Django-nonrel Documentation

Release 0

Tom Brander

Jul 12, 2017

Contents

This is is a compilation of the original documentation provided by Waldemar Kornewald.

Contents:

CHAPTER 1

Django-nonrel - NoSQL support for Django

Django-nonrel is an independent branch of Django that adds NoSQL database support to the ORM. The long-term goal is to add NoSQL support to the official Django release. Take a look at the documentation below for more information.

Why Django on NoSQL

We believe that development on non-relational databases is unnecessarily unproductive:

- you can't reuse code written for other non-relational DBs even if they're very similar
- you can't reuse code written for SQL DBs even if it could actually run unmodified on a non-relational DB
- you have to manually maintain indexes with hand-written code (denormalization, etc.)
- you have to manually write code for merging the results of multiple queries (JOINs, `select_related()`, etc.)
- some databases are coupled to a specific provider (App Engine, SimpleDB), so you're locked into their hosting solution and can't easily move away if at some point you need something that's impossible on their platform

How can we fix the situation? Basically, we need a powerful abstraction that automates all the manual indexing work and provides an advanced query API which simplifies common nonrel development patterns. The Django ORM is such an abstraction and it fits our goals pretty well. It even works with SQL and thus allows to reuse existing code written for SQL DBs.

Django-nonrel is in fact only a small patch to Django that fixes a few assumptions (e.g.: `AutoField` might not be an integer). The real work is done by django-dbindexer. The dbindexer will automatically take care of denormalization, JOINs, and other important features in the future too. Currently, it extends the NoSQL DB's native query language with support for filters like `__month` and `__icontains` and adds support for simple JOINs. If you want to stay up-to-date with our progress you should subscribe to the Django-nonrel blog for the latest news and tutorials.

This page hosts general information that is independent of the backend. You should also take a look at the 4 things to know for NoSQL Django coders blog post for practical examples.

The App Engine documentation is hosted on the djangoappengine project site. We've also published some information in the Django on App Engine blog post.

The MongoDB backend is introduced in the MongoDB backend for Django-nonrel blog post.

Backends for ElasticSearch and Cassandra are also in development.

Documentation

Differences to Django

Django should mostly behave as described in the Django documentation. You can run Django-nonrel in combined SQL and NoSQL multi-database setups without any problems. Whenever possible, we'll try to reuse existing Django APIs or at least provide our own APIs that abstract away the platform-specific API, so you can write portable, database-independent code. There might be a few exceptions and you should consult the respective backend documentation (e.g., djangoappengine) for more details. Here are a few general rules which can help you write better code:

Some backends (e.g., MongoDB) use a string instead of an integer for `AutoField`. If you want to be on the safe side always write code and urlpatterns that would work with both strings and integers. Note that `QuerySet` automatically converts strings to integers where necessary, so you don't need to deal with that in your code.

There is no integrity guarantee. When deleting a model instance the related objects will **not** be deleted. This had to be done because such a deletion process can take too much time. We might solve this in a later release.

JOINs don't work unless you use django-dbindexer, but even then they're very limited at the moment. Without dbindexer, queries can only be executed on one single model (filters can't span relationships like `user__username=...`).

You can't use Django's transactions API. If your particular DB supports a special kind of transaction (e.g., `run_in_transaction()` on App Engine) you have to use the platform-specific functions.

Not all DBs provide strong consistency. If you want your code to be fully portable you should assume an eventual consistency model. For example, recently App Engine introduced an eventually-consistent high-replication datastore which is now the preferred DB type because it provides very high availability. Strongly consistent operations should be implemented via `QuerySet.update()` instead of `Model.save()` (unless you use `run_in_transaction()` on App Engine, of course).

Workarounds

You can only edit users in the admin interface if you add "djangotoolbox" to your `INSTALLED_APPS`. Otherwise you'll get an exception about an unsupported query which requires JOINs.

Florian Hahn has also written an authentication backend which provides permission support on non-relational backends. You should use that backend if you want to use Django's permission system.

Writing a backend

The backend template provides a simple starting point with simple code samples, so you understand what each function does.

Sample projects

You can use allbuttonspressed or django-testapp as starting points to see what a nonrel-compatible project looks like.

Internals

Django-nonrel is based on Django 1.3. The modifications to Django are minimal (maybe less than 100 lines). Backends can override the `save()` process, so no distinction between `INSERT` and `UPDATE` is made and so there is no unnecessary `exists()` check in `save()`. Also, deletion of related objects is disabled on NoSQL because it conflicts with transaction support at least on App Engine. These are the most important changes we've made. The actual DB abstraction is handled by django-dbindexer and it will stay a separate component even after NoSQL support is added to the official Django release.

Contributors

- Alberto Paro
- Andi Albrecht
- Flavio Percoco Premoli
- Florian Hahn
- Jonas Haag
- Kyle Finley
- George Karpenkov
- Mariusz Kryński
- Matt Bierner
- Thomas Wanschik
- Waldemar Kornewald

CHAPTER 2

All Buttons Pressed Blog Sample Project

This is the the website you're currently looking at. All Buttons Pressed provides a simple "CMS" with support for multiple independent blogs. It's compatible with Django-nonrel and the normal SQL-based Django, so you can use it with App Engine, MongoDB, MySQL, PostgreSQL, sqlite, and all other backends that work with either Django or Django-nonrel. Actually the project was started to demonstrate what's possible with Django-nonrel and to give you an impression of how to write code for non-relational databases.

Documentation

Note: Some parts of the code are explained in the 4 things to know for NoSQL Django coders blog post.

Before you start you'll need to install Sass and Compass. Then you can just download the zip file. Make sure that your App Engine SDK is at least version 1.5.0 prerelease.

Alternatively, if you want to clone the latest code you'll have to also install django-mediagenerator. By default, the settings file is configured to use App Engine, so you'll also need djangoappengine and all of its dependencies (Django-nonrel, djangotoolbox, django-dbindexer, and django-autoload). In contrast, on SQL databases you won't need any of those dependencies except for django-mediagenerator.

First, create an admin user with `manage.py createsupeuser`. Then, run `manage.py runserver` and go to http://localhost:8000/admin/ and create a few pages and posts. Otherwise you'll only see a 404 error page.

All Buttons Pressed has a concept called "Block". Blocks can be created and edited via the admin UI. The sidebar's content is defined via a block called "sidebar".

The menu is defined via a "menu" block where each menu item is in the format `<label> <url>`:

```
Some label     /url
Other label    http://www.other.url/
```

CHAPTER 3

djangoappengine - Django App Engine backends (DB, email, etc.)

Djangoappengine contains all App Engine backends for Django-nonrel, e.g. the database and email backends. In addition we provide a testapp which contains minimal settings for running Django-nonrel on App Engine. Use it as a starting point if you want to use App Engine as your database for Django-nonrel. We've also published some details in the Django on App Engine blog post.

Take a look at the documentation below and subscribe to our Django-nonrel blog for the latest updates.

Installation

Make sure you've installed the App Engine SDK. On Windows simply use the default installation path. On Linux you can put it in /usr/local/google_appengine. On MacOS it should work if you put it in your Applications folder. Alternatively, on all systems you can add the google_appengine folder to your PATH (not PYTHONPATH) environment variable.

Download the following zip files:

- django-nonrel (or clone it)
- djangoappengine (or clone it)
- djangotoolbox (or clone it)
- django-autoload (or clone it)
- django-dbindexer (or clone it)
- django-testapp (or clone it)

Unzip everything.

The django-testapp folder contains a sample project to get you started. If you want to start a new project or port an existing Django project you can just copy all ".py" and ".yaml" files from the root folder into your project and adapt settings.py and app.yaml to your needs.

Copy the following folders into your project (e.g., django-testapp):

- django-nonrel/django => `<project>`/django
- djangotoolbox/djangotoolbox => `<project>`/djangotoolbox
- django-autoload/autoload => `<project>`/autoload
- django-dbindexer/dbindexer => `<project>`/dbindexer
- djangoappengine => `<project>`/djangoappengine

That's it. Your project structure should look like this:

- <project>/django
- <project>/djangotoolbox
- <project>/autoload
- <project>/dbindexer
- <project>/djangoappengine

Alternatively, you can of course clone the respective repositories and create symbolic links instead of copying the folders to your project. That might be easier if you have a lot of projects and don't want to update each one manually.

Management commands

You can directly use Django's manage.py commands. For example, run `manage.py createsuperuser` to create a new admin user and `manage.py runserver` to start the development server.

Important: Don't use dev_appserver.py directly. This won't work as expected because `manage.py runserver` uses a customized dev_appserver.py configuration. Also, never run `manage.py runserver` together with other management commands at the same time. The changes won't take effect. That's an App Engine SDK limitation which might get fixed in a later release.

With djangoappengine you get a few extra manage.py commands:

- `manage.py remote` allows you to execute a command on the production database (e.g., `manage.py remote shell` or `manage.py remote createsuperuser`)
- `manage.py deploy` uploads your project to App Engine (use this instead of `appcfg.py update`)

Note that you can only use `manage.py remote` if your app is deployed and if you have enabled authentication via the Google Accounts API in your app settings in the App Engine Dashboard. Also, if you use a custom app.yaml you have to make sure that it contains the remote_api handler.

Supported and unsupported features

Field types

All Django field types are fully supported except for the following:

- `ImageField`
- `ManyToManyField`

The following Django field options have no effect on App Engine:

- `unique`
- `unique_for_date`

- `unique_for_month`
- `unique_for_year`

Additionally djangotoolbox provides non-Django field types in `djangotoolbox.fields` which you can use on App Engine or other non-relational databases. These are

- `ListField`
- `BlobField`

The following App Engine properties can be emulated by using a `CharField` in Django-nonrel:

- `CategoryProperty`
- `LinkProperty`
- `EmailProperty`
- `IMProperty`
- `PhoneNumberProperty`
- `PostalAddressProperty`

QuerySet methods

You can use the following field lookup types on all Fields except on `TextField` (unless you use *indexes*) and `BlobField`

- `__exact` equal to (the default)
- `__lt` less than
- `__lte` less than or equal to
- `__gt` greater than
- `__gte` greater than or equal to
- `__in` (up to 500 values on primary keys and 30 on other fields)
- `__range` inclusive on both boundaries
- `__startswith` needs a composite index if combined with other filters
- `__year`
- `__isnull` requires django-dbindexer to work correctly on `ForeignKey` (you don't have to define any indexes for this to work)

Using django-dbindexer all remaining lookup types will automatically work too!

Additionally, you can use

- `QuerySet.exclude()`
- `Queryset.values()` (only efficient on primary keys)
- `Q`-objects
- `QuerySet.count()`
- `QuerySet.reverse()`
- ...

In all cases you have to keep general App Engine restrictions in mind.

Model inheritance only works with abstract base classes:

```
class MyModel(models.Model):
    # ... fields ...
    class Meta:
        abstract = True # important!

class ChildModel(MyModel):
    # works
```

In contrast, multi-table inheritance (i.e. inheritance from non-abstract models) will result in query errors. That's because multi-table inheritance, as the name implies, creates separate tables for each model in the inheritance hierarchy, so it requires JOINs to merge the results. This is not the same as multiple inheritance which is supported as long as you use abstract parent models.

Many advanced Django features are not supported at the moment. A few of them are:

- JOINs (with django-dbindexer simple JOINs will work)
- many-to-many relations
- aggregates
- transactions (but you can use `run_in_transaction()` from App Engine's SDK)
- `QuerySet.select_related()`

Other

Additionally, the following features from App Engine are not supported:

- entity groups (we don't yet have a `GAEPKField`, but it should be trivial to add)
- batch puts (it's technically possible, but nobody found the time/need to implement it, yet)

Indexes

It's possible to specify which fields should be indexed and which not. This also includes the possibility to convert a `TextField` into an indexed field like `CharField`. You can read more about this feature in our blog post Managing per-field indexes on App Engine.

Email handling

You can (and should) use Django's mail API instead of App Engine's mail API. The App Engine email backend is already enabled in the default settings (`from djangoappengine.settings_base import *`). By default, emails will be deferred to a background task on the production server.

Cache API

You can (and should) use Django's cache API instead of App Engine's memcache module. The memcache backend is already enabled in the default settings.

Sessions

You can use Django's session API in your code. The `cached_db` session backend is already enabled in the default settings.

Authentication

You can (and probably should) use `django.contrib.auth` directly in your code. We don't recommend to use App Engine's Google Accounts API. This will lock you into App Engine unnecessarily. Use Django's auth API, instead. If you want to support Google Accounts you can do so via OpenID. Django has several apps which provide OpenID support via Django's auth API. This also allows you to support Yahoo and other login options in the future and you're independent of App Engine. Take a look at Google OpenID Sample Store to see an example of what OpenID login for Google Accounts looks like.

Note that username uniqueness is only checked at the form level (and by Django's model validation API if you explicitly use that). Since App Engine doesn't support uniqueness constraints at the DB level it's possible, though very unlikely, that two users register the same username at exactly the same time. Your registration confirmation/activation mechanism (i.e., user receives mail to activate his account) must handle such cases correctly. For example, the activation model could store the username as its primary key, so you can be sure that only one of the created users is activated.

File uploads/downloads

See django-filetransfers for an abstract file upload/download API for `FileField` which works with the Blobstore and X-Sendfile and other solutions. The required backends for the App Engine Blobstore are already enabled in the default settings.

Background tasks

Contributors: We've started an experimental API for abstracting background tasks, so the same code can work with App Engine and Celery and others. Please help us finish and improve the API here: https://bitbucket.org/wkornewald/django-defer

Make sure that your `app.yaml` specifies the correct `deferred` handler. It should be:

```
- url: /_ah/queue/deferred
  script: djangoappengine/deferred/handler.py
  login: admin
```

This custom handler initializes `djangoappengine` before it passes the request to App Engine's internal `deferred` handler.

dbindexer index definitions

By default, djangoappengine installs `__iexact` indexes on `User.username` and `User.email`.

High-replication datastore settings

In order to use `manage.py remote` with the high-replication datastore you need to add the following to the top of your `settings.py`:

```
from djangoappengine.settings_base import *
DATABASES['default']['HIGH_REPLICATION'] = True
```

App Engine for Business

In order to use `manage.py remote` with the `googleplex.com` domain you need to add the following to the top of your `settings.py`:

```
from djangoappengine.settings_base import *
DATABASES['default']['DOMAIN'] = 'googleplex.com'
```

Checking whether you're on the production server

```
from djangoappengine.utils import on_production_server, have_appserver
```

When you're running on the production server `on_production_server` is `True`. When you're running either the development or production server `have_appserver` is `True` and for any other `manage.py` command it's `False`.

Zip packages

Important: Your instances will load slower when using zip packages because zipped Python files are not precompiled. Also, i18n doesn't work with zip packages. Zipping should only be a **last resort**! If you hit the 3000 files limit you should better try to reduce the number of files by, e.g., deleting unused packages from Django's "contrib" folder. Only when **nothing** (!) else works you should consider zip packages.

Since you can't upload more than 3000 files on App Engine you sometimes have to create zipped packages. Luckily, djangoappengine can help you with integrating those zip packages. Simply create a "zip-packages" directory in your project folder and move your zip packages there. They'll automatically get added to `sys.path`.

In order to create a zip package simply select a Python package (e.g., a Django app) and zip it. However, keep in mind that only Python modules can be loaded transparently from such a zip file. You can't easily access templates and JavaScript files from a zip package, for example. In order to be able to access the templates you should move the templates into your global "templates" folder within your project before zipping the Python package.

Contribute

If you want to help with implementing a missing feature or improving something please fork the source and send a pull request via BitBucket or a patch to the discussion group.

CHAPTER 4

django-autoload

django-autoload is a reuseable django app which allows for correct initialization of your django project by ensuring the loading of signal handlers or indexes before any request is being processed.

Installation

1. **Add django-autoload/autoload to settings.INSTALLED_APPS**

```
INSTALLED_APPS = (
    'autoload',
)
```

2. **Add the `autoload.middleware.AutoloadMiddleware` before any other middelware**

```
MIDDLEWARE_CLASSES = ('autoload.middleware.AutoloadMiddleware', ) + \
                     MIDDLEWARE_CLASSES
```

How does django-autoload ensures correct initialization of my project?

django-autoload provides two mechanisms to allow for correct initializations:

1. The `autoload.middleware.AutoloadMiddleware` middleware will load all `models.py` from settings.INSTALLED_APPS as soon as the first request is being processed. This ensures the registration of signal handlers for example.
2. django-autoload provides a site configuration module loading mechanism. Therefore, you have to create a site configuration module. The module name has to be specified in your settings:

```
# settings.py:
AUTOLOAD_SITECONF = 'indexes'
```

Now, django-autoload will load the module `indexes.py` before any request is being processed. An example module could look like this:

```
# indexes.py:
from dbindexer import autodiscover
autodiscover()
```

This will ensure the registration of all database indexes specified in your project.

autoload.autodiscover(module_name)

Some apps like django-dbindexer or nonrel-search provide an autodiscover-mechanism which tries to import index definitions from all apps in settings.INSTALLED_APPS. Since the autodiscover-mechanism is so common django-autoload provides an `autodiscover` function for these apps.

`autodiscover` will search for `[module_name].py` in all `settings.INSTALLED_APPS` and load them. django-dbindexer's autodiscover function just looks like this for example:

```
def autodiscover():
    from autoload import autodiscover as auto_discover
    auto_discover('dbindexes')
```

CHAPTER 5

Django-dbindexer

With django-dbindexer you can use SQL features on NoSQL databases and abstract the differences between NoSQL databases. For example, if your database doesn't support case-insensitive queries (`iexact`, `istartswith`, etc.) you can just tell the dbindexer which models and fields should support these queries and it'll take care of maintaining the required indexes for you. It's similar for JOINs. Tell the dbindexer that you would like to use in-memory JOINs for a specific query for example and the dbindexer will make it possible. Magically, previously unsupported queries will just work. Currently, this project is in an early development stage. The long-term plan is to support more complex JOINs and at least some simple aggregates, possibly even much more.

Tutorials

- Getting started: Get SQL features on NoSQL with django-dbindexer
- JOINs for NoSQL databases via django-dbindexer - First steps

Documentation

Dependencies: djangotoolbox, django-autoload

Installation

For installation see Get SQL features on NoSQL with django-dbindexer

How does django-dbindexer make unsupported field lookup types work?

For each filter you want to use on a field for a given model, django-dbindexer adds an additional field to that model. For example, if you want to use the `contains` filter on a `CharField` you have to add the following index definition:

```
register_index(MyModel, {'name': 'contains'})
```

django-dbindexer will then store an additional `ListField` called 'idxf_<char_field_name>_l_contains' on `MyModel`. When saving an entity, django-dbindexer will fill the `ListField` with all substrings of the `CharField`'s reversed content i.e. if `CharField` stores `'Jiraiya'` then the `ListField` stores `['J', 'iJ', 'riJ', 'ariJ' ..., 'ayiariJ']`. When querying on that `CharField` using `contains`, django-dbindexer delegates this filter using `startswith` on the `ListField` with the reversed query string i.e. `filter(<char_field_name>__contains='ira')` => `filter('idxf_<char_field_name>_l_contains'__startswith='ari')` which matches the content of the list and gives back the correct result set. On App Engine `startswith` gets converted to ">=" and "<" filters for example.

In the following is listed which fields will be added for a specific filter/lookup type:

- `__iexact` using an additional `CharField` and a `__exact` query
- `__istartswith` creates an additional `CharField`. Uses a `__startswith` query
- `__endswith` using an additional `CharField` and a `__startswith` query
- `__iendswith` using an additional `CharField` and a `__startswith` query
- `__year` using an additional `IntegerField``and a ``__exact` query
- `__month` using an additional `IntegerField` and a `__exact` query
- `__day` using an additional `IntegerField` and a `__exact` query
- `__week_day` using an additional `IntegerField` and a `__exact` query
- `__contains` using an additional `ListField` and a `__startswith` query
- `__icontains` using an additional `ListField` and a `__startswith` query
- `__regex` using an additional `ListField` and a `__exact` query
- `__iregex` using an additional `ListField` and a `__exact` query

For App Engine users using djangoappengine this means that you can use all django field lookup types for example.

MongoDB users using django-mongodb-engine can benefit from this because case-insensitive filters can be handled as efficient case-sensitive filters for example.

For regex filters you have to specify which regex filter you would like to execute:

```
register_index(MyModel, {'name': ('iexact', re.compile('\/\*.*?\*\/', re.I)})
```

This will allow you to use the following filter:

```
MyModel.objects.all().filter(name__iregex='\/\*.*?\*\/')
```

Backend system

django-dbindexer uses backends to resolve lookups. You can specify which backends to use via DBINDEXER_BACKENDS

```
# settings.py:

DBINDEXER_BACKENDS = (
    'dbindexer.backends.BaseResolver',
    'dbindexer.backends.InMemoryJOINResolver',
)
```

The BaseResolver is responsible for resolving lookups like __iexact or __regex for example. The InMemoryJOINResolver is used to resolve JOINs in-memory. The ConstantFieldJOINResolver uses denormalization in order to resolve JOINs. For more information see JOINs via denormalization for NoSQL coders, Part 1 is then done automatically by the ConstantFieldJOINResolver for you. :)

Loading indexes

First of all, you need to install django-autoload. Then you have to create a site configuration module which loads the index definitions. The module name has to be specified in the settings:

```
# settings.py:
AUTOLOAD_SITECONF = 'dbindexes'
```

Now, there are two ways to load database index definitions in the AUTOLOAD_SITECONF module: auto-detection or manual listing of modules.

Note: by default AUTOLOAD_SITECONF is set to your ROOT_URLCONF.

dbindexer.autodiscover

autodiscover will search for dbindexes.py in all INSTALLED_APPS and load them. It's like in django's admin interface. Your AUTOLOAD_SITECONF module would look like this:

```
# dbindexes.py:
import dbindexer
dbindexer.autodiscover()
```

Manual imports

Alternatively, you can import the desired index definition modules directly:

```
# dbindexes.py:
import myapp.dbindexes
import otherapp.dbindexes
```

CHAPTER 6

django-filetransfers

With django-filetransfers you can write reusable Django apps that handle uploads and downloads in an abstract way. Django's own file upload and storage API alone is too limited because (1) it doesn't provide a mechanism for file downloads and (2) it can only handle direct uploads which eat a lot of resources and aren't compatible with cloud services like the App Engine Blobstore or asynchronous Amazon S3 uploads (where the file isn't piped through Django, but sent directly to S3). This is where django-filetransfers comes in. You can continue to use Django's `FileField` and `ModelForm` in your apps. You just need to add a few very simple API calls to your file handling views and templates and select a django-filetransfers backend via your settings.py. With this you can transparently support cloud services for file hosting or even the X-Sendfile mechanism.

Installation

You can install the package via `setup.py install` or by copying or linking the "filetransfers" folder to your project (App Engine developers have to use the copy/link method). Then, add "filetransfers" to your `INSTALLED_APPS`.

Note for App Engine users: All nrequired backends are already enabled in the default settings. You don't need any special configuration. In order to use the Blobstore on the App Engine production server you have to enable billing. Otherwise, the Blobstore API is disabled.

Model and form

In the following we'll use this model and form:

```
class UploadModel(models.Model):
    file = models.FileField(upload_to='uploads/%Y/%m/%d/%H/%M/%S/')

class UploadForm(forms.ModelForm):
    class Meta:
        model = UploadModel
```

The `upload_to` parameter for `FileField` defines the target folder for file uploads (here, we add the date).

Note for App Engine users: When accessing a file object from `UploadedModel` you can get the file's `BlobInfo` object via `uploadedmodel.file.blobstore_info`. Use this to e.g. convert uploaded images via the Image API.

Handling uploads

File uploads are handled with the `prepare_upload()` function which takes the request and the URL of the upload view and returns a tuple with a generated upload URL and extra POST data for the upload. The extra POST data is just a `dict`, so you can pass it to your JavaScript code if needed. This is an example upload view:

```
from filetransfers.api import prepare_upload

def upload_handler(request):
    view_url = reverse('upload.views.upload_handler')
    if request.method == 'POST':
        form = UploadForm(request.POST, request.FILES)
        form.save()
        return HttpResponseRedirect(view_url)

    upload_url, upload_data = prepare_upload(request, view_url)
    form = UploadForm()
    return direct_to_template(request, 'upload/upload.html',
        {'form': form, 'upload_url': upload_url, 'upload_data': upload_data})
```

Note that it's important that you send a redirect after an upload. Otherwise, some file hosting services won't work correctly.

Now, you have to use the generated upload URL and the upload's extra POST data in the template:

```
{% load filetransfers %}
<form action="{{ upload_url }}" method="POST" enctype="multipart/form-data">
  {% csrf_token %}
  {% render_upload_data upload_data %}
  <table>{{ form }}</table>
  <input type="submit" value="Upload" />
</form>
```

Here we use the `{% render_upload_data %}` tag which generates `<input type="hidden" />` fields for the extra POST data.

Security and permissions

By default, uploads are assumed to have a publicly accessible URL if that's supported by the backend. You can tell the backend to mark the upload as private via `prepare_upload(..., private=True)`. If the backend has no control over the permissions (e.g., because it's your task to configure the web server correctly and not make private files publicly accessible) the `private=True` argument might just be ignored.

Asynchronous backends (like async S3 or even Blobstore) have to take special care of preventing faked uploads. After a successful upload to the actual server these backends have to generate a separate request which contains the POST data and a file ID identifying the uploaded file (the Blobstore automatically sends the blob key and async S3 would send the file and bucket name). The problem here is that a user can manually generate a request which matches the ID

of some other user's private file, thus getting access to that file because it's now fake-uploaded to his private files, too. In order to prevent this asynchronous backends have to guarantee that no file ID is used twice for an upload.

Handling downloads

Since the actual download permissions can be out of the backend's control the download solution consists of two layers.

The `serve_file()` function primarily takes care of private file downloads, but in some configurations it might also have to take care of public downloads because the file hosting solution doesn't provide publicly accessible URLs (e.g., App Engine Blobstore). This means that you should also use that function as a fallback even if you only have public downloads. The function takes two required arguments: the request and the Django `File` object that should be served (e.g. from `FileField`):

```
from filetransfers.api import serve_file

def download_handler(request, pk):
    upload = get_object_or_404(UploadModel, pk=pk)
    return serve_file(request, upload.file)
```

The `public_download_url` function, which is also available as a template filter, returns a file's publicly accessible URL if that's supported by the backend. Otherwise it returns `None`.

Important: Use `public_download_url` only for files that should be publicly accessible. Otherwise you should only use `serve_file()`, so you can check permissions before approving the download.

A complete solution for public downloads which falls back to `serve_file()` would look like this in a template for an instance of `UploadModel` called `upload`:

```
{% load filetransfers %}
{% url upload.views.download_handler pk=upload.pk as fallback_url %}
<a href="{% firstof upload.file|public_download_url fallback_url %}">Download</a>
```

The second line stores the `serve_file()` fallback URL in a variable. In the third line we then use the `public_download_url` template filter in order to get the file's publicly accessible URL. If that returns `None` the `{% firstof %}` template tag returns the second argument which is our fallback URL. Otherwise the public download URL is used.

Configuration

There are three backend types which are supported by django-filetransfers: one for uploads, one for downloads via `serve_file()`, and one for public downloads. You can specify the backends in your settings.py:

```
PREPARE_UPLOAD_BACKEND = 'filetransfers.backends.default.prepare_upload'
SERVE_FILE_BACKEND = 'filetransfers.backends.default.serve_file'
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.default.public_download_url'
```

The default upload backend simply returns the URL unmodified. The default download backend transfers the file in chunks via Django, so it's definitely not the most efficient mechanism, but it uses only a small amount of memory (important for large files) and requires less resources than passing a file object directly to the response. The default public downloads backend simply returns `None`. This default configuration should work with practically all servers, but it's not the most efficient solution. Please take a look at the backends which are shipped with django-filetransfers to see if something fits your solution better.

Private download backends

xsendfile.serve_file

Many web servers (at least Apache, Lighttpd, and nginx) provide an "X-Sendfile" module which allows for handing off the actual file transfer to the web server. This is much more efficient than the default download backend, so you should install the required module for your web server and then configure the xsendfile download backend in your settings.py:

```
SERVE_FILE_BACKEND = 'filetransfers.backends.xsendfile.serve_file'
```

url.serve_file

We also provide a backend which simply redirects to `file.url`. You have to make sure that `file.url` actually generates a private download URL, though. This backend should work with the Amazon S3 and similar storage backends from the django-storages project. Just add the following to your settings.py:

```
SERVE_FILE_BACKEND = 'filetransfers.backends.url.serve_file'
```

Public download backends

url.public_download_url

If `file.url` points to a public download URL you can use this backend:

```
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.url.public_download_url'
```

base_url.public_download_url

Alternatively, there's also a simple backend that merely points to a different URL. You just need to specify a base URL and the backend appends `file.name` to that base URL.

```
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.base_url.public_download_url'
PUBLIC_DOWNLOADS_URL_BASE = '/downloads/'
```

Upload backends

delegate.prepare_upload

This backend delegates the upload to some other backend based on `private=True` or `private=False`. This way you can, for instance, use the App Engine Blobstore for private files and Amazon S3 for public files:

```
# Configure "delegate" backend
PREPARE_UPLOAD_BACKEND = 'filetransfers.backends.delegate.prepare_upload'
PRIVATE_PREPARE_UPLOAD_BACKEND = 'djangoappengine.storage.prepare_upload'
PUBLIC_PREPARE_UPLOAD_BACKEND = 's3backend.prepare_upload'
```

```
# Use S3 for public_download_url and Blobstore for serve_file
SERVE_FILE_BACKEND = 'djangoappengine.storage.serve_file'
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.base_url.public_download_url'
PUBLIC_DOWNLOADS_URL_BASE = 'http://s3.amazonaws.com/my-public-bucket/'
```

Reference: `filetransfers.api` module

prepare_upload(request, url, private=False, backend=None)

Returns a tuple with a target URL for the upload form and a `dict` with additional POST data for the upload request.

Required arguments:

- `request`: The view's request.
- `url`: The target URL where the files should be sent to.

Optional arguments:

- `private`: If `False` the backend will try to make the upload publicly accessible, so it can be served via the `public_download_url` template filter. If `True` the backend will try to make the upload non-accessible to the public, so it can only be served via `serve_file()`.
- `backend`: If defined, you can override the backend specified in settings.py.

serve_file(request, file, backend=None, save_as=False, content_type=None)

Serves a file to the browser. This is used either for checking permissions before approving a downoad or as a fallback if the backend doesn't support publicly accessible URLs. So, you always have to provide a view that uses this function.

Required arguments:

- `request`: The view's request.
- `file`: The `File` object (e.g. from `FileField`) that should be served.

Optional arguments:

- `save_as`: Forces the browser to save the file instead of displaying it (useful for PDF documents, for example). If this is True the file object's `name` attribute will be used as the file name in the download dialog. Alternatively, you can pass a string to override the file name. The default is to let the browser decide how to handle the download.
- `content_type`: Overrides the file's content type in the response. By default the content type will be detected via `mimetypes.guess_type()` using `file.name`.
- `backend`: If defined, you can override the backend specified in settings.py.

public_download_url(file, backend=None)

Tries to generate a publicly accessible URL for the given file. Returns `None` if no URL could be generated. The same function is available as a template filter.

Required arguments:

- `file`: The `File` object (e.g. from `FileField`) that should be served.

Optional arguments:

- `backend`: If defined, you can override the backend specified in settings.py.

Reference: `filetransfers` template library

{% render_upload_data upload_data %}

Renders `<input type="hidden" ... />` fields for the extra POST data (`upload_data`) as returned by `prepare_upload()`.

public_download_url

This template filter does the same as the `public_upload_url()` function in the `filetransfers.api` module: It returns a publicly accessible URL for the given file or `None` if it no such URL exists.

It takes the `File` object (e.g. from `FileField`) that should be served and optionally a second parameter to override the backend specified in settings.py.

CHAPTER 7

Django Toolbox

Small set of useful Django tools. Goals: 1) be usable with non-relational Django backends 2) load no unnecessary code (faster instance startups) 3) provide good coding conventions and tools which have a real impact on your code (no “matter of taste” utilities).

We’ll add some documentation, soon.

CHAPTER 8

HTML5 offline manifests with django-mediagenerator

This is actually part 3 of our django-mediagenerator Python canvas app series (see part 1 and part 2), but since it has nothing to do with client-side Python we name it differently. In this part you'll see how to make your web app load without an Internet connection. HTML5 supports offline web apps through manifest files.

Manifest files

First here's some background, so you know what a manifest file is. A manifest file is really simple. In its most basic form it lists the URLs of the files that should be cached. Here's an `example.manifest`:

```
CACHE MANIFEST
/media/main.css
/media/main.js
```

The first line is always `CACHE MANIFEST`. The next lines can list the files that should be cached. In this case we've added the `main.css` and `main.js` bundles. Additionally, the main HTML file which includes the manifest is cached, automatically. You can include the manifest in the `<html>` tag:

```
<html manifest="example.manifest">
```

When the browser sees this it loads the manifest and adds the current HTML and manifest file and all files listed in the manifest to the cache. The next time you visit the page the browser will try to load the manifest file from your server and compare it to the cached version. If the content of the manifest file hasn't changed the browser just loads all files from the cache. If the content of the manifest file has changed the browser refreshes its cache.

This is important, so I repeat it: The browser updates its cache only when the **content** of the **manifest** file is modified. Changes to your JavaScript, CSS, and image files will go unnoticed if the manifest file is not changed! That's exactly where things become annoying. Imagine you've changed the `main.js` file. Now you have to change your manifest file, too. One possibility is to add a comment to their manifest file which represents the current version number:

```
CACHE MANIFEST
# version 2
```

```
/media/main.css
/media/main.js
```

Whenever you change something in your JS or CSS or image files you have to increment the version number, manually. That's not really nice.

django-mediagenerator to the rescue

This is where the media generator comes in. It automatically modifies the manifest file whenever your media files are changed. Since media files are versioned automatically by django-mediagenerator the version hash in the file name serves as a natural and automatic solution to our problem. With the media generator a manifest file could look like this:

```
CACHE MANIFEST
/media/main-bf1e7dfbd511baf660e57a1f36048750f1ee660f.css
/media/main-fb16702a27fc6c8073aa4df0b0b5b3dd8057cc12.js
```

Whenever you change your media files the version hash of the affected files becomes different and thus the manifest file changes automatically, too.

Now how do we tell django-mediagenerator to create such a manifest file? Just add this to your `settings.py`:

```
OFFLINE_MANIFEST = 'webapp.manifest'
```

With this simple snippet the media generator will create a manifest file called `webapp.manifest`. However, the manifest file will contain **all** of the assets in your project. In other words, the whole `_generated_media` folder will be listed in the manifest file.

Often you only want specific files to be cached. You can do that by specifying a list of regular expressions matching path names (relative to your media directories, exactly like in `MEDIA_BUNDLES`):

```
OFFLINE_MANIFEST = {
    'webapp.manifest': {
        'cache': (
            r'main\.css',
            r'main\.js',
            r'webapp/img/.*',
        ),
        'exclude': (
            r'webapp/img/online-only/.*',
        )
    },
}
```

Here we've added the `main.css` and `main.js` bundles and all files under the `webapp/img/` folder, except for files under `webapp/img/online-only/`. Also, you might have guessed it already: You can create multiple manifest files this way. Just add more entries to the `OFFLINE_MANIFEST` dict.

Finally, we also have to include the manifest file in our template:

```
{% load media %}
<html manifest="{% media_url 'webapp.manifest' %}">
```

Manifest files actually provide more features than this. For example, you can also specify `FALLBACK` handlers in case there is no Internet connection. In the following example the "/offline.html" page will be displayed for resources which can't be reached while offline:

```
OFFLINE_MANIFEST = {
    'webapp.manifest': {
        'cache': (...),
        'fallback': {
            '/': '/offline.html',
        },
    },
}
```

Here `/` is a pattern that matches all pages. You can also define `NETWORK` entries which specify allowed URLs that can be accessed even though they're not cached:

```
OFFLINE_MANIFEST = {
    'webapp.manifest': {
        'cache': (...),
        'network': (
            '*',
        ),
    },
}
```

Here `*` is a wildcard that allows to access any URL. If you just had an empty `NETWORK` section you wouldn't be able to load uncached files, even when you're online (however, not all browsers are so strict).

Serving manifest files

Manifest files should be served with the MIME type `text/cache-manifest`. Also it's **critical** that you disable HTTP caching for manifest files! Otherwise the browser will **never** load a new version of your app because it always loads the cached manifest! Make sure that you've configured your web server correctly.

As an example, on App Engine you'd configure your `app.yaml` like this:

```
handlers:
- url: /media/(.*\.manifest)
  static_files: _generated_media/\1
  mime_type: text/cache-manifest
  upload: _generated_media/(.*\.manifest)
  expiration: '0'

- url: /media
  static_dir: _generated_media/
  expiration: '365d'
```

Here we first catch all manifest files and serve them with an expiration of "0" and the correct MIME type. The normal `/media` handler must be installed **after** the manifest handler.

Like a native iPad/iPhone app

Offline-capable web apps have a nice extra advantage: We can put them on the iPad's/iPhone's home screen, so they appear exactly like native apps! All browser bars will disappear and your whole web app will be full-screen (except for the top-most status bar which shows the current time and battery and network status). Just add the following to your template:

```
<head>
<meta name="apple-mobile-web-app-capable" content="yes" />
...
```

Now when you're in the browser you can tap on the "+" icon in the middle of the bottom toolbar (**update:** I just updated to iOS 4.2.1 and the "+" icon got replaced with some other icon, but it's still in the middle of the bottom toolbar :) and select "Add to Home Screen":

Then you can enter the name of the home screen icon:

Tapping "Add" will add an icon for your web app to the home screen:

When you tap that icon the canvas demo app starts in full-screen:

We can also specify an icon for your web app. For example, if your icon is in `img/app-icon.png` you can add it like this:

```
{% load media %}
<head>
<link rel="apple-touch-icon" href="{% media_url 'img/app-icon.png' %}" />
...
```

The image should measure 57x57 pixels.

Finally, you can also add a startup image which is displayed while your app loads. The following snippet assumes that the startup image is in `img/startup.png`:

```
{% load media %}
<head>
<link rel="apple-touch-startup-image" href="{% media_url 'img/startup.png' %}" />
...
```

The image dimensions should be 320x460 pixels and it should be in portrait orientation.

Summary

- The manifest file just lists the files that should be cached
- Files are only reloaded if the manifest file's content has changed
- The manifest file must not be cached (!) or the browser will never reload anything
- django-mediagenerator automatically maintains the manifest file for you
- Offline web apps can appear like native apps on the iPad and iPhone
- Download the latest canvas drawing app source which is now offline-capable

As you've seen in this post, it's very easy to make your web app offline-capable with django-mediagenerator. This is also the foundation for making your app look like a native app on the iPhone and iPad. Offline web apps open up exciting possibilities and allow you to become independent of Apple's slow approval processes for the app store and the iOS platform in general because web apps can run on Android, webOS, and many other mobile platforms. It's also possible to write a little wrapper for the App Store which just opens Safari with your website. That way users can still find your app in the App Store (in addition to the web).

The next time you want to write a native app for the iOS platform, consider making a web app, instead (unless you're writing e.g. a real-time game, of course).

CHAPTER 9

Managing per-field indexes on App Engine

An annoying problem when trying to reuse an existing Django app is that some apps use `TextField` instead of `CharField` and still want to filter on that field. On App Engine `TextField` is not indexed and thus can't be filtered against. One app which has this problem is django-openid-auth. Previously, you had to modify the model source code directly and replace `TextField` with `CharField` where necessary. However, this is not a good solution because whenever you update the code you have to apply the patch, again. Now, djangoappengine provides a solution which allows you to configure indexes for individual fields without changing the models. By decoupling DB-specific indexes from the model definition we simplify maintenance and increase code portability.

Example

Let's see how we can get django-openid-auth to work correctly without modifying the app's source code. First, you need to create a module which defines the indexing settings. Let's call it "gae_openid_settings.py":

```
from django_openid_auth.models import Association, UserOpenID

FIELD_INDEXES = {
    Association: {'indexed': ['server_url', 'assoc_type']},
    UserOpenID: {'indexed': ['claimed_id']},
}
```

Then, in your settings.py you have to specify the list of gae settings modules:

```
GAE_SETTINGS_MODULES = (
    'gae_openid_settings',
)
```

That's it. Now the `server_url`, `assoc_type`, and `claimed_id` `TextField`s will behave like `CharField` and get indexed by the datastore.

Note that we didn't place the index definition in the `django_openid_auth` package. It's better to keep them separate because that way upgrades are easier: Just update the `django_openid_auth` folder. No need to re-add the index definition (and you can't mistakenly delete the index definition during updates).

Optimization

You can also use this to optimize your models. For example, you might have fields which don't need to be indexed. The more indexes you have the slower `Model.save()` will be. Fields that shouldn't be indexed can be specified via `'unindexed'`:

```
from myapp.models import MyContact

FIELD_INDEXES = {
    MyContact: {
        'indexed': [...],
        'unindexed': ['creation_date', 'last_modified', ...],
    },
}
```

This also has a nice extra advantage: If you specify a `CharField` as "unindexed" it will behave like a `TextField` and allow for storing strings that are longer than 500 bytes. This can also be useful when trying to integrate 3rd-party apps.

We hope you'll find this feature helpful. Feel free to post sample settings for other reusable Django apps in the comments. See you next time with some very exciting releases!

CHAPTER 10

Writing a non-relational Django backend

In our April 1st post we claimed to have a simplified backend API. Well, this wasn't true, of course, but yesterday it has become true. The Django ORM is pretty complicated and it takes too much time for contributors to understand all the necessary details. In order to make the process as easy as possible we've implemented a backend template which provides a simple starting point for a new backend based on our simplified API. It also contains sample code, so you can better understand what each function does. All places where you have to make changes are marked with "`# TODO:`" comments. Note, you'll need djangotoolbox which provides the base classes for nonrel backends.

Let's start with `base.py`. You can use the `DatabaseCreation` class to define a custom `data_types` mapping from Django's fields to your database types. The types will later be passed to functions which you'll have to implement to convert values from and to the DB (`convert_value_from_db()` and `convert_value_to_db()`). If the default values work for you just leave the class untouched.

Also, if you want to maintain a DB connection we'd recommend storing it in `DatabaseWrapper`:

```
class DatabaseWrapper(NonrelDatabaseWrapper):
    def __init__(self, *args, **kwds):
        super(DatabaseWrapper, self).__init__(*args, **kwds)
        ...
        self.db_connection = connect(
            self.settings_dict['HOST'], self.settings_dict['PORT'],
            self.settings_dict['USER'], self.settings_dict['PASSWORD'])
```

The real meat is in `compiler.py`. Here, you have to define a BackendQuery class which handles query creation and execution. In the constructor you should create a low-level query instance for your connection. Depending on your DB API this might be nothing more than a dict, but let's say your DB provides a `LowLevelQuery` class:

```
class BackendQuery(NonrelQuery):
    def __init__(self, compiler, fields):
        super(BackendQuery, self).__init__(compiler, fields)
        self.db_query = LowLevelQuery(self.connection.db_connection)
```

Note, `self.connection` is the `DatabaseWrapper` instance which is the high-level DB connection object in Django.

Then, you need to define a function that converts Django's filters from Django's internal query object (`SQLQuery`, accessible via `self.query`) to their counterparts for your DB. This should be done in the `add_filters()` function. Since quite a few nonrel DBs seem to only support AND queries we provide a default implementation which makes sure that there is no OR filter (well, it has some logic for converting certain OR filters to AND filters). It expects an `add_filter()` function (without the trailing "s"):

```
@safe_call
def add_filter(self, column, lookup_type, negated, db_type, value):
    # Emulated/converted lookups
    if column == self.query.get_meta().pk.column:
        column = '_id'

    if negated:
        try:
            op = NEGATION_MAP[lookup_type]
        except KeyError:
            raise DatabaseError("Lookup type %r can't be negated" % lookup_type)
    else:
        try:
            op = OPERATORS_MAP[lookup_type]
        except KeyError:
            raise DatabaseError("Lookup type %r isn't supported" % lookup_type)

    # Handle special-case lookup types
    if callable(op):
        op, value = op(lookup_type, value)

    db_value = self.convert_value_for_db(db_type, value)
    self.db_query.filter(column, op, db_value)
```

This is just an example implementation. You don't have to use the same code. At first, we convert the primary key column to the DB's internal reserved column for the primary key. Then, we check if the filter should be negated or not and retrieve the respective DB comparison operator from a mapping like this:

```
OPERATORS_MAP = {
    'exact': '=',
    'gt': '>',
    'gte': '>=',
    # ...
    'isnull': lambda lookup_type, value: ('=' if value else '!=', None),
}

NEGATION_MAP = {
    'exact': '!=',
    'gt': '<=',
    # ...
    'isnull': lambda lookup_type, value: ('!=' if value else '=', None),
}
```

In our example implementation the operator can be a string or a callable that returns the comparison operator and a modified value. Finally, in the last two lines of `add_filter()` we convert the value to its low-level DB type and then add a filter to the low-level query object.

You might have noticed the `@save_call` decorator. This is important. It catches database exceptions and converts them to Django's `DatabaseError`. That decorator should be used for all your public API methods. Just modify the sample implementation in `compiler.py` to match your DB's needs.

Next, you have to define a `fetch()` function for retrieving the results from the configured query:

```
@safe_call
def fetch(self, low_mark, high_mark):
    if high_mark is None:
        # Infinite fetching
        results = self.db_query.fetch_infinite(offset=low_mark)
    elif high_mark > low_mark:
        # Range fetching
        results = self.db_query.fetch_range(high_mark - low_mark, low_mark)
    else:
        results = ()

    for entity in results:
        entity[self.query.get_meta().pk.column] = entity['_id']
        del entity['_id']
        yield entity
```

Here, `low_mark` and `high_mark` define the query range. If `high_mark` is not defined you should allow for iterating through the whole result set. At the end, we convert the internal primary key column, again, and return a dict representing the entity. If your DB also supports only fetching specific columns you should get the requested fields from `self.fields` (`field.column` contains the column name).

All values in the resulting dict are automatically converted via `SQLCompiler.convert_value_from_db()`. You have to implement that function (the backend template contains a sample implementation). That function gets a `db_type` parameter which is the type string as defined in your field type mapping in `DatabaseCreation.data_types`.

We won't look at the whole API in this post. There are additional functions for ordering, counting, and deleting the query results. It's pretty simple. The API might later get extended with support for aggregates, but currently you'll have to handle them at a lower level in your `SQLCompiler` implementation if your DB supports those features.

Another important function is called on `Model.save()`:

```
class SQLInsertCompiler(NonrelInsertCompiler, SQLCompiler):
    @safe_call
    def insert(self, data, return_id=False):
        pk_column = self.query.get_meta().pk.column
        if pk_column in data:
            data['_id'] = data[pk_column]
            del data[pk_column]

        pk = save_entity(self.connection.db_connection,
            self.query.get_meta().db_table, data)
        return pk
```

Again, `data` is a dict because that maps naturally to nonrel DBs. Note, before insert() is called, all values are automatically converted via `SQLCompiler.convert_value_for_db()` (which you have to implement, too), so you don't have to deal with value conversions in that function.

I hope this gives you enough information to get started with a new backend. Please spread the word, so we can find backend contributors for all non-relational DBs. Django 1.3 development is getting closer and in order to get officially integrated into Django we have to prove that it's possible to use Django-nonrel with a wide variety of NoSQL DBs.

Please comment on the API. Should we improve anything? Is it flexible and easy enough?

CHAPTER 11

Indices and tables

- genindex
- modindex
- search