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.

Documentation Source Download Discussion group Blog

Documentation

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.