise it to hide how many sales you've made, and avoid certain letters that sound the same to help customers spell it out over the phone.

Instead of writing code to implement all the different ways you could do this, simply sharing the list of things that need to be considered and possible approaches should allow the end developer to quickly write their own custom version. However they want it to look like.

What was the point of all of that? If you're writing something to be reused, respect the end developer, let them have as much freedom as possible. But not too much :-)