django-bakery Documentation

Release 0.10.5

Los Angeles Times Data Desk

Jul 27, 2018

Contents

A set of helpers for baking your Django site out as flat files

CHAPTER 1

Why and what for

The code documented here is intended to make it easier to save every page generated by a database-backed site as a flat file. This allows you to host the site using a static-file service like Amazon S3.

At the Los Angeles Times Data Desk, we call this process "baking." It's our path to cheap, stable hosting for simple sites. We've used it for publishing election results, timelines, documents, interactive tables, special projects and numerous other things.

The system comes with some major advantages, like:

1. No database crashes
2. Zero server configuration and upkeep
3. No need to optimize your app code
4. You don't pay to host CPUs, only bandwidth
5. An offline administration panel is more secure
6. Less stress (This one can change your life)

There are drawbacks. For one, you have to integrate the "bakery" into your code base. More important, a flat site can only be so complex. No online database means your site is all read and no write, which means no user-generated content and no complex searches.

Django's class-based views are at the heart of our approach. Putting all the pieces together is a little tricky at first, particularly if you haven't studied the Django source code or lack experience working with Python classes in general. But once you figure it out, you can do all kinds of crazy things: Like configuring Django to bake out your entire site with a single command.

Here's how.

CHAPTER 2

Documentation

2.1 Getting started

2.1.1 Installation

Before you begin, you should have a Django project created and configured.

Install our library from PyPI, like so:

```
$ pip install django-bakery
```

Edit your `settings.py` and add the app to your `INSTALLED_APPS` list.

```
INSTALLED_APPS = (
    # ...
    # other apps would be above this of course
    # ...
    'bakery',
)
```

2.1.2 Configuration

Also in `settings.py`, add a build directory where the site will be built as flat files. This is where bakery will create the static version of your website that can be hosted elsewhere.

```
BUILD_DIR = '/home/you/code/your-site/build/'
```

The trickiest step is to refactor your views to inherit our *buildable class-based views*. They are similar to Django's generic class-based views, except extended to know how to automatically build themselves as flat files.

Here is a list view and a detail view using our system.

```
from yourapp.models import DummyModel
from bakery.views import BuildableDetailView, BuildableListView


class DummyListView(BuildableListView):
    """
    Generates a page that will feature a list linking to detail pages about
    each object in the queryset.
    """
    queryset = DummyModel.live.all()


class DummyDetailView(BuildableDetailView):
    """
    Generates a separate HTML page for each object in the queryset.
    """
    queryset = DummyModel.live.all()
```

If you've never seen class-based views before, you should study up in the Django docs because we aren't going to rewrite their documentation here.

If you've already seen class-based views and decided you dislike them, you're not alone but you'll have to take our word that they're worth the trouble in this case. You'll see why soon enough.

After you've converted your views, add them to a list in `settings.py` where all buildable views should be recorded as in the `BAKERY_VIEWS` variable.

```
BAKERY_VIEWS = (
    'yourapp.views.DummyListView',
    'yourapp.views.DummyDetailView',
)
```

2.1.3 Execution

Then run the management command that will bake them out.

```
$ python manage.py build
```

This will create a copy of every page that your views support in the `BUILD_DIR`. You can review its work by firing up the `buildserver`, which will locally host your flat files in the same way the Django's `runserver` hosts your dynamic database-driven pages.

```
$ python manage.py buildserver
```

To publish the site on Amazon S3, all that's necessary yet is to create a "bucket" inside Amazon's service. You can go to aws.amazon.com/s3/ to set up an account. If you need some basic instructions you can find them here. Then set your bucket name in `settings.py`.

```
AWS_BUCKET_NAME = 'your-bucket'
```

While you're in there, you also need to set your credentials.

```
AWS_ACCESS_KEY_ID = 'your-key'
AWS_SECRET_ACCESS_KEY = 'your-secret-key'
```

Finally, now that everything is set up, publishing your files to S3 is as simple as:

```
$ python manage.py publish
```

You should be able to visit your bucket's live URLs and see the site in action. To make your bucket act more like a normal website or connect it to a domain you control follow these instructions.

2.1.4 Optimization

If you are publishing to S3, you can reduce the size of HTML, JavaScript and CSS files by having bakery compress them using gzip as they are uploaded. Read more about this feature here, here or here.

Getting started is as simple as returning to `settings.py` and adding the following:

```
BAKERY_GZIP = True
```

Then rebuilding and publishing your files.

```
$ python manage.py build
$ python manage.py publish
```

2.2 Common challenges

2.2.1 Configuring where detail pages are built

If you are seeking to publish a detail page for each record in a database model, our recommended way is using the *BuildableDetailView*.

When the view is executed via bakery's *standard build process*, it will loop through each object in the table and build a corresponding page at a path determined by the view's `get_url` method.

You can override `get_url` to build the pages anywhere you want, but the easiest route is by configuring Django's standard get_absolute_url method on the model, which is where `get_url` looks by default.

Here's an example. Let's start with a model that will contain a record for each of America's 50 states. Notice how we have defined Django's standard `get_absolute_url` method to return a URL that features each state's unique postal code.

```
from django.db import models
from bakery.models import BuildableModel


class State(BuildableModel):
    name = models.CharField(max_length=100)
    postal_code = models.CharField(max_length=2, unique=True)

    def get_absolute_url(self):
        return '/%s/' % self.postal_code
```

That model is then connected to a `BuildableDetailView` that can create a page for every state.

```
from myapp.models import State
from bakery.views import BuildableDetailView


class StateDetailView(BuildableDetailView):
```

(continues on next page)

(continued from previous page)

```
    model = State
    template_name = 'state_detail.html'
```

As described in the *getting started guide*, that view will need to be added to the `BAKERY_VIEWS` list in `settings.py`.

Now, because the URL has been preconfigured with `get_absolute_url`, all 50 pages can be built with the standard management command (assuming your settings have been properly configured).

```
$ python manage.py build
```

That will create pages like this in the build directory.

```
build/AL/index.html
build/AK/index.html
build/AR/index.html
build/AZ/index.html
build/CA/index.html
build/CO/index.html
... etc ...
```

If you wanted to build objects using a pattern independent of the model, you can instead override the `get_url` method on the `BuildableDetailView`.

```
from myapp.models import State
from bakery.views import BuildableDetailView


class ExampleDetailView(BuildableDetailView):
    model = State
    template_name = 'state_detail.html'

    def get_url(self, obj):
        return '/my-fancy-pattern/state/%s/' % obj.postal_code
```

That will create pages like this in the build directory.

```
build/my-fancy-pattern/state/AL/index.html
build/my-fancy-pattern/state/AK/index.html
build/my-fancy-pattern/state/AR/index.html
build/my-fancy-pattern/state/AZ/index.html
build/my-fancy-pattern/state/CA/index.html
build/my-fancy-pattern/state/CO/index.html
... etc ...
```

2.2.2 Building JSON instead of HTML

Suppose you have a view the acts like an API, generating a small snippet of JSON. In this case, the official Django documentation recommends the following usage of class-based views to render the page in a dynamic website.

```
import json
from django.http import HttpResponse
from django.views.generic import TemplateView


```

(continues on next page)

(continued from previous page)

```
class JSONResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return HttpResponse(
            self.convert_context_to_json(context),
            content_type='application/json',
            **response_kwargs
        )

    def convert_context_to_json(self, context):
        "Convert the context dictionary into a JSON object"
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return json.dumps(context)


class JSONView(JSONResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)

    def get_context_data(self, **kwargs):
        return {'this-is': 'dummy-data'}
```

The same design pattern can work with django-bakery to build a flat version of the JSON response. All that's necessary is to substitute a buildable view with some additional configuration.

```
import json
from django.http import HttpResponse
from bakery.views import BuildableTemplateView


class JSONResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return HttpResponse(
            self.convert_context_to_json(context),
            content_type='application/json',
            **response_kwargs
        )

    def convert_context_to_json(self, context):
        "Convert the context dictionary into a JSON object"
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
```

(continues on next page)

(continued from previous page)

```
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return json.dumps(context)


class BuildableJSONView(JSONResponseMixin, BuildableTemplateView):
    # Nothing more than standard bakery configuration here
    build_path = 'jsonview.json'

    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)

    def get_context_data(self, **kwargs):
        return {'this-is': 'dummy-data'}

    def get_content(self):
        """
        Overrides an internal trick of buildable views so that bakery
        can render the HttpResponse substituted above for the typical Django
        template by the JSONResponseMixin
        """
        return self.get(self.request).content
```

2.2.3 Building a single view on demand

The `build` management command can regenerate all pages for all views in the `BAKERY_VIEWS` settings variable. A *buildable model* can recreate all pages related to a single object. But can you rebuild all pages created by just one view? Yes, and all it takes is importing the view and invoking its `build_method`.

```
>>> from yourapp.views import DummyDetailView
>>> DummyDetailView().build_method()
```

A simple way to automate that kind of targeted build might be to create a custom management command and connect it to a cron job.

```
from django.core.management.base import BaseCommand, CommandError
from yourapp.views import DummyDetailView

class Command(BaseCommand):
    help = 'Rebuilds all pages created by the DummyDetailView'

    def handle(self, *args, **options):
        DummyDetailView().build_method()
```

Or, if you wanted to rebuild the view without deleting everything else in the existing build directory, you could pass it as an argument to the standard `build` command with instructions to skip everything else it normally does.

```
$ python manage.py build yourapp.views.DummyDetailView --keep-build-dir --skip-static␣
↪--skip-media
```

2.2.4 Enabling Amazon's accelerated uploads

If your bucket has enabled Amazon's S3 transfer acceleration service, you can configure bakery it use by overriding the default `AWS_S3_HOST` variable in `settings.py`.

```
AWS_S3_HOST = 's3-accelerate.amazonaws.com'
```

2.3 Buildable views

Django's class-based views are used to render HTML pages to flat files. Putting all the pieces together is a little tricky at first, particularly if you haven't studied the Django source code or lack experience working with Python classes in general. But if you figure it out, we think it's worth the trouble.

2.3.1 BuildableTemplateView

class BuildableTemplateView(*TemplateView*, *BuildableMixin*)

Renders and builds a simple template as a flat file. Extended from Django's generic TemplateView. The base class has a number of options not documented here you should consult.

build_path

The target location of the built file in the `BUILD_DIR`. `index.html` would place it at the built site's root. `foo/index.html` would place it inside a subdirectory. Required.

template_name

The name of the template you would like Django to render. Required.

build()

Writes the rendered template's HTML to a flat file. Only override this if you know what you're doing.

build_method

An alias to the `build` method used by the *management commands*

Example myapp/views.py

```
from bakery.views import BuildableTemplateView

class ExampleTemplateView(BuildableTemplateView):
    build_path = 'examples/index.html'
    template_name = 'examples.html'
```

2.3.2 BuildableListView

class BuildableListView(*ListView*, *BuildableMixin*)

Render and builds a page about a list of objects. Extended from Django's generic ListView. The base class has a number of options not documented here you should consult.

model

A Django database model where the list of objects can be drawn with a `Model.objects.all()` query. Optional. If you want to provide a more specific list, define the `queryset` attribute instead.

queryset

The list of objects that will be provided to the template. Can be any iterable of items, not just a Django queryset. Optional, but if this attribute is not defined the `model` attribute must be defined.

build_path

The target location of the flat file in the `BUILD_DIR`. Optional. The default is `index.html`, would place the flat file at the site's root. Defining it as `foo/index.html` would place the flat file inside a subdirectory.

template_name
The template you would like Django to render. You need to override this if you don't want to rely on the Django `ListView` defaults.

build_method
An alias to the `build_queryset` method used by the *management commands*

build_queryset()
Writes the rendered template's HTML to a flat file. Only override this if you know what you're doing.

Example myapp/views.py

```
from myapp.models import MyModel
from bakery.views import BuildableListView


class ExampleListView(BuildableListView):
    model = MyModel
    template_name = 'mymodel_list.html'


class DifferentExampleListView(BuildableListView):
    build_path = 'mymodel/index.html'
    queryset = MyModel.objects.filter(is_published=True)
    template_name = 'mymodel_list.html'
```

2.3.3 BuildableDetailView

class BuildableDetailView(*DetailView*, *BuildableMixin*)
Render and build a "detail" page about an object or a series of pages about a list of objects. Extended from Django's generic DetailView. The base class has a number of options not documented here you should consult.

model
A Django database model where the list of objects can be drawn with a `Model.objects.all()` query. Optional. If you want to provide a more specific list, define the `queryset` attribute instead.

queryset
The Django model queryset objects are to be looked up from. Optional, but if this attribute is not defined the `model` attribute must be defined.

template_name
The name of the template you would like Django to render. You need to override this if you don't want to rely on the default, which is `os.path.join(settings.BUILD_DIR, obj.get_absolute_url(), 'index.html')`.

get_build_path(*obj*)
Used to determine where to build the detail page. Override this if you would like your detail page at a different location. By default it will be built at `os.path.join(obj.get_url(), "index.html"`.

get_html(*obj*)
How to render the output for the provided object's page. If you choose to render using something other than a Django template, like HttpResponse for instance, you will want to override this. By default it uses the template object's default `render` method.

get_url(*obj*)
Returns the build directory, and therefore the URL, where the provided object's flat file should be placed. By default it is `obj.get_absolute_url()`, so simplify defining that on your model is enough.

build_method
An alias to the `build_queryset` method used by the *management commands*

build_object(*obj*)
Writes the rendered HTML for the template and the provided object to the build directory.

build_queryset()
Writes the rendered template's HTML for each object in the `queryset` or `model` to a flat file. Only override this if you know what you're doing.

unbuild_object(*obj*)
Deletes the directory where the provided object's flat files are stored.

Example myapp/models.py

```
from django.db import models
from bakery.models import BuildableModel


class MyModel(BuildableModel):
    detail_views = ('myapp.views.ExampleDetailView',)
    title = models.CharField(max_length=100)
    slug = models.SlugField(max_length=100)

    def get_absolute_url(self):
        """
        If you are going to publish a detail view for each object,
        one easy way to set the path where it will be built is to
        configure Django's standard get_absolute_url method.
        """
        return '/%s/' % self.slug
```

Example myapp/views.py

```
from myapp.models import MyModel
from bakery.views import BuildableDetailView


class ExampleDetailView(BuildableListView):
    queryset = MyModel.objects.filter(is_published=True)
    template_name = 'mymodel_detail.html'
```

2.3.4 BuildableArchiveIndexView

class BuildableArchiveIndexView(*ArchiveIndexView*, *BuildableMixin*)
Renders and builds a top-level index page showing the "latest" objects, by date. Extended from Django's generic ArchiveIndexView. The base class has a number of options not documented here you should consult.

model
A Django database model where the list of objects can be drawn with a `Model.objects.all()` query. Optional. If you want to provide a more specific list, define the `queryset` attribute instead.

queryset
The list of objects that will be provided to the template. Can be any iterable of items, not just a Django queryset. Optional, but if this attribute is not defined the `model` attribute must be defined.

build_path
The target location of the flat file in the `BUILD_DIR`. Optional. The default is `archive/index.html`, would place the flat file at the '/archive/' URL.

template_name
The template you would like Django to render. You need to override this if you don't want to rely on the Django default, which is `<model_name_lowercase>_archive.html`.

build_method
An alias to the `build_queryset` method used by the *management commands*

build_queryset()
Writes the rendered template's HTML to a flat file. Only override this if you know what you're doing.

Example myapp/views.py

```
from myapp.models import MyModel
from bakery.views import BuildableArchiveIndexView


class ExampleArchiveIndexView(BuildableArchiveIndexView):
    model = MyModel
    date_field = "pub_date"


class DifferentExampleArchiveIndexView(BuildableArchiveIndexView):
    build_path = 'my-archive-directory/index.html'
    queryset = MyModel.objects.filter(is_published=True)
    date_field = "pub_date"
    template_name = 'mymodel_list.html'
```

2.3.5 BuildableYearArchiveView

class BuildableYearArchiveView(*YearArchiveView*, *BuildableMixin*)
Renders and builds a yearly archive showing all available months (and, if you'd like, objects) in a given year. Extended from Django's generic YearArchiveView. The base class has a number of options not documented here you should consult.

model
A Django database model where the list of objects can be drawn with a `Model.objects.all()` query. Optional. If you want to provide a more specific list, define the `queryset` attribute instead.

queryset
The list of objects that will be provided to the template. Can be any iterable of items, not just a Django queryset. Optional, but if this attribute is not defined the `model` attribute must be defined.

template_name
The template you would like Django to render. You need to override this if you don't want to rely on the Django default, which is `<model_name_lowercase>_archive_year.html`.

get_build_path()
Used to determine where to build the detail page. Override this if you would like your detail page at a different location. By default it will be built at `os.path.join(obj.get_url(), "index.html"`.

get_url()
The URL at which the detail page should appear. By default it is /archive/ + the year in the generic view's `year_format` attribute. An example would be /archive/2016/

build_method
An alias to the `build_dated_queryset` method used by the *management commands*

build_dated_queryset()
Writes the rendered HTML for all publishable dates to the build directory.

build_year(*dt*)
Writes the rendered HTML for the provided year to the build directory.

unbuild_year(*dt*)
Deletes the directory where the provided year's flat files are stored.

Example myapp/views.py

```
from myapp.models import MyModel
from bakery.views import BuildableYearArchiveView


class ExampleArchiveYearView(BuildableYearArchiveView):
    model = MyModel
    date_field = "pub_date"
```

2.3.6 BuildableMonthArchiveView

class BuildableMonthArchiveView(*MonthArchiveView*, *BuildableMixin*)
Renders and builds a monthly archive showing all objects in a given month. Extended from Django's generic MonthArchiveView. The base class has a number of options not documented here you should consult.

model
A Django database model where the list of objects can be drawn with a `Model.objects.all()` query. Optional. If you want to provide a more specific list, define the `queryset` attribute instead.

queryset
The list of objects that will be provided to the template. Can be any iterable of items, not just a Django queryset. Optional, but if this attribute is not defined the `model` attribute must be defined.

template_name
The template you would like Django to render. You need to override this if you don't want to rely on the Django default, which is `<model_name_lowercase>_archive_month.html`.

get_build_path()
Used to determine where to build the detail page. Override this if you would like your detail page at a different location. By default it will be built at `os.path.join(obj.get_url(), "index.html"`.

get_url()
The URL at which the detail page should appear. By default it is /archive/ + the year in self.year_format + the month in self.month_format. An example would be /archive/2016/01/.

build_method
An alias to the `build_dated_queryset` method used by the *management commands*

build_dated_queryset()
Writes the rendered HTML for all publishable dates to the build directory.

build_month(*dt*)
Writes the rendered HTML for the provided month to the build directory.

unbuild_month(*dt*)
Deletes the directory where the provided month's flat files are stored.

Example myapp/views.py

```
from myapp.models import MyModel
from bakery.views import BuildableMonthArchiveView

```

(continues on next page)

(continued from previous page)

```
class ExampleMonthArchiveView(BuildableMonthArchiveView):
    model = MyModel
    date_field = "pub_date"
```

2.3.7 BuildableDayArchiveView

class BuildableDayArchiveView(*DayArchiveView*, *BuildableMixin*)

Renders and builds a day archive showing all objects in a given day. Extended from Django's generic DayArchiveView. The base class has a number of options not documented here you should consult.

model

A Django database model where the list of objects can be drawn with a `Model.objects.all()` query. Optional. If you want to provide a more specific list, define the `queryset` attribute instead.

queryset

The list of objects that will be provided to the template. Can be any iterable of items, not just a Django queryset. Optional, but if this attribute is not defined the `model` attribute must be defined.

template_name

The template you would like Django to render. You need to override this if you don't want to rely on the Django default, which is `<model_name_lowercase>_archive_day.html`.

get_build_path()

Used to determine where to build the detail page. Override this if you would like your detail page at a different location. By default it will be built at `os.path.join(obj.get_url(), "index.html"`.

get_url()

The URL at which the detail page should appear. By default it is /archive/ + the year in self.year_format + the month in self.month_format + the day in the self.day_format. An example would be /archive/2016/01/01/.

build_method

An alias to the `build_dated_queryset` method used by the *management commands*

build_dated_queryset()

Writes the rendered HTML for all publishable dates to the build directory.

build_day(*dt*)

Writes the rendered HTML for the provided day to the build directory.

unbuild_day(*dt*)

Deletes the directory where the provided day's flat files are stored.

Example myapp/views.py

```
from myapp.models import MyModel
from bakery.views import BuildableDayArchiveView


class ExampleDayArchiveView(BuildableDayArchiveView):
    model = MyModel
    date_field = "pub_date"
```

2.3.8 Buildable404View

class Buildable404View (*BuildableTemplateView*)

Renders and builds a simple 404 error page template as a flat file. Extended from the `BuildableTemplateView` above. The base class has a number of options not documented here you should consult.

All it does

```
from bakery.views import BuildableTemplateView


class Buildable404View(BuildableTemplateView):
    build_path = '404.html'
    template_name = '404.html'
```

2.3.9 BuildableRedirectView

class BuildableRedirectView (*RedirectView*, *BuildableMixin*)

Render and build a redirect. Extended from Django's generic RedirectView. The base class has a number of options not documented here you should consult.

build_path

The URL being requested, which will be published as a flatfile with a redirect away from it.

url

The URL where redirect will send the user. Operates in the same way as the standard generic RedirectView.

Example myapp/views.py

```
from bakery.views import BuildableRedirectView


class ExampleRedirectView(BuildableRedirectView):
    build_path = "mymodel/oldurl.html"
    url = '/mymodel/'
```

2.4 Buildable models

2.4.1 Models that build themselves

If your site publishes numerous pages built from a large database, the build-and-publish routine can take a long time to run. Sometimes that's acceptable, but if you're periodically making small updates to the site it can be frustrating to wait for the entire database to rebuild every time there's a minor edit.

We tackle this problem by hooking targeted build routines to our Django models. When an object is edited, the model is able to rebuild only those pages that object is connected to. We accomplish this with a `BuildableModel` class you can inherit. It works the same as a standard Django model, except that you are asked define a list of the detail views connected to each object.

BuildableModel

class BuildableModel (*models.Model*)

An abstract base model that creates an object that can builds out its own detail pages.

detail_views
An iterable containing paths to the views that are built using the object, which should inherit from *buildable class-based views*.

build()
Iterates through the views pointed to by `detail_views`, running each view's `build_object` method with `self`. Then calls `_build_extra()` and `_build_related()`.

unbuild()
Iterates through the views pointed to by `detail_views`, running each view's `unbuild_object` method with `self`. Then calls `_unbuild_extra()` and `_build_related()`.

_build_extra()
A place to include code that will build extra content related to the object that is not rendered by the `detail_views`, such a related image. Empty by default.

_build_related()
A place to include code that will build related content, such as an RSS feed, that does not require passing in the object to a view. Empty by default.

_unbuild_extra()
A place to include code that will remove extra content related to the object that is not rendered by the `detail_views`, like deleting a related image. Empty by default.

```
from django.db import models
from bakery.models import BuildableModel


class MyModel(BuildableModel):
    detail_views = ('myapp.views.ExampleDetailView',)
    title = models.CharField(max_length=100)
    slug = models.SlugField(max_length=100)
    description = models.TextField()
    is_published = models.BooleanField(default=False)

    def get_absolute_url(self):
        """
        If you are going to publish a detail view for each object,
        one easy way to set the path where it will be built is to
        configure Django's standard get_absolute_url method.
        """
        return '/%s/' % self.slug

    def _build_related(self):
        from myapp import views
        views.MySitemapView().build_queryset()
        views.MyRSSFeed().build_queryset()
```

2.4.2 Models that publish themselves

With a buildable model in place, you can take things a step further with the `AutoPublishingBuildableModel` so that a update posted to the database by an entrant using the Django admin can set into motion a small build that is then synced with your live site on Amazon S3.

At the Los Angeles Times Data Desk, we use that system to host applications with in-house Django administration panels that, for the entrant, walk and talk like a live website, but behind the scenes automatically figure out how to serve themselves on the Web as flat files. That's how a site like graphics.latimes.com is managed.

This is accomplished by handing off the build from the user's save request in the admin to a job server that does the work in the background. This prevents a user who makes a push-button save in the admin from having to wait for the full process to complete before receiving a response.

This is done by passing off build instructions to a Celery job server. **You need to install Celery and have it fully configured before this model will work.**

AutoPublishingBuildableModel

class AutoPublishingBuildableModel(*BuildableModel*)

Integrates with Celery tasks to automatically publish or unpublish objects when they are saved.

This is done using an override on the save method that inspects if the object ought to be published, republished or unpublished.

Requires an indicator of whether the object should been published or unpublished. By default it looks to a BooleanField called `is_published` for the answer, but other methods could be employed by overriding the `get_publication_status` method.

publication_status_field

The name of the field that this model will inspect to determine the object's publication status. By default it is `is_published`.

get_publication_status()

Returns a boolean (True or False) indicating whether the object is "live" and ought to be published or not.

Used to determine whether the save method should seek to publish, republish or unpublish the object when it is saved.

By default, it looks for a BooleanField with the name defined in the model's `publication_status_field`.

If your model uses a list of strings or other more complex means to indicate publication status you need to override this method and have it negotiate your object to return either True or False.

save(*publish=True*)

A custom save that uses Celery tasks to publish or unpublish the object where appropriate.

Save with keyword argument obj.save(publish=False) to skip the process.

delete(*unpublish=True*)

Triggers a task that will unpublish the object after it is deleted.

Save with keyword argument obj.delete(unpublish=False) to skip it.

```
from django.db import models
from bakery.models import AutoPublishingBuildableModel


class MyModel(AutoPublishingBuildableModel):
    detail_views = ('myapp.views.ExampleDetailView',)
    title = models.CharField(max_length=100)
    slug = models.SlugField(max_length=100)
    description = models.TextField()
    is_published = models.BooleanField(default=False)

    def get_absolute_url(self):
        """
        If you are going to publish a detail view for each object,
        one easy way to set the path where it will be built is to
```

(continues on next page)

(continued from previous page)

```
        configure Django's standard get_absolute_url method.
        """
        return '/%s/' % self.slug
```

2.5 Buildable feeds

You can build a RSS feed in much the same manner as *buildable class-based views*.

2.5.1 BuildableFeed

class BuildableFeed(*Feed*, *BuildableMixin*)

Extends the base Django Feed class to be buildable. Configure it in the same way you configure that plus our bakery options listed below.

build_path

The target location of the flat file in the `BUILD_DIR`. Optional. The default is `latest.xml`, would place the flat file at the site's root. Defining it as `foo/latest.xml` would place the flat file inside a subdirectory.

build_method

An alias to the `build_queryset` method used by the *management commands*

build_queryset()

Writes the rendered template's HTML to a flat file. Only override this if you know what you're doing.

Example myapp/feeds.py

```
from myapp.models import MyModel
from bakery.feeds import BuildableFeed


class ExampleRSSFeed(BuildableFeed):
    link = 'http://www.mysite.com/rss.xml'
    build_path = 'rss.xml'

    def items(self):
        return MyModel.objects.filter(is_published=True)
```

2.6 Settings variables

Configuration options for your `settings.py`.

2.6.1 ALLOW_BAKERY_AUTO_PUBLISHING

ALLOW_BAKERY_AUTO_PUBLISHING

Decides whether the *AutoPublishingBuildableModel* is allowed to run the *publish* management command as part of its background task. True by default.

```
# So if you are in your dev environment and want to prevent
# the task from publishing to s3, do this.
ALLOW_BAKERY_AUTO_PUBLISHING = False
```

2.6.2 BUILD_DIR

BUILD_DIR

The location on the filesystem where you want the flat files to be built.

```
BUILD_DIR = '/home/you/code/your-site/build/'

# I like something a little snappier like...
import os
BUILD_DIR = os.path.join(__file__, 'build')
```

2.6.3 BAKERY_FILESYSTEM

BAKERY_FILESYSTEM

Files are built using PyFilesystem, a module that provides a common interface to a variety of filesystem backends. The default setting is the OS filesystem backend that saves files to the local directory structure. If you don't set the variable, it will operates as follows:

```
BAKERY_FILESYSTEM = 'osfs:///'
```

Here's how you could change to an in-memory backend instead. The complete list of alternatives are documented here.

```
BAKERY_FILESYSTEM = 'mem://'
```

2.6.4 BAKERY_VIEWS

BAKERY_VIEWS

The list of views you want to be built out as flat files when the `build` *management command* is executed.

```
BAKERY_VIEWS = (
    'myapp.views.ExampleListView',
    'myapp.views.ExampleDetailView',
    'myapp.views.MyRSSView',
    'myapp.views.MySitemapView',
)
```

2.6.5 AWS_BUCKET_NAME

AWS_BUCKET_NAME

The name of the Amazon S3 "bucket" on the Internet were you want to publish the flat files in your local `BUILD_DIR`.

```
AWS_BUCKET_NAME = 'your-bucket'
```

2.6.6 AWS_ACCESS_KEY_ID

AWS_ACCESS_KEY_ID
A part of your secret Amazon Web Services credentials. Necessary to upload files to S3.

```
AWS_ACCESS_KEY_ID = 'your-key'
```

2.6.7 AWS_SECRET_ACCESS_KEY

AWS_SECRET_ACCESS_KEY
A part of your secret Amazon Web Services credentials. Necessary to upload files to S3.

```
AWS_SECRET_ACCESS_KEY = 'your-secret-key'
```

2.6.8 AWS_REGION

AWS_REGION
The name of the Amazon Web Services' region where the S3 bucket is stored. Results depend on the endpoint and region, but if you are not using the default `us-east-1` region you may need to set this variable.

```
AWS_REGION = 'us-west-2'
```

2.6.9 AWS_S3_ENDPOINT

AWS_S3_ENDPOINT
The URL to use when connecting with Amazon Web Services' S3 system. If the setting is not provided the boto package's default is used.

```
# Substitute in Amazon's accelerated upload service
AWS_S3_ENDPOINT = 'https://s3-accelerate.amazonaws.com'
# Specify the region of the bucket to work around bugs with S3 in certain version of␣
↪boto
AWS_S3_ENDPOINT = 'https://s3-%s.amazonaws.com' % AWS_REGION
```

2.6.10 BAKERY_GZIP

BAKERY_GZIP
Opt in to automatic gzipping of your files in the build method and addition of the required headers when deploying to Amazon S3. Defaults to `False`.

```
BAKERY_GZIP = True
```

2.6.11 GZIP_CONTENT_TYPES

GZIP_CONTENT_TYPES
A list of file mime types used to determine which files to add the 'Content-Encoding: gzip' metadata header when syncing to Amazon S3.

Defaults to include all 'text/css', 'text/html', 'application/javascript', 'application/x-javascript' and everything else recommended by the HTML5 boilerplate guide.

Only matters if you have set `BAKERY_GZIP` to `True`.

```
GZIP_CONTENT_TYPES = (
    "application/atom+xml",
    "application/javascript",
    "application/json",
    "application/ld+json",
    "application/manifest+json",
    "application/rdf+xml",
    "application/rss+xml",
    "application/schema+json",
    "application/vnd.geo+json",
    "application/vnd.ms-fontobject",
    "application/x-font-ttf",
    "application/x-javascript",
    "application/x-web-app-manifest+json",
    "application/xhtml+xml",
    "application/xml",
    "font/eot",
    "font/opentype",
    "image/bmp",
    "image/svg+xml",
    "image/vnd.microsoft.icon",
    "image/x-icon",
    "text/cache-manifest",
    "text/css",
    "text/html",
    "text/javascript",
    "text/plain",
    "text/vcard",
    "text/vnd.rim.location.xloc",
    "text/vtt",
    "text/x-component",
    "text/x-cross-domain-policy",
    "text/xml"
)
```

2.6.12 DEFAULT_ACL

DEFAULT_ACL

Set the access control level of the files uploaded. Defaults to 'public-read'

```
# defaults to 'public-read',
DEFAULT_ACL = 'public-read'
```

2.6.13 BAKERY_CACHE_CONTROL

BAKERY_CACHE_CONTROL

Set cache-control headers based on content type. Headers are set using the `max-age=` format so the passed values should be in seconds (`'text/html': 900` would result in a `Cache-Control: max-age=900` header for all `text/html` files). By default, none are set.

```
BAKERY_CACHE_CONTROL = {
    'text/html': 900,
```

(continues on next page)

(continued from previous page)

```
    'application/javascript': 86400
}
```

2.7 Management commands

Custom Django management commands for this library that help make things happen.

2.7.1 build

Bake out a site as flat files in the build directory, defined in settings.py by `BUILD_DIR`.

By default, this wipes the build directory if it exists, then builds any static files in the `STATIC_ROOT` to the `STATIC_URL`, any media files in the `MEDIA_ROOT` to the `MEDIA_URL` and all pages generated by the buildable views listed in `BAKERY_VIEWS`.

As a small bonus, files named `robots.txt` and `favicon.ico` will be placed at the build directory's root if discovered at the `STATIC_ROOT`.

Defaults can be modified with the following command options.

--build_dir <path>
: Specify the path of the build directory. Will use `settings.BUILD_DIR` by default.

--keep-build-dir
: Skip deleting and recreating the build directory before building files. By default the entire directory is wiped out.

--skip-static
: Skip collecting the static files when building.

--skip-media
: Skip collecting the media files when building.

```
$ python manage.py build
```

View names passed as arguments will override the `BAKERY_VIEWS` list.

```
$ python manage.py yourapp.views.DummyListView
```

2.7.2 buildserver

Starts a variation of Django's runserver designed to serve the static files you've built in the build directory.

```
$ python manage.py buildserver
```

2.7.3 publish

Syncs your Amazon S3 bucket to be identical to the local build directory. New files are uploaded, changed files are updated and absent files are deleted.

--aws-bucket-name <name>
: Specify the AWS bucket to sync with. Will use settings.AWS_BUCKET_NAME by default.

--build_dir <path>
Specify the path of the build directory. Will use settings.BUILD_DIR by default.

--force
Force a re-upload of all files in the build directory to the AWS bucket.

--dry-run
Provide output of what the command would perform, but without changing anything.

--no-delete
Keep files in S3, even if they do not exist in the build directory. The default behavior is to delete files in the bucket that are not in the build directory.

```
$ python manage.py publish
```

2.7.4 unbuild

Empties the build directory.

```
$ python manage.py unbuild
```

2.7.5 unpublish

Empties the Amazon S3 bucket defined in `settings.py`.

```
$ python manage.py unpublish
```

2.8 Changelog

2.8.1 0.12.6

- Refactored BuildableTemplateView to allow for using *reverse_lazy* to concoct the build path.

2.8.2 0.12.5

- Small logging improvement

2.8.3 0.12.4

- Moved fs config from the AppConfig's out of the ready method and set it as a base attribute on the class.

2.8.4 0.12.0

- Refactored the build methods to write to files using the PyFilesystem interface

2.8.5 0.11.1

- Skip gzipping of static files that are already gzipped.

2.8.6 0.11.0

- Django 2.0 testing and support.

2.8.7 0.10.5

- Added *get_view_instance* method to the *build* command to allow for more creative subclassing.

2.8.8 0.10.4

- Patched the `publish` command to calculate multipart md5 checksums for uploads large enough to trigger boto3's automatic multipart upload. This prevents large files from being improperly reuploaded during syncs.

2.8.9 0.10.3

- `AWS_REGION` setting now passed on to the s3 connection as an initialization option.

2.8.10 0.10.2

- Added a `--aws-bucket-prefix` option to the `publish` command. When specified, the local files will be synced with only those files in the bucket that have that prefix.

2.8.11 0.10.0

- Default pooling of file comparisons between published and local files for faster performance
- Option to opt-in to pooling of building of files locally for faster performance
- When `--force` and `--no-delete` options are both passed to publish command the s3 object list is not retrieved for faster performance

2.8.12 0.9.3

- Restored RedirectView boto code after upgrading it to boto3.

2.8.13 0.9.2

- Removed boto code from RedirectView until we can figure out a boto3 replacement.

2.8.14 0.9.1

- Added `S3_ENDPOINT_URL` for boto3 configuration and a fallback so we can continue to support the boto convention of `S3_AWS_HOST`

2.8.15 0.9.0

- Replaced `boto` dependency with `boto3` and refactored publish command to adjust
- More verbose logging of gzipped paths during build routine
- Reduced some logging in management commands when verbosity=0
- Added testing for Django 1.11

2.8.16 0.8.14

- Management command drops `six.print` for `self.output.write`
- Only strip first slash of urls with lstrip

2.8.17 0.8.13

- Fixed bug in `BuildableDayArchiveView` argument handling.

2.8.18 0.8.12

- Added `create_request` method to the base view mixin so there's a clearer method for overriding the creation of a `RequestFactory` when building views.

2.8.19 0.8.10

- Expanded default `GZIP_CONTENT_TYPES` to cover SVGs and everything else recommended by the HTML5 boilerplate guides.

2.8.20 0.8.9

- Removed `CommandError` exception handling in `build` command because errors should never pass silently, unless explicitly silenced.

2.8.21 0.8.8

- Django 1.10 support and testing

2.8.22 0.8.7

- `get_month` and `get_year` fix on the month archive view

2.8.23 0.8.6

- `get_year` fix on the year archive view.

2.8.24 0.8.5

- `get_absolute_url` bug fix on detail view.

2.8.25 0.8.3

- Added support for `AWS_S3_HOST` variable to override the default with connecting to S3 via boto.

2.8.26 0.8.2

- Upgraded to Django new style of management command options.

2.8.27 0.8.1

- Patch to allow for models to be imported with `django.contrib.contenttypes` being installed.

2.8.28 0.8.0

- Added new date-based archive views BuildableArchiveIndexView, BuildableYearArchiveView, BuildableMonthArchiveView, BuildableDayArchiveView
- get_url method on the BuildableDetailView now raises a ImproperlyConfigured error
- Refactored views into separate files

2.8.29 0.7.8

- Improved error handling and documentation of BuildableDetailView's `get_url` method.

2.8.30 0.7.7

- Patch provided backwards compatibility to a boto bug fix.

2.8.31 0.7.6

- Patched `set_kwargs` to override the key name of the slug when it is configured by the detail view's `slug_field` setting

2.8.32 0.7.5

- BAKERY_CACHE_CONTROL settings variable and support
- Better tests for publish and unpublish
- Delete operations in publish and unpublish command breaks keys into batches to avoid S3 errors on large sets

2.8.33 0.7.4

- Fixed content_type versus mimetype bug in the static views for Django 1.7 and 1.8
- A few other small Python 3 related bugs

2.8.34 0.7.3

- Added a `--no-delete` option to the `publish` management command.
- Fixed testing in Django 1.7

2.8.35 0.7.1

- Added `BuildableRedirectView`

2.8.36 0.6.4

- Added `BuildableFeed` for RSS support

2.8.37 0.6.3

- Changed `AutoPublishingBuildableModel` to commit to the database before triggering a task
- Changed celery tasks to accept primary keys instead of model objects

2.8.38 0.6.0

- An `AutoPublishingBuildableModel` that is able to use a Celery job queue to automatically build and publish objects when they are saved
- Refactored `build` management command to allow for its different tasks to be more easily overridden
- Added a `--keep-build-dir` option to the `build` command.

2.8.39 0.5.0

- Refactored the `publish` and `unpublish` management commands to use boto instead of s3cmd.
- `build` and `publish` management commands use file mimetypes instead of a regex on the filename to decide if a file will be gzipped.
- `publish` management command includes –force and –dry-run uploads to force an upload of all file, regardless of changes, and to print output without uploading files, respectively.
- `publish` management command now pools uploads to increase speed

2.8.40 0.4.2

- Added a `get_content` method to all of the buildable views to make it easier to build pages that don't require a template, like JSON outputs

2.8.41 0.4.1

- Bug fix with calculating Python version in the views in v0.4.0

2.8.42 0.4.0

- Added optional gzip support to build routine for Amazon S3 publishing (via @emamd)
- Mixin buildable view with common methods

2.8.43 0.3.0

- Python 3 support
- Unit tests
- Continuous integration test by Travis CI
- Coverage reporting by Coveralls.io
- PEP8 compliance
- PyFlakes compliance
- Refactored `buildserver` management command to work with latest versions of Django

2.8.44 0.2.0

- Numerous bug fixes

2.8.45 0.1.0

- Initial release

2.9 Credits

This application was written by Ben Welsh, Ken Schwencke and Armand Emamdjomeh at the Los Angeles Times Data Desk.

The ideas behind django-bakery were first presented at the 2012 conference of the National Institute for Computer-Assisted Reporting. The NICAR community is a constant source of challenge and inspiration. Many of our ideas, here and elsewhere, have been adapted from things the community has taught us.

CHAPTER 3

In the wild

- Hundreds of Los Angeles Times custom pages at latimes.com/projects and graphics.latimes.com
- The California Civic Data Coalition's data downloads
- A series of projects by the Austin American Statesman
- The Dallas Morning News' legislative tracker
- Newsday's police misconduct investigation
- Southern California Public Radio's water report tracker
- The Daily Californian's sexual misconduct case tracker
- The pretalx open-source conference management system
- The static-site extension to the Wagtail content management system

Have you used django bakery for something cool? Send a link to ben.welsh@gmail.com and we will add it to this list.

CHAPTER 4

Considering alternatives

If you are seeking to "bake" out a very simple site, maybe you don't have a database or only a single page, it is quicker to try Tarbell or Frozen-Flask, which don't require all the overhead of a full Django installation.

This library is better to suited for projects that require a database, want to take advantage of other Django features (like the administration panel) or require more complex views.

CHAPTER 5

Contributing

- Code repository: https://github.com/datadesk/django-bakery
- Issues: https://github.com/datadesk/django-bakery/issues
- Packaging: https://pypi.python.org/pypi/django-bakery
- Testing: https://travis-ci.org/datadesk/django-bakery
- Coverage: https://coveralls.io/r/datadesk/django-bakery

Index

Symbols

A

B

C

D

E

G

M

P

Q

S

T

U