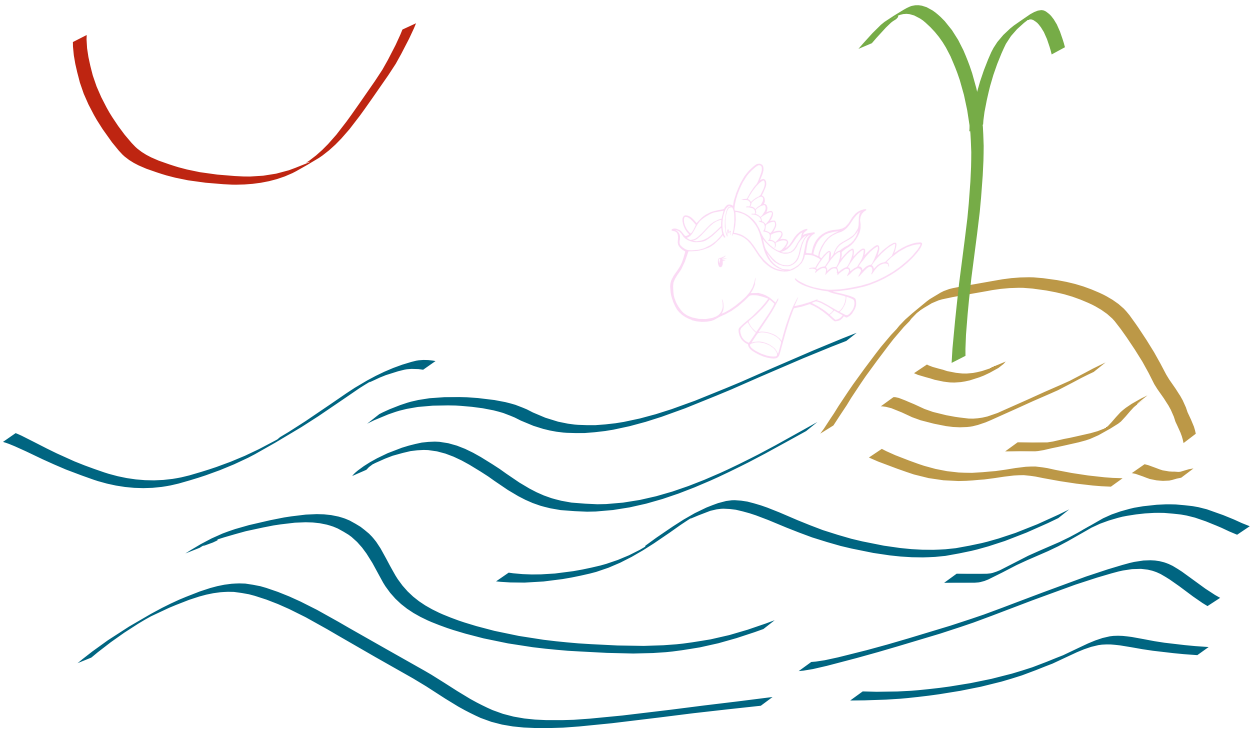
Django: Beyond the SQL



Jerry Stratton
February 26, 2011

Installation

Tutorial files

- RTF tutorial
- PDF tutorial
- Django tar.gz file
- Sample post titles, content, and topics
- Initial HTML template
- Smultron

For PHP programmers

Most of the common programming languages today resemble C. Braces are used to mark off blocks of code, such as for functions, loops, and conditional code. And semicolons are used to mark the end of a piece of code. That style of coding gives you a lot of flexibility in how you format your code, but over time standards have emerged; one of those standards is indentation, another is one one line per statement.

Django uses Python as its programming language. Python doesn't use braces or semicolons. Python was designed with the idea that since you're going to indent anyway, the braces are redundant. So a function that looks like this in PHP:

```
function englishJoin(myList) {
    lastItem = array_pop(myList);
    commaList = implode(', ', myList);
    if (count(myList) > 1) {
        theAnd = ', and ';
    } else {
        theAnd = ' and ';
    }
    englishList = commaList . theAnd . lastItem;
    return englishList;
}
```

might look like this in Python:

```
def englishJoin(myList):
    lastItem = myList.pop()
    commaList = ', '.join(myList)
    if len(myList) > 1:
        theAnd = ', and '
    else:
        theAnd = ' and '
    englishList = commaList + theAnd + lastItem;
    return englishList
```

Where PHP uses “function”, Python uses “def”. You’ll also notice lots of periods. Python uses classes a lot; just about everything is a class in Python. The period is the equivalent of PHP’s -> symbol for accessing properties and methods. So, .pop() is a method on lists, and .join() is a method on strings.

The latter one is a little weird: you don’t tell give a string a concatenator and tell it to pull itself together; you give a concatenator a string and tell it to get to work.

Classes in Python are similar to classes in PHP. Instead of “class ClassName { … }”, you use “class ClassName(object):”, which is the equivalent of PHP’s “class ClassName extends OtherClassName { … }”. In Python, all or almost all of the classes you create will extend another class, even if that class is just the basic “object”.

Requirements

You need Python 2.5 or 2.6, and SQLite 3. Any modern system should have these. If you have Mac OS X Leopard or Snow Leopard, for example, you have it.

You’ll need to perform the installation from your administrative account.

Install

http://www.djangoproject.com/download/

Download the latest Django from http://www.djangoproject.com/. As I write this, the latest release is 1.1.1. Download it; it will be Django-1.1.1.tar.gz. Use ‘gunzip’ to decompress it; then use tar -xvf to dearchive it. You can also just double-click it in Mac OS X to decompress and dearchive it.

Once you’ve got the Django-1.1.1 folder ready, you need to go to the command line. In Mac OS X, this is the Terminal app in your Utilities folder. Type ‘cd’, a space, and then the path to the Django folder (you can probably just drag and drop the folder onto the terminal window after typing ‘cd ’.

Once you’re in the Django folder in the command line, type “sudo python setup.py install”.

Django is now installed. If your administrative account is not your normal account, go back to your normal account.

Create a project

http://docs.djangoproject.com/en/dev/intro/tutorial01/

Use the command line to get to whatever folder you want to store your Django project in. Type “django-admin.py startproject Blog”. Django will create a “Blog” folder and put some initial files into it for you.

Change directory into the Blog folder, and start the development server:

```
python manage.py runserver
```

Watch the messages, and then go to a browser to the URL “http://localhost:8000/”. If it says “It worked!” then you’re good to go.

Edit settings

Your project has a file called “settings.py”. You need to change a few things there, mainly the name and location of the database you’re using.

Your DATABASE_ENGINE should be “sqlite3”. Your DATABASE_NAME should be “Blog.sqlite3”.

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = 'Blog.sqlite3'
```

You’ll also need to tell Django what time zone you’re in:

```
TIME_ZONE = 'America/Los_Angeles'
```

Django comes with a wonderful administration tool for SQL databases, but you need to enable it. Head down to the bottom of settings.py and look for INSTALLED_APPS. Duplicate the last line inside the parentheses (it probably says “django.contrib.sites”) and change the new line to “django.contrib.admin”.

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
)
```

Go to the command line and type “python manage.py syncdb” inside your Blog project folder. This will create your database file. You’ll need to set up a superuser username, password, and e-mail. Remember your password!

If you look in your Blog project folder, you’ll see Blog.sqlite3; this is your SQLite database file.

Finally, open “urls.py” and remove the hash mark (comment) in front of the lines with the comment about enabling the admin:

```
from django.contrib import admin
admin.autodiscover()
...
(r'^admin/', include(admin.site.urls)),
```

Go to “http://localhost:8000/admin/” in your browser, and you should now see the basic administrative page for the built-in user databases.

Create your models

http://docs.djangoproject.com/en/dev/topics/db/models/

Each table in your database is modeled using a class in Python. Django sets up an empty "models.py" file that you should use for these models. We'll set up a very basic blog in models.py. It will have posts, authors, and topics.

The first thing you need to do is create an *app*. An app is a collection of related models. Our tutorial Blog will only have one app, "postings".

```
python manage.py startapp postings
```

This creates a new folder, postings, with models.py and views.py, among other files.

Posts

The basic unit of our blog is the post. Each post has a title, some content, the date it was published, whether or not it is public, some topics, an author, and when it was last edited.

Open models.py and create your Post model.

```
class Post(models.Model):
    title = models.CharField(max_length=120)
    slug = models.SlugField(unique=True)
    content = models.TextField()
    date = models.DateTimeField()
    live = models.BooleanField(default=False)
    topics = models.ManyToManyField(Topic, blank=True)
    author = models.ForeignKey(Author)
    changed = models.DateTimeField(auto_now=True)

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

I like to include a "changed" or "modified" timestamp on every model. It makes a lot of things easier down the line, and it also helps you see recent changes in the admin.

The "slug" is what we'll use for the URL of the post. Titles often have URL-unfriendly characters such as spaces, ampersands, percentages, and other strange things. The slug must be unique, because if two posts had the same slug, there would be no way to differentiate their URLs.

The Post class is a subclass of Django's built-in Model class. We'll see later that this means it inherits a lot of useful functionality.

The "__unicode__" method tells Django, and Python, how to display each instance of the class when displaying it as a string of text.

Authors

Above the Post model, add an Author model. It has to be above the Post model, because the Post model references it as a ForeignKey.

```
class Author(models.Model):
    firstname = models.CharField(max_length=80)
    lastname = models.CharField(max_length=100)
    bio = models.TextField()
    homepage = models.URLField(blank=True)
    changed = models.DateTimeField(auto_now=True)

    class Meta:
        unique_together = (('firstname', 'lastname'),)

    def __unicode__(self):
        return self.firstname + ' ' + self.lastname
```

Some luddites don't have home pages, so we allow that field to be blank. We also require that no author have the same first name and last name. Otherwise, we'd always make mistakes setting the author if we gave two people the same name. If two people have similar names, they'll need to differentiate themselves some way, such as using a middle initial or middle name.

Topics

Above the Post model, add a Topic model.

```
class Topic(models.Model):
    title = models.CharField(max_length=120, unique=True)
    slug = models.SlugField(unique=True)
    changed = models.DateTimeField(auto_now=True)

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

Synchronize your database

Once you've got the model designed, add "Blog.postings" to settings.py. Add another line to the list of INSTALLED_APPS:

```
'INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'Blog.postings',
)
```

Now that Django knows about your app, go to the command line and "python manage.py syncdb". Django will create the necessary database tables for you.

```
$ python manage.py syncdb
Creating table postings_author
Creating table postings_topic
Creating table postings_post
Installing index for postings.Post model
```

Create your administration models

http://docs.djangoproject.com/en/dev/ref/contrib/admin/

Django's administration page will let us create posts, authors, and topics. But we need to tell the admin page about our new app.

Create a new file in the "postings" directory, and call it "admin.py".

```
from django.contrib import admin
from Blog.postings.models import Post, Author, Topic

admin.site.register(Post)
admin.site.register(Author)
admin.site.register(Topic)
```

You'll need to cancel the runserver (CTRL-C) and restart it to get Django to recognize that there's a new file in the postings directory. But once you do, you can go to http://localhost:8000/admin/postings/ to edit your posts, authors, and topics.

Now that you've got a model and an administration screen, go ahead and add H. L. Mencken's "War" essay and George Orwell's "Atomic Bomb" essay.

Customizing the admin

A couple of things immediately come to mind. First, the slug will almost always be very similar to the title. Second, while the list of posts is perfectly serviceable, it could be a lot more useful. When we have a hundred or a thousand posts, it'll be nice if we can see the date, author, and other items in the list of posts.

We can customize the administration screen by subclassing the basic ModelAdmin class and replacing the registration of Post:

```
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'author', 'date', 'live']
admin.site.register(Post, PostAdmin)
```

Notice that the column headers are now links: click on them to change the sorting.

Filters

That's already a lot better, but we can make the admin page even more useful. There are three very useful filters that help us quickly focus on the posts we want to see. Below list_display, add:

```
ordering = ['-changed']
date_hierarchy = 'date'
search_fields = ['title', 'slug']
list_filter = ['author', 'topics', 'live']
```

The first line sets the default order so that the most recently-edited posts show first. The second adds a date bar across the top that lets you focus in on year, then month, then day. The third adds a search box at the top that lets you search for items. And the fourth adds a column on the far right that lets you choose which author, which topic or which status of posts you want to see.

Editable fields

Some fields, such as the “live” field, it may make sense to be able to do more than one at a time. Add this line to the PostAdmin class:

```
list_editable = ['live']
```

The red or green icon will change to a checkbox. You can check or uncheck as many as you wish to change the status of a post.

Pre-populated fields

The slug is usually the title but without spaces and other special characters. We can tell Django to try and handle this for us:

```
prepopulated_fields = {'slug': ('title',)}
```

You can change it if you need to, but most of the time the prepopulation will be fine.

Customize Author and Topic

Go ahead and make a custom ModelAdmin for Author and Topic also:

```
class AuthorAdmin(admin.ModelAdmin):
    search_fields = ['firstname', 'lastname']
    ordering = ['lastname', 'firstname']

admin.site.register(Author, AuthorAdmin)

class TopicAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug']
    search_fields = ['title', 'slug']
    ordering = ['-changed']
    prepopulated_fields = {'slug': ('title',)}

admin.site.register(Topic, TopicAdmin)
```

Public pages

Add more data

At this point, we have a standard set of database tables that we can use the same as we use any tables. We can look at them in Sequel Pro, and we can even display them using PHP on a server that doesn't have Django installed. But that would be a waste of a very good framework.

Finish adding the posts for Orwell and Mencken, so that we can start displaying them.

Views

In a web framework such as Django, you don't have separate pages. What you have are views into a (usually) database. You'll often have one template that supplies one view with as many pages as there are records in the database.

We're going to start with a single view that displays all live posts. In your Blog folder make a folder called "templates", and copy the template.html file into that folder as "index.html".

Open views.py in postings and add:

```
from django.shortcuts import render_to_response

def listPosts(request):
    return render_to_response('index.html')
```

Open urls.py and add this to the bottom of urlpatterns:

```
(r'^$', 'Blog.postings.views.listPosts'),
```

It should look something like this:

```
urlpatterns = patterns('',
    ...
    (r'^admin/', include(admin.site.urls)),
    (r'^$', 'Blog.postings.views.listPosts'),
)
```

The urlpatterns items use regular expressions to send different URL requests to different functions.

Look in settings.py for TEMPLATE_DIRS. Add the full path to your templates folder to it:

```
TEMPLATE_DIRS = (
    ...
    '/Users/jerry/Desktop/Django/Blog/templates',
)
```

If you now go to http://localhost:8000/ you should see the Old Dead Guys Blog main page.

Using the Django API

The usefulness of Django is not that it can send flat files to web browsers. It is that Django provides an API for creating complex dynamic content that can be inserted into templates.

First, modify index.html. Replace " World, say hello. " with:

```
{% for post in postings %}
    <h2>{{ post.title }}</h2>
{% endfor %}
```

Add this to the top of views.py:

```
from Blog.postings.models import Post
```

Change listPosts to:

```
def listPosts(request):
    posts = Post.objects.filter(live=True).order_by('-date')
    context = {'postings': posts}
    return render_to_response('index.html', context)
```

If you see an empty blog page, make sure that you've marked some posts as "live". Otherwise, you should see a series of headlines.

Templates

http://docs.djangoproject.com/en/dev/ref/templates/builtins/

The Django template language is a very simple templating language. It uses {{ … }} to refer to the things you put in the "context" that you send the template. It uses {% … %} to refer to structural code, such as "for… endfor". If something has a property or a method, you can use the period to access that property or method. For example, "postings" was the list of posts we sent it. Each "post" has a title on it, so {{ post.title }} is that title.

Let's fill out the blog's main page. Replace "<h2>{{ post.title }}</h2>" with:

```
<h2>{{ post.title }}</h2>
<p class="author">{{ post.author }}, {{ post.date|date:"F jS, Y h:i a" }}</p>
{{ post.content|linebreaks }}
<p class="topics">{{ post.topics.all|join:", " }}</p>
```

Besides post.xxxx, this template also includes filters. The post's date is filtered through the "date" filter. See the Django documentation for what those letters mean and what other letters are available. The post's content is filtered through "linebreaks", which takes all double blank lines and turns them into paragraphs, and all single blank lines and turns them into
 tags.

Finally, the list of all post topics is joined together on a comma using the "join" filter. The ":" after a filter gives the filter parameters.

Template inheritance

Often, blogs will provide a means of linking to an individual post. That's easy enough to do. We need to be a little careful of polluting our URL namespace, however. Posts can have any slug, but if we want to add new features to our web site, a post might end up interfering with those other URLs.

It is common to create a separate urls.py for each app in a project, and to make all of the features of that project start with the project's name. For example, our archive of past postings might use the URL path /postings/archive/slug.

Add this to urls.py:

```
(r'^postings/', include('Blog.postings.urls')),
```

This will cause any URL beginning with /postings/ to be sent to urls.py in the postings folder.

In your postings folder, create a new urls.py that contains:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('postings.views',
    (r'^archive/(.+)$', 'showPost'),
)
```

Because this urls.py is loaded from URLs beginning with "postings/", the showPost function is called for any URLs beginning with "postings/archive/" and then something. The stuff at the end is a *regular expression*. The "." means "match any character", the "+" means "match one or more of the previous character". So there has to be something after the slash. It can be just about anything, but it has to be something. The parentheses mean, "send whatever this matches to the function".

Add a new method, showPost, to views.py:

```
def showPost(request, postSlug):
    post = get_object_or_404(Post, slug=postSlug, live=True)
    context = {'post': post}
    return render_to_response('post.html', context)
```

You can see that this function has two parameters instead of just one. The "postSlug" parameter is the stuff between the parentheses of the regular expression.

You'll need to change the import to also import "get_object_or_404":

```
from django.shortcuts import render_to_response, get_object_or_404
```

Currently our template's content expects a list of posts. We could create a completely new template, but most of the time we have styled our pages so that subpages are similar. In Django, we can set up blocks of HTML that can be replaced by subpages.

Go into index.html and change the title, the headline, and the content. Everything's the same except that we're surrounding each section with {% block name %} and {% endblock %}.

```
<title>{% block title %}Old Dead Guys Blog{% endblock %}</title>
<h1>{% block headline %}The Old Dead Guys Blog{% endblock %}</h1>

<div id="content">
    {% block content %}
        {% for post in postings %}
            <h2>{{ post.title }}</h2>
            <p class="author">{{ post.author }}, {{ post.date|date:"F jS, Y h:i a" }}</p>
            {{ post.content|linebreaks }}
            <p class="topics">{{ post.topics.all|join:", " }}</p>
        {% endfor %}
    {% endblock %}
</div>
```

Finally, create a new file in your templates folder, "post.html":

```
{% extends "index.html" %}
{% block title %}ODGB: {{ post.title }}{% endblock %}
{% block headline %}Dead Guys: {{ post.title }}{% endblock %}
{% block content %}
    {{ post.content|linebreaks }}
    <p class="topics">{{ post.topics.all|join:", " }}</p>
    <p class="author">
        <a href="{{ post.author.homepage }}">{{ post.author }}</a>,
        {{ post.date|date:"l, F jS, Y h:i a" }}
    </p>
    <div class="bio">{{ post.author.bio|linebreaks }}</div>
{% endblock %}
```

You should now be able to view any individual page by going to the URL http://127.0.0.1:8000/postings/archive/slug, replacing "slug" with the slug for that posting (such as http://127.0.0.1:8000/postings/archive/hanging). Remember that if the post hasn't been made public yet, it won't be displayed.

Linking

If you want people to link to your posts, it's a good idea to provide links. Django can provide the URL of a model's instance using the {% url %} tag.

In index.html, replace the <h2>{{ post.title }}</h2> headline with:

```
<h2><a href="{% url postings.views.showPost post.slug %}">{{ post.title }}</a></h2>
```

This will link each posting's headline on the main page with that post's individual page.

If… Then

If you take a look at post.html, there's a potential problem with the author link. We link to the author's home page, but homepage is an optional field. If the author doesn't have a home page, that link will just go to the same page.

There's a template tag for "if" to help with situations like this.

```
<p class="author">
    {% if post.author.homepage %}
        <a href="{{ post.author.homepage }}">{{ post.author }}</a>,
    {% else %}
        {{ post.author }}
    {% endif %}
    {{ post.date|date:"l, F jS, Y h:i a" }}
</p>
```

You can find other "ifs" at http://docs.djangoproject.com/en/dev/ref/templates/builtins/.

Topic template

Often, blogs will have a page showing all posts on a particular topic. Create a new template, “topic.html”:

```
{% extends "index.html" %}
{% block title %}ODGB Topic: {{ topic.title }}{% endblock %}
{% block headline %}Dead Guys Topic: {{ topic.title }}{% endblock %}
```

Create a new function in views.py:

```
def topicalPosts(request, topicSlug):
    topic = get_object_or_404(Topic, slug=topicSlug)
    posts = Post.objects.filter(live=True, topics=topic).order_by('-date')
    context = {'postings': posts, 'topic': topic}
    return render_to_response('topic.html', context)
```

And add Topic to the list of models being imported:

```
from Blog.postings.models import Post, Topic
```

In urls.py, make this view use the URL /topic/slug:

```
(r'^topic/(.+)$', 'topicalPosts'),
```

This will send any request for the URL …/postings/topic/slug to the “topicalPosts” function. Go ahead and view it as http://127.0.0.1:8000/postings/topic/slug, replacing “slug” with the slug for one of your topics, such as http://127.0.0.1:8000/postings/topic/war.

Include subtemplates

We’ll also link the topics to their listing pages, but that’s going to make the topics “for” loop long. Rather than maintain it in two places, we’re going to separate it out to its own template file.

In index.html and post.html, replace “ {{ post.topics.allljoin:", " }} ” with:

```
{% include "parts/post_topics.html" %}
```

Then, make a folder called “parts” in your templates folder, and put this in post_topics.html:

```
<p class="topics">
    {% for topic in post.topics.all %}
        <a href="{% url postings.views.topicalPosts topic.slug %}">{{ topic.title }}</a>{% if
            not forloop.last %},{% endif %}
    {% endfor %}
</p>
```

For loops can be nested; you can also check to see if this is the first or last item using forloop.first or forloop.last. In this case, a comma isn’t necessary after the last item.

Simple redirects

If you take a look at our URL structure right now, we've got / for all postings, /postings/archive/slug for individual posts, and /postings/topic/slug for topical listings. There are three URLs that people might expect to use but that don't have anything matching them: /postings, /postings/archive, and /postings/topic. The latter two I'll leave to you as an exercise. It might make sense to limit the main page to only the top 20 postings and put the rest of them on the archive page, and perhaps show a listing of all available topics on the topic page.

But what about /postings? It makes sense for /postings to show the same information that the main page shows. Perhaps some day, when the site will contain a blog section, a news section, an entertainment section, and the main page will reflect that, relegating the blog postings to the /postings URL will make sense. But for now, /postings and / are the same thing.

Rather than have two URLs that display the same information, we can redirect one URL to another one.

In urls.py for the postings project, add another line:

```
(r'^$', 'goHome'),
```

Because this is in the posting's urls.py, the URL automatically begins with postings/. By using the regular expression for "nothing", this line will match only postings/ and nothing else. When matched, it will call the goHome function.

So, in views.py, add goHome:

```
def goHome(request):
    homePage = reverse('Blog.postings.views.listPosts')
    return HttpResponseRedirect(homePage)
```

You'll notice two new functions here: reverse and HttpResponseRedirect. Import them at the top of views.py.

```
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
```

If you now go to http://127.0.0.1:8000/postings/ you will be immediately redirected to http://127.0.0.1:8000/. The "reverse" function is a lot like the "url" template tag. It takes the name of a view and returns the URL that the view belongs to. HttpResponseRedirect is like render_to_response, except that it creates a response specifically optimized for redirects.

Customization

Remember that you can add any method to your models, and use that method in your templates. If you have a method on the Post class called "otherAuthorPosts" that returns all of the other posts by this posting's author, you can reference it as "post.otherAuthorPosts". You can also provide any variable—including classes, lists, and dictionaries—to the context that your view

creates, and pass it to your templates. But there are also special-purpose customizations that you can do to your Django apps.

Custom managers

http://docs.djangoproject.com/en/dev/topics/db/managers/

Just about everything is an object in Django. The "objects" that you use for "Blog.objects.filter()", "Blog.objects.all()", and "Blog.objects.exclude()" is a "manager" object, and you can override it. For example, it might be nice to be able to post-date blog articles so that they become public at a specific time. If we do that, we need to pay attention to both "live=True" and "Date__lte=datetime.now()". Currently, that means changing two functions, but you can imagine a more complex blog where the code asks for "all live posts" all over the place.

It would be nice to have a single place to ask for all live—or all public—posts. We can override the manager, and then tell Django to use our custom manager for all uses of Post.objects.

First, edit models.py to add the new manager.

```
class PostManager(models.Manager):
    def public(self):
        return self.all().filter(live=True, date__lte=datetime.now()).order_by('-date')
```

There are several __ modifiers you can use in filter and exclude; "lte" means "less than or equal to". You'll also need to import the "datetime" functionality so that it can use "datetime.now()" to get the current date and time:

```
from datetime import datetime
```

And then tell the Post object to use this new manager:

```
class Post(models.Model):
    title = models.CharField(max_length=120)
    ...
    changed = models.DateTimeField(auto_now=True)

    objects = PostManager()
```

Then, in listPosts in views.py, change "posts = Post.objects.filter(live=True).order_by('-date')" to:

```
posts = Post.objects.public()
```

And in showPost, change "post = get_object_or_404(Post, slug=postSlug, live=True)" to:

```
post = get_object_or_404(Post.objects.public(), slug=postSlug)
```

The "or_404" functions accept both a model class and a queryset.

And finally, in topicalPosts, change "posts = Post.objects.filter(live=True, topics=topic).order_by('-date')" to:

```
posts = Post.objects.public().filter(topics=topic)
```

Yes, you can chain filter and exclude. If you have any custom manager methods, however, they must go first, and you can't use more than one custom manager method in the chain.

If you now change the date and time on one of the posts to be a minute from now, it will temporarily disappear from the list, and then reappear when its time comes. And it will do this on every page that used to display it.

Extra filters and tags

http://docs.djangoproject.com/en/dev/ref/contrib/humanize/

The topics page ought to say how many posts are in that topic. Add this to topic.html:

```
{% block content %}
    <p>The Old Dead Guys have posted {{ postings.count }} time{{ postings|pluralize }} on
            <em>{{ topic.title }}</em>.</p>
    {{ block.super }}
{% endblock %}
```

Here, we're not just overriding the content block, we're modifying it: "block.super" uses the parent block's content as well.

When we get to ten or more posts, it will be fine to use numbers instead of words, but it would be nice to use words "two times" instead of "2 times" when the numbers are nine or below.

There are a collection of extra filters for "humanizing" numbers, but we need to specifically load them into our template. In your settings.py file, add django.contrib.humanize to INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'django.contrib.humanize',
    'Blog.postings',
)
```

At the top of topic.html, add:

```
{% load humanize %}
```

And then add the "apnumber" filter to postings.count:

```
<p>The Old Dead Guys have posted {{ postings.count|apnumber }} time{{ postings|
            pluralize }} on <em>{{ topic.title }}</em>.</p>
```

If you need to generate random text, look at the "django.contrib.webdesign" for a lorem ipsum tag: http://docs.djangoproject.com/en/dev/ref/contrib/webdesign/.

Custom tags

http://docs.djangoproject.com/en/dev/howto/custom-template-tags/

You can make your own tags as well. You need a folder to put them in. Create a "templatetags" folder inside of your "postings" folder. It must be a Python package; this means it must have a file called "__init__.py" inside it. The file can be empty, but it must be there:

```
cd postings
mkdir templatetags
touch templatetags/__init__.py
```

You can create as many .py files in the templatetags folder as you want. If you want to load the tags (or filters) from a particular file into a template, use "{% load filename %}", without the ".py". For example, if you have a "media.py" file with tags and filters, you can load them with "{% load media %}".

Let's say we want to be able to include Youtube videos throughout our pages. Youtube has complex code for embedding videos; we can make a template snippet just for embedding Youtube videos, and make a tag that renders that code.

```
{% embed "6ugx0ZO239Y" %}
{% embed "TO68zwTXFWk" "wide" %}
```

Create a media.py file with:

```
from django import template

register = template.Library()
@register.simple_tag
def embed(videoCode, aspect="standard"):
    if aspect == 'wide':
        width = 560
        height = 340
    else:
        width = 425
        height = 344

    context = {'videocode': videoCode, 'width': width, 'height': height}
    return template.loader.render_to_string('parts/youtube.html', context)
```

Add "{% load media %}" to the top of index.html. Then, add a "Video of the day" section to the top of the content div:

```
<div id="content">
    <h2>Video of the day!</h2>
    {% embed "6ugx0ZO239Y" %}

    ...
</div>
```

You should see a clip from the L'il Abner musical at the top of the index page and the topics page. Replace the embed tag with:

```
{% embed "TO68zwTXFWk" "wide" %}
```

And you should see a widescreen clip from Ferris Bueller's Day Off instead.

You can also use model properties and methods in tags. If you had a slugfield for Youtube videos on each post, called "video", you could use this to display it:

```
{% embed post.video %}
```

Django will pull the Youtube code string out of post.video and send that to the "embed" tag.

You don't need to add your custom templatetags files to settings.py: Django automatically looks into each app folder, and makes the files in that app's templatetags folder available to any app. Django knows that "embed" is a tag because of the "@register.simple_tag" *decorator* above the function. Decorators are a feature of Python that adds common functionality to functions. Anything with an @ symbol above a function definition "decorates" that function.

Custom filters

http://docs.python.org/library/re.html

Filters are the things that use a pipe (|) to alter a value. For example, we used the pipe earlier to filter a number into a word.

Filters require more care, because Django (like many web scripting environments) maintains the concept of "safe" and "unsafe" text. Unsafe text must be "escaped" so that all HTML is not rendered; this avoids cross-site scripting attacks.

So a filter that returns HTML needs to know whether it needs to escape its text before adding the HTML; and then once it adds the HTML it needs to mark the result as safe so that other filters don't escape it.

This filter will link any mention of a topic to that topic's page:

```
from django.utils.safestring import mark_safe
from django.utils.html import conditional_escape
from Blog.postings.models import Topic
import re

@register.filter
def linkTopics(text, autoescape=None):
    if autoescape:
        topicalText = conditional_escape(text)

    linkTemplate = template.loader.get_template('parts/inline_topic.html')
    for topic in Topic.objects.all():
        regex = re.compile('\\b(' + topic.title + ')\\b', re.IGNORECASE)
        replacement = linkTemplate.render(context=template.Context({'topic': topic}))
        topicalText = re.sub(regex, replacement, topicalText, 1)
    topicalText = mark_safe(topicalText)
```

```
    return topicalText
linkTopics.needs_autoescape = True
```

Like the tag, we have to register this filter using a decorator so that Django knows about it. Because this filter is returning HTML, we need to tell Django that the filter needs to know whether or not autoescape is on. That's what "linkTopics.needs_autoescape = True" does.

If autoescape is on, the text is first escaped, so that HTML tags (such as <script>) are escaped (become <script>).

Then, we load the template for the link so that we can use it over and over again.

Inside the loop, a regular expression converts any occurrence of any topic title to a link to that topic's page. The replacement is case-insensitive, and happens only to the first instance of the topic title in the text being filtered.

Create parts/inline_topic.html:

```
<a href="{% url postings.views.topicalPosts topic.slug %}">\1</a>
```

And load this templatetags file into post.html:

```
{% load media %}
```

Replace "{{ post.content|linebreaks }}" with:

```
{{ post.content|linkTopics|linebreaks }}
```

You should now see that topics are linked to their listing when you read individual posts. Make the same change to the author's bio:

```
{{ post.author.bio|linkTopics|linebreaks }}
```

If you want, you can go ahead and make the same change in index.html, so that topics are linked in the main listing and in the topics listings.

By putting all of your HTML into templates, you can easily change the HTML without worrying about adding bugs to your code. For example, you can easily add a class of "topics" to the <a> tag in inline_topic.html, and then add a style for it in index.html:

```
a.topic {
    border: solid .1em;
    border-top: none;
    border-bottom: none;
    padding-right: .2em;
    padding-left: .2em;
}
```

Because of template inheritance, that style will also be available on topics pages and post pages.

Custom admin actions

http://docs.djangoproject.com/en/dev/ref/contrib/admin/actions/

If we're doing a more newspaper-style publication, we might want to be able to quickly set both the live and the date of the posts at the same time. While editable_fields makes it easy to change one field across multiple posts, they don't let us perform more complex actions.

In the PostAdmin class, add a new method called "makePublic()". This method will accept a list of postings and will set "live" to True and "date" to midnight on the morning of the current day. The method also requires two imports at the top of the file to help it along.

```
from django.template import Template, Context
from datetime import date
...

class PostAdmin(admin.ModelAdmin):
    ...
    actions = ['makePublic']

    def makePublic(self, request, chosenPostings):
        today = date.today()
        for post in chosenPostings:
            post.live = True
            post.date = today
            post.save()

        #send notification to the admin
        message = Template('{% load humanize %}Made {{ count|apnumber }} post
            {{ count|pluralize }} public as of {{ date }}')
        messageContext = Context({'count':chosenPostings.count(), 'date': today})
        messageText = message.render(messageContext)
        self.message_user(request, messageText)
    makePublic.short_description = 'Make selected posts public'
```

There are three parts to an admin action. First, the action must be listed in the list of "actions". Second, the action needs a method in the ModelAdmin class. And third, the method needs a short description for use in the admin pages.

Django's admin sends this method a "request" (the same kind we used already in views), as well as the list of postings that the editor has chosen. The method loops through each post, sets "live" and "date", and then saves the post.

The largest part of the method provides feedback that the action was performed successfully. It creates a quick template so that we can more easily handle plurals and friendly numbers, creates a context for that template, and then renders it to the text that is pushed into the editor's message queue.

Check the pull-down "action" menu on the Post administration page. It should now have a new option below "Delete selected posts": "Make select posts public".

Cacheing

http://docs.djangoproject.com/en/1.0/topics/cache/

Once you get really into the power of a framework like Django, you'll start using it to add really cool features you would never have had the time for if you had to code them from scratch. Some of those really cool features will end up taking more time than your server can afford. Django also has a cacheing mechanism that you can use to cache expensive tasks. For example, looking at that custom autoLink filter, you might discover, after adding a thousand topics, that auto linking text takes too long.

When your tests show you that some feature is taking too much time and causing requests to queue on the server, you can either cache the entire page, or cache portions of the page.

If you want to use cacheing, the first thing you need to do is modify your settings.py file.

```
CACHE_BACKEND = 'file:///Users/jerry/Desktop/Django/Blog/cache?max_entries=500'
```

The "max_entries" value is the maximum number of caches Django will create before it starts throwing out entries. When that number of caches are created, Django will arbitrarily remove about a third of the caches.

Make sure that the folder you're using exists; in this case, there needs to be a "cache" folder inside the Blog folder I've made for testing.

Page cacheing

The easiest way to set up cacheing is to just tell Django to cache every page. You can do this by adding a cacheing "middleware". Look for MIDDLEWARE_CLASSES in settings.py. "Middleware" is a low-level "plugin" system for altering page requests; that includes things like providing a cached version of a page instead of recreating it for every request.

We need a middleware at the very beginning of the list, and a middleware at the very end. Change MIDDLEWARE_CLASSES to:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

Also, add this line below that:

```
CACHE_MIDDLEWARE_SECONDS = 120
```

That's the number of seconds that full pages should be cached.

Your pages will now be cached for two minutes. You can test this by making a change in one of your posts immediately after viewing the public version; the public version will not immediately change, but if you wait two minutes, it will change.

Template tag cacheing

Sometimes you don't want to cache an entire page. You have a site where some things on the pages change every time the page is loaded. But you also don't want to recreate expensive tasks that rarely change, such as the list of authors or topics. You can hook directly into the cacheing system using a template tag

Comment out (put a hash mark in front of) the two middleware.cache lines in MIDDLEWARE_CLASSES in settings.py, to disable per-page caching. Also, comment out the CACHE_MIDDLEWARE_SECONDS line, as it's unnecessary.

At the very top of the parent template, add:

```
{% load cache %}
```

And around the video area, put: the cache tag:

```
{% cache 600 video %}
    <h2>Video of the day!</h2>
    {% embed "6ugx0Z0239Y" %}
{% endcache %}
```

This will cache that section of the template for a five minutes—600 seconds.

The {% cache %} template tag caches based on the name you give the tag. If you use that tag in multiple places, each place will share the same cache. So if you want to cache a posting in post.html, you can't just say {% cache 120 post %}:

```
{% load cache %}
...
{% block content %}
    {% cache 120 post %}
        {{ post.content|linkTopics|linebreaks }}
        {% include "parts/post_topics.html" %}
        <p class="author">
            {% if post.author.homepage %}
                <a href="{{ post.author.homepage }}">{{ post.author }}</a>,
            {% else %}
                {{ post.author }}
            {% endif %}
            {{ post.date|date:"l, F jS, Y h:i a" }}
        </p>
        <div class="bio">{{ post.author.bio|linkTopics|linebreaks }}</div>
    {% endcache %}
{% endblock %}
```

Try it, and you'll see that for two-minute intervals, each post is exactly the same.

We need to give the {% cache %} tag another identifier to differentiate each post. For example, each of our posts has a unique slug, so we can use that as an identifier:

```
{% cache 120 post post.slug %}
...
{% endcache %}
```

If you try it now, each post will cache for two minutes, but will not pollute other posts with their cache.

The cache identifier is shared across pages, so if we want to cache the postings as they're displayed on a listing page, we need to use a different identifier; "post_index" instead of "post", for example. Modify index.html again:

```
{% block content %}
    {% for post in postings %}
        {% cache 120 post_index post.slug %}
            <h2><a href="{% url postings.views.showPost post.slug %}">{{ post.title }}</
            a></h2>
            <p class="author">{{ post.author }}, {{ post.date|date:"F jS, Y h:i a" }}</p>
            {{ post.content|linkTopics|linebreaks }}
            {% include "parts/post_topics.html" %}
        {% endcache %}
    {% endfor %}
{% endblock %}
```

The main page and the topics pages will now only regenerate each post entry if two minutes have past. And because we're caching each post rather than the entire list (by putting "{% cache %}" around the for loop), new postings will still show up immediately. With per-page cacheing, new postings won't show up until after the page's cache expires.

Custom cacheing

The above two methods are very easy. They cache for a specific amount of time and then recreate the page or section on the next request after the cache expires. If the page doesn't need recaching, Django still recaches it, and if the page does need recaching Django will still wait until the cache expires. That's usually fine: you'll set cache expiration to be some reasonable amount of time, and five or ten minutes here or there won't matter.

Sometimes, though, we know exactly when a piece of the page needs to be recached. For example, in our current blog example, past postings probably never change, so we could set the cache value to an extremely high amount—except that when a page does change, we would want the change to take effect quickly. So we end up recaching unchanging information every couple of minutes on the off chance that someday the author makes a correction to their post.

The most likely reason for caching a post in our above code is that linkTopics filter. If we know that the last cache was made *after* the last topic was changed and the text itself was changed, then we know we don't need to recreate it. It doesn't matter if the cache was created two hours

ago or two months ago: if there haven't been any new topics in that time and the text hasn't changed, we don't need to recache.

In situations like that, we can perform the cache within our code. At the top of your templatetags/custom.py file, add:

```
from django.core.cache import cache
from datetime import datetime
```

If you still have page cacheing middleware, remove it, and also remove the {% cache %} tag from around the post in index.html and post.html. (You can leave it around the video if you still have it there; it won't affect this example.)

The cache module includes cache.set, for setting the cache, and cache.get, for getting something out of the cache if it exists. In your linkTopics function, add some new code at the top and bottom:

```
def linkTopics(text, autoescape=None):
    #check for precached topicalized text
    (cacheStamp, topicalText) = cache.get(text, (None, None))
    newStamp = datetime.now()

    #if the text hasn't changed we need to check for new or changed topics
    if not topicalText or Topic.objects.filter(changed__gte=cacheStamp):
        #recreate the cache
        #print "Caching topicalization"
        if autoescape:
            topicalText = conditional_escape(text)

        linkTemplate = template.loader.get_template('parts/inline_topic.html')
        for topic in Topic.objects.all():
            regex = re.compile('\\b(' + topic.title + ')\\b', re.IGNORECASE)
            replacement = linkTemplate.render(context=template.Context({'topic': topic}))
            topicalText = re.sub(regex, replacement, topicalText, 1)

        topicalText = mark_safe(topicalText)

    #cache both the time of this cache, and the linked text
    #can't timeout never, so timeout in a year instead
    cache.set(text, (newStamp, topicalText), 31536000)

    return topicalText
linkTopics.needs_autoescape = True
```

You can uncomment the print line and watch the terminal to see when the function is caching.

The cache.get() method takes two parameters: the "key" for the cache and the default return value. By using the text itself as the key, if the text changes cacheing will be triggered automatically. If there is no matching text in the cache, the default return value gets None for the text, and the current date and time for the cacheStamp.

If no cached topicalText was returned, obviously we need to recreate the topicalText. But we also need to recreate it if there are new topics or if topic titles have changed. So, after "not topicalText" we say "or there are any topics whose changed field is greater than or equal to the cacheStamp". If topicalText is empty, Python never bothers to run that filter, because it doesn't need to—it already knows the "if" is True. But if topicalText has something in it, it runs that filter, and if anything at all comes back, we need to create the topicalized text.

Finally, at the end, we always set the cache again with a timeout of 31,536,000 seconds. That's one year. Unfortunately, Django's cache does not allow you to cache forever. All you can do is cache for a really long time. In theory, since the cache stamp is continually refreshed a one-year timeout will never be reached except for extremely unpopular postings[1].

We cache both the time we started looking at the text and the new topicalized text; for the key, we use the original text[2]. Thus, if the original text ever changes that will trigger a recache.

When to worry about cacheing?

Generally, you don't want to go overboard when cacheing. Cacheing adds complexity, and you often don't know where it's needed. When you've finished adding a feature, you can test how long it takes using the {% now %} tag in your code and by calculating elapsed time inside of your Python code.

```
from datetime import datetime
...
start = datetime.now()
...
elapsed = datetime.now() - start
print "Time spent calculating swallow speed: ", elapsed
```

Watching your cache

Determining the right number of max_entries can be tricky. In this simple example, we know exactly how many entries we'll be caching per post, but in a more complex setup we might only be able to guess. So it's useful to watch the cache fill up, especially immediately before and after loading a page.

Go to the command line and cd into your Blog folder (the same folder that has "manage.py" in it).

```
$ python manage.py shell
>>> from django.core.cache import cache
>>> cache._max_entries
500
>>> cache._num_entries
```

[1] And because search engines will hit it regularly, even unpopular postings will get restamped.

[2] If you select a caching mechanism that imposes size limits on the key, look into the hashlib module and md5. You should be able to get a usable key using "hashlib.md5(text).digest()".

```
14
```

Visit a new page, and you should see the number change:

```
>>> cache._num_entries
15
```

It's important, if you have a complex system, to ensure that the maximum number of cache entries is significantly greater than the number of cache entries used per page view.

Modifying models

Django's syncdb command only creates new models. It doesn't update existing models. If you make a change to your model in models.py, you will also need to make a corresponding change to the database table for that model.

MySQL

> http://www.sequelpro.com/

In MySQL it's usually pretty easy to add, modify, or delete a column in a table.

SQLite

SQLite makes it easy to add a column, but not to modify or delete one. If you need to modify or delete a column, you need to rename the table, recreate the table with the new structure (you can use syncdb for this) and then copy the old data back into the new table.

Scripting Django

Django's amazing API is useful for all sorts of tasks, not just web tasks. Any command-line script that deals with a database can take advantage of Django, and often your web databases have command-line scripts that work with them.

Validate pages?

Probably not, it's probably too much to require an easy_install. What else would be good for command-line looping?