Django App Engine Documentation

Release 1.6.0

AllButtonsPressed, Potato London, Wilfred Hughes

Jul 14, 2017

Contents

Djangoappengine contains 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.

Take a look at the documentation below and subscribe to our discussion group for the latest updates.

CHAPTER 1

Contents

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.

Note: Since Google App Engine runs your Python code from within a sandbox, some standard Python installation methods are unavailable. For example, you cannot install django or other Python modules in your system's Python library. All code for your app must be installed in your project directory.

Create a new directory for your project.

Download the following zip files:

- django-nonrel
- djangoappengine
- djangotoolbox
- django-autoload
- django-dbindexer

Unzip everything and copy the modules into your project directory.

Now you need to create a django project. Djangoappengine includes a project template which you can generate using the `django-admin.py` command. Run this command from within your project directory to create a new project:

```
PYTHONPATH=. python django/bin/django-admin.py startproject \
   --name=app.yaml --template=djangoappengine/conf/project_template myapp .
```

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

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

To start the local dev server, run this command:

```
./manage.py runserver
```

Note: You can also clone the Git repositories and copy the modules from there into your project. The repositories are available here: django-nonrel, djangoappengine, djangotoolbox, django-autoload, django-dbindexer. 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.

Fields, queries and indexes

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`.

Managing per-field indexes

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.

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
```

Email, caching and other services

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 *`).

Emails will be deferred to the task queue specified in the EMAIL_QUEUE_NAME setting. If you run the dev appserver with –disable_task_running then you'll see the tasks being deposited in the queue. You can manually execute those tasks from the GUI at /_ah/admin/tasks.

If you execute the dev appserver with the options `--smtp_host=localhost --smtp_port=1025` and run the dev smtp server in a terminal with `python -m smtpd -n -c DebuggingServer localhost:1025` then you'll see emails delivered to that terminal for debug.

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.

The django-permission-backend-nonrel repository contains fixes for Django's auth to make permissions and groups work on GAE (including the auth admin screens).

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.

Running djangoappengine

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. Running 'remote' executes your *local code*, but proxies your datastore access against the *remote datastore*.

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 10000 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.

Django admin

The native django admin site is a compelling reason to use django nonrel on the appengine, rather than webapp2 with native GAE datastore bindings.

However, there are limitations to be aware of when running contrib.admin against the GAE datastore instead of a rich SQL database:

- inequality queries against one index only
- need to use dbindexer to access 'contains'
- need to use dbindexer to index date/datetime fields used for date filters

Some of the consequences of the above restrictions:

- can't find users created before 2012 sorted by name
- can't find users like "Peter" sorted by date joined
- can't sort by multiple columns

For end-users this does lead to a limited admin site compared with 'relational' django. You may find that alternative/custom screens need to be implemented to meet all users' requirements on GAE (if they can be satisfied at all).

indexes.yaml

Beware of adding all the suggested composite indexes to your indexes.yaml when you use the admin site. It's easy to rack up the maximum 200 indexes and find that you struggle to add new functionality that needs further indexes. Instead, digest the Google index documentation and aim to exploit the zigzag merge query planner to reduce the number of indexes needed. This is particularly useful when you want to use several admin filters.

Sorting and searching

You may have an admin screen for some model sorted by 'name'. But users may also want to be able to search for an article, which depends on an inequality query against a dbindexer icontains index (a model's generated 'idxf_name_l_icontains' field). Since the datastore can't query against two inequality relations, you can't simply have this in your admin definition:

```
search_fields = ('name',)
ordering = ('name',)
```

This fails when you use the search, because that query is an inequality against dbindexer's generated 'idxf_name_l_icontains' field, which the datastore can't handle at the same time as sorting (inequality) against the 'name' field.

A useful pattern to get round this is to make your admin screen's ordering vary depending on whether a search is in action or not:

```
search_fields = ('name',)
def get_ordering(self, request):
    if 'q' in request.GET and request.GET['q'] != '':
        return ('idxf_name_l_icontains',)
    return ('name',)
```

In other words, don't specify the admin 'ordering' variable, but defer to the 'get_ordering' method, which sorts by name if no search is in action, or by the dbindexer field if it is.

Filters and sorting and searching

Date/datetime filters do work if you define some indexes using dbindexer, but you can't apply a date filter (which uses an inequality query for some of the options) at the same time as a search. And the sort order for a filtered model must

match the filter. For example:

```
list_filter = ('expired', OwnedFranchiseRegionFilter, 'status', 'start_date',)
ordering = ('-start_date',)
```

where 'start_date' is a Datefield and the other filters are simple equality filters. In this case, the admin screen cannot be ordered by name, and we can't enable search by name.

MapReduce

Overview

Djangoappengine provides a few classes and utilities to make working with Google App Engine's MapReduce library easier. There are many tutorials and documents on MapReduce in general, as well as on the specifics of App Engine's MapReduce library.

- MapReduce Wikipedia page
- Original Google MapReduce paper
- App Engine MapReduce library
- Getting started guide for Python MapReduce

Djangoappengine provides two modules to simplify running MapReduce jobs over Django models. DjangoModelMapreduce and DjangoModelMap are helper functions that quickly allow you to create a mapreduce or a simple map job. These functions make use of DjangoModelInputReader, an InputReader subclass, which returns model instances for a given Django model.

Installation

Checkout the mapreduce folder into your application directory:

```
svn checkout http://appengine-mapreduce.googlecode.com/svn/trunk/python/src/mapreduce
```

Add the mapreduce handlers to your app.yaml:

```
handlers:
- url: /mapreduce/pipeline/images
  static_dir: mapreduce/lib/pipeline/ui/images

- url: /mapreduce(/.*)?
  script: djangoappengine.mapreduce.handler.application
  login: admin
```

DjangoModelMapreduce and DjangoModelMap

DjangoModelMapreduce and DjangoModelMap are helper functions which return MapreducePipeline and MapperPipeline instances, respectively.

DjangoModelMap allows you to specify a model class and a single function called a mapper. The mapper function can do anything to your model instance, including When running the pipeline, the mapper function is called on each instance of your model class. As an example, consider this simple model:

```
class User(models.Model):
    name = models.CharField()
    city = models.CharField()
    age = models.IntegerField()
```

A mapper class which output the name and age of each User in a tab-delimited format looks like this:

```
from djangoappengine.mapreduce import pipeline as django_pipeline
from mapreduce import base_handler

def name_age(user):
    yield "%s\t%s\n" % (user.name, user.age)

class UserNameAgePipeline(base_handler.PipelineBase):
    def run(self):
        yield django_pipeline.DjangoModelMap(User, name_age)
```

A mapreduce class which outputs the average age of the users in each city looks like this:

```
from djangoappengine.mapreduce import pipeline as django_pipeline
from mapreduce import base_handler

def avg_age_mapper(user):
    yield (user.city, user.age)

def avg_age_reducer(city, age_list):
    yield "%s\t%s\n" % (city, float(sum(age_list))/len(age_list))

class AverageAgePipeline(base_handler.PipelineBase):
    def run(self):
        yield django_pipeline.DjangoModelMapreduce(User, avg_age_mapper, avg_age_
↪reducer)
```

CHAPTER 2

Contribute

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