django-extensions Documentation

Release 1.7.9

Michael Trier, Bas van Oostveen, and contributors

Jun 25, 2017

Contents

Django Extensions is a collection of custom extensions for the Django Framework.

These include management commands, additional database fields, admin extensions and much more.

CHAPTER 1

Getting Started

The easiest way to figure out what Django Extensions are all about is to watch the excellent screencast by Eric Holscher. In a couple minutes Eric walks you through a half a dozen command extensions.

CHAPTER 2

Getting it

You can get Django Extensions by using pip or easy_install:

```
$ pip install django-extensions
or
$ easy_install django-extensions
```

If you want to install it from source, grab the git repository and run setup.py:

```
$ git clone git://github.com/django-extensions/django-extensions.git
$ cd django-extensions
$ python setup.py install
```

For more detailed instructions check out our *Installation instructions*. Enjoy.

CHAPTER 3

Compatibility with versions of Python and Django

We follow the Django guidelines for supported Python and Django versions. See more at Django Supported Versions

This might mean the django-extensions may work with older or unsupported versions but we do not guarantee it and most likely will not fix bugs related to incompatibilities with older versions.

At this time we test on and thrive to support valid combinations of Python 2.7, 3.4, 3.5, pypy and pypy3 with Django versions 1.8 and 1.9.

CHAPTER 4

Contents

Installation instructions

synopsis Installing django-extensions

Installation

For usage

You can use pip to install django-extensions for usage:

```
$ pip install django-extensions
```

For development

Django-extensions is hosted on github:

```
https://github.com/django-extensions/django-extensions
```

Source code can be accessed by performing a Git clone.

Tracking the development version of *django command extensions* should be pretty stable and will keep you up-to-date with the latests fixes.

$ pip install -e git+https://github.com/django-extensions/django-extensions.git#egg=django-extensions

You find the sources in src/django-extensions now.

You can verify that the application is available on your PYTHONPATH by opening a python interpreter and entering the following commands:

```
>>> import django_extensions
>>> django_extensions.VERSION
(0, 8)
```

Keep in mind that the current code in the git repository may be different from the packaged release. It may contain bugs and backwards-incompatible changes but most likely also new goodies to play with.

Configuration

You will need to add the *django_extensions* application to the INSTALLED_APPS setting of your Django project *settings.py* file.:

```
INSTALLED_APPS = (
    ...
    'django_extensions',
)
```

This will make sure that Django finds the additional management commands provided by *django-extensions*.

The next time you invoke *./manage.py help* you should be able to see all the newly available commands.

Some commands or options require additional applications or python libraries, for example:

- 'export_emails' will require the *python vobject* module to create vcard files.
- 'graph_models' requires *pygraphviz* to render directly to image file.

If the given application or python library is not installed on your system (or not in the python path) the executed command will raise an exception and inform you of the missing dependency.

Current Command Extensions

synopsis Current Command Extensions

- *shell_plus* - An enhanced version of the Django shell. It will autoload all your models making it easy to work with the ORM right away.
- *admin_generator* - Generate automatic Django Admin classes by providing an app name. Outputs source code at STDOUT.
- *clean_pyc* - Remove all python bytecode compiled files from the project
- create_app - Creates an application directory structure for the specified app name. This command allows you to specify the –template option where you can indicate a template directory structure to use as your default.
- *create_command* - Creates a command extension directory structure within the specified application. This makes it easy to get started with adding a command extension to your application.
- *create_template_tags* - Creates a template tag directory structure within the specified application.
- *create_jobs* - Creates a Django jobs command directory structure for the given app name in the current directory. This is part of the impressive jobs system.
- *clear_cache* - Clear django cache, useful when testing or deploying.
- *compile_pyc* - Compile python bytecode files for the project.
- *describe_form* - Used to display a form definition for a model. Copy and paste the contents into your forms.py and you're ready to go.

- *delete_squashed_migrations* - Deletes leftover migrations after squashing and converts squashed migration to a normal one.
- *dumpscript* - Generates a Python script that will repopulate the database using objects. The advantage of this approach is that it is easy to understand, and more flexible than directly populating the database, or using XML.
- export_emails - export the email addresses for your users in one of many formats. Currently supports Address, Google, Outlook, LinkedIn, and VCard formats.
- *find_template* - Finds the location of the given template by resolving its path
- *generate_secret_key* - Creates a new secret key that you can put in your settings.py module.
- graph_models - Creates a GraphViz dot file. You need to send this output to a file yourself. Great for graphing your models. Pass multiple application names to combine all the models into a single dot file.
- *mail_debug* - Starts a mail server which echos out the contents of the email instead of sending it.
- *notes* - Show all annotations like TODO, FIXME, BUG, HACK, WARNING, NOTE or XXX in your py and HTML files.
- *passwd* - Makes it easy to reset a user's password.
- *pipchecker* - Scan pip requirement file(s)s for out-of-date packages. Similar to `pip list -o` which used installed packages (in virtualenv) instead of requirements file(s).
- print_settings - Similar to `diffsettings` but shows *selected* active Django settings or *all* if no args passed.
- *print_user_for_session* - Print the user information for the provided session key. this is very helpful when trying to track down the person who experienced a site crash. It seems this works only if setting `SESSION_ENGINE` is `'django.contrib.sessions.backends.db'` (default value).
- *drop_test_database* - Drops the test database. Usefull when running Django test via some automated system (BuildBot, Jenkins, etc) and making sure that the test database is always dropped at the end.
- *reset_db* - Resets a database (currently sqlite3, mysql, postgres). Uses "DROP DATABASE" and "CREATE DATABASE".
- *runjob* - Run a single maintenance job. Part of the jobs system.
- *runjobs* - Runs scheduled maintenance jobs. Specify hourly, daily, weekly, monthly. Part of the jobs system.
- *runprofileserver* - Starts *runserver* with hotshot/profiling tools enabled. I haven't had a chance to check this one out, but it looks really cool.
- runscript - Runs a script in the django context.
- runserver_plus - The standard runserver stuff but with the Werkzeug debugger baked in. Requires Werkzeug. This one kicks ass.
- *set_fake_emails* - Give all users a new email based on their account data ("%(username)s@example.com" by default). Possible parameters are: username, first_name, last_name. *DEBUG only*
- *set_fake_passwords* - Sets all user passwords to a common value (*password* by default). *DEBUG only*.
- *show_template_tags* - Displays template tags and filters available in the current project.
- *show_urls* - Displays the url routes that are defined in your project. Very crude at this point.
- *sqldiff* - Prints the (approximated) difference between an app's models and what is in the database. This is very nice, but also very experimental at the moment. It can not catch everything but it's a great sanity check.
- *sqlcreate* - Generates the SQL to create your database for you, as specified in settings.py.
- *sqldsn* - Reads the Django settings and extracts the parameters needed to connect to databases using other programs.

- sync_s3 - Copies files found in settings.MEDIA_ROOT to S3. Optionally can also gzip CSS and Javascript files and set the Content-Encoding header, and also set a far future expires header for browser caching.
- *syncdata* - Makes the current database have the same data as the fixture(s), no more, no less.
- *unreferenced_files* - Prints a list of all files in MEDIA_ROOT that are not referenced in the database.
- *update_permissions* - Reloads permissions for specified apps, or all apps if no args are specified.
- *validate_templates* - Validate templates on syntax and compile errors.
- *set_default_site* - Set parameters of the default *django.contrib.sites* Site using *name* and *domain* or *system-fqdn*.

Ideas for New Command Extensions

synopsis Here are some ideas for some future command extensions.

- create form/manager for App
- CSS and JS concatenation and minification scripts

Command Signals

synopsis Signals fired before and after a command is executed.

A signal is thrown pre/post each management command allowing your application to hook into each commands execution.

Basic Example

An example hooking into show_template_tags:

```
from django_extensions.management.signals import pre_command, post_command
from django_extensions.management.commands.show_template_tags import Command

def pre_receiver(sender, args, kwargs):
  # I'm executed prior to the management command

def post_receiver(sender, args, kwargs, outcome):
  # I'm executed after the management command

pre_command.connect(pre_receiver, Command)
post_command.connect(post_receiver, Command)
```

Custom Permissions For All Models

You can use the post signal to hook into the `update_permissions` command so that you can add your own permissions to each model.

For instance, lets say you want to add `list` and `view` permissions to each model. You could do this by adding them to the `permissions` tuple inside your models `Meta` class but this gets pretty tedious.

An easier solution is to hook into the `update_permissions` call, as follows;

```
from django.db.models.signals import post_syncdb
from django.contrib.contenttypes.models import ContentType
from django.contrib.auth.models import Permission
from django_extensions.management.signals import post_command
from django_extensions.management.commands.update_permissions import Command as␣
↪UpdatePermissionsCommand

def add_permissions(sender, **kwargs):
  """
  Add view and list permissions to all content types.
  """
  # for each of our content types
  for content_type in ContentType.objects.all():

    for action in ['view', 'list']:
      # build our permission slug
      codename = "%s_%s" % (action, content_type.model)

      try:
        Permission.objects.get(content_type=content_type, codename=codename)
        # Already exists, ignore
      except Permission.DoesNotExist:
        # Doesn't exist, add it
        Permission.objects.create(content_type=content_type,
                    codename=codename,
                    name="Can %s %s" % (action, content_type.name))
        print "Added %s permission for %s" % (action, content_type.name)
post_command.connect(add_permissions, UpdatePermissionsCommand)
```

Each time `update_permissions` is called `add_permissions` will be called which ensures there are view and list permissions to all content types.

Using pre/post signals on your own commands

The signals are implemented using a decorator on the handle method of a management command, thus using this functionality in your own application is trivial:

```
from django_extensions.management.utils import signalcommand

class Command(BaseCommand):

  @signalcommand
  def handle(self, *args, **kwargs):
    ...
    ...
```

Current Admin Extensions

synopsis Current Field Extensions

- *ForeignKeyAutocompleteAdmin* - ForeignKeyAutocompleteAdmin will enable the admin app to show ForeignKey fields with an search input field. The search field is rendered by the ForeignKeySearchInput form widget and uses jQuery to do configurable autocompletion.

Example Usage

To enable the Admin Autocomplete you can follow this code example in your admin.py file:

```
from django.contrib import admin
from foo.models import Permission
from django_extensions.admin import ForeignKeyAutocompleteAdmin


class PermissionAdmin(ForeignKeyAutocompleteAdmin):
    # User is your FK attribute in your model
    # first_name and email are attributes to search for in the FK model
    related_search_fields = {
       'user': ('first_name', 'email'),
    }

    fields = ('user', 'avatar', 'is_active')

    ...

admin.site.register(Permission, PermissionAdmin)
```

If you are using django-reversion you should follow this code example:

```
from django.contrib import admin
from foo.models import MyVersionModel
from reversion.admin import VersionAdmin
from django_extensions.admin import ForeignKeyAutocompleteAdmin


class MyVersionModelAdmin(VersionAdmin, ForeignKeyAutocompleteAdmin):
    ...

admin.site.register(MyVersionModel, MyVersionModelAdmin)
```

If you need to limit the autocomplete search, you can override the `get_related_filter` method of the admin. For example if you want to allow non-superusers to attach attachments only to articles they own you can use:

```
class AttachmentAdmin(ForeignKeyAutocompleteAdmin):

    ...

    def get_related_filter(self, model, request):
        user = request.user
        if not issubclass(model, Article) or user.is_superuser():
            return super(AttachmentAdmin, self).get_related_filter(
                model, request
            )
        return Q(owner=user)
```

Note that this does not protect your application from malicious attempts to circumvent it (e.g. sending fabricated requests via cURL).

shell_plus

synopsis Django shell with autoloading of the apps database models

Interactive Python Shells

There is support for three different types of interactive python shells.

IPython:

```
$ ./manage.py shell_plus --ipython
```

bpython:

```
$ ./manage.py shell_plus --bpython
```

ptpython:

```
$ ./manage.py shell_plus --ptpython
```

Python:

```
$ ./manage.py shell_plus --plain
```

The default resolution order is: ptpython, bpython, ipython, python.

You can also set the configuration option SHELL_PLUS to explicitly specify which version you want.

```
# Always use IPython for shell_plus
SHELL_PLUS = "ipython"
```

It is also possible to use IPython Notebook, an interactive Python shell which uses a web browser as its user interface, as an alternative shell:

```
$ ./manage.py shell_plus --notebook
```

In addition to being savable, IPython Notebooks can be updated (while running) to reflect changes in a Django application's code with the menu command *Kernel* > *Restart*.

Configuration

Sometimes, models from your own apps and other people's apps have colliding names, or you may want to completely skip loading an app's models. Here are some examples of how to do that.

Note: These settings are only used inside shell_plus and will not affect your environment.

```
# Rename the automatic loaded module Messages in the app blog to blog_messages.
SHELL_PLUS_MODEL_ALIASES = {'blog': {'Messages': 'blog_messages'},}
```

```
# Prefix all automatically loaded models in the app blog with myblog.
SHELL_PLUS_APP_PREFIXES = {'blog': 'myblog',}
```

```
# Dont load the 'sites' app, and skip the model 'pictures' in the app 'blog'
SHELL_PLUS_DONT_LOAD = ['sites', 'blog.pictures']
```

You can also combine model_aliases and dont_load.

When referencing nested modules, e.g. *somepackage.someapp.models.somemodel*, omit the package name and the reference to *models*. For example:

```
SHELL_PLUS_DONT_LOAD = ['someapp.somemodel', ]  # This works
SHELL_PLUS_DONT_LOAD = ['somepackage.someapp.models.somemodel', ]  # This does NOT␣
↪work
```

It is possible to ignore autoloaded modules when using manage.py, like:

```
$ ./manage.py shell_plus --dont-load app1 --dont-load app2.module1
```

Commandline parameters and settings in the configuration file are merged, so you can safely append modules to ignore from the commandline for one-time usage.

There are two settings that you can use to pass your custom options to the IPython Notebook in your Django settings.

The first one is `NOTEBOOK_ARGUMENTS` that can be used to hold those options that available via:

```
$ ipython notebook -h
```

For example:

```
NOTEBOOK_ARGUMENTS = [
    '--ip', 'x.x.x.x',
    '--port', 'xx',
]
```

Another one is `IPYTHON_ARGUMENTS` that for those options that available via:

```
$ ipython -h
```

The Django settings module and database models are auto-loaded into the interactive shell's global namespace also for IPython Notebook.

Auto-loading is done by a custom IPython extension which is activated by default by passing the `--ext django_extensions.management.notebook_extension` argument to the Notebook. If you need to pass custom options to the IPython Notebook, you can override the default options in your Django settings using the `IPYTHON_ARGUMENTS` setting. For example:

```
IPYTHON_ARGUMENTS = [
    '--ext', 'django_extensions.management.notebook_extension',
    '--ext', 'myproject.notebook_extension',
    '--debug',
]
```

To activate auto-loading, remember to either include the django-extensions' default notebook extension or copy its auto-loading code into your own extension.

Note that the IPython Notebook feature doesn't currently honor the `--dont-load` option.

Additional Imports

In addition to importing the models you can specify other items to import by default. These are specified in SHELL_PLUS_PRE_IMPORTS and SHELL_PLUS_POST_IMPORTS. The former is imported before any other imports (such as the default models import) and the latter is imported after any other imports. Both have similar syntax. So in your settings.py file:

```
SHELL_PLUS_PRE_IMPORTS = [
    ('module.submodule1', ('class1', 'function2')),
    ('module.submodule2', 'function3'),
```

```
    ('module.submodule3', '*'),
    'module.submodule4'
]
```

The above example would directly translate to the following python code which would be executed before the automatic imports:

```
from module.submodule1 import class1, function2
from module.submodule2 import function3
from module.submodule3 import *
import module.submodule4
```

These symbols will be available as soon as the shell starts.

Database application signature

If using PostgreSQL the `application_name` is set by default to `django_shell` to help identify queries made under shell_plus.

SQL queries

It is possible to print SQL queries as they're executed in shell_plus like:

```
$ ./manage.py shell_plus --print-sql
```

You can also set the configuration option SHELL_PLUS_PRINT_SQL to omit the above command line option.

```
# print SQL queries in shell_plus
SHELL_PLUS_PRINT_SQL = True
```

create_app

synopsis Creates an application directory structure for the specified application name.

This command allows you to specify the –template option where you can indicate a template directory structure to use as your default.

The –diagram option generates the models.py and admin.py from a .dia file.

Example Usage

All examples assume your current directory is the project directory and settings.py is under it.

```
# Get command help
./manage.py create_app --help
```

```
# Generate models.py and admin.py from [APP_NAME].dia file. This file should
# be placed in the settings.py directory.
./manage.py create_app -d APP_NAME
```

Example generated from sample.dia

```
./manage.py create_app --diagram=sample.dia webdata
```

-d switch or –diagram option use dia2django to generate models.py and is better documented in django wiki.

delete_squashed_migrations

synopsis Deletes leftover migrations after squashing and converts squashed migration to a normal one.

Deletes leftover migrations after squashing and converts squashed migration to a normal one by removing the replaces attribute. This automates the clean up procedure outlined at the end of the Django migration squashing documentation. Modifies your source tree! Use with care!

Example Usage

With *django-extensions* installed you cleanup squashed migrations using the *delete_squashed_migrations* command:

```
# Delete leftover migrations from the first squashed migration found in myapp
$ ./manage.py delete_squashed_migrations myapp

# As above but non-interactive
$ ./manage.py --noinput delete_squashed_migrations myapp

# Explicitly specify the squashed migration to clean up
$ ./manage.py delete_squashed_migrations myapp 0001_squashed
```

dumpscript

synopsis Generates a standalone Python script that will repopulate the database using objects.

The *dumpscript* command generates a standalone Python script that will repopulate the database using objects. The advantage of this approach is that it is easy to understand, and more flexible than directly populating the database, or using XML.

Why?

There are a few benefits to this:

- less drama with model evolution: foreign keys handled naturally without IDs, new and removed columns are ignored
- edit script to create 1,000s of generated entries using for loops, generated names, python modules etc.

For example, an edited script can populate the database with test data:

```
for i in xrange(2000):
    poll = Poll()
    poll.question = "Question #%d" % i
    poll.pub_date = date(2001,01,01) + timedelta(days=i)
    poll.save()
```

Real databases will probably be bigger and more complicated so it is useful to enter some values using the admin interface and then edit the generated scripts.

Features

- *ForeignKey* and *ManyToManyFields* (using python variables, not object IDs)
- Self-referencing *ForeignKey* (and M2M) fields
- Sub-classed models
- *ContentType* fields and generic relationships
- Recursive references
- *AutoFields* are excluded
- Parent models are only included when no other child model links to it
- Individual models can be referenced

How?

To dump the data from all the models in a given Django app (*appname*):

```
$ ./manage.py dumpscript appname > scripts/testdata.py
```

To dump the data from just a single model (*appname.ModelName*):

```
$ ./manage.py dumpscript appname.ModelName > scripts/testdata.py
```

To reset a given app, and reload with the saved data:

```
$ ./manage.py reset appname
$ ./manage.py runscript testdata
```

Note: Runscript needs *scripts* to be a module, so create the directory and a *__init__.py* file.

Caveats

Naming conflicts

Please take care that when naming the output files these filenames do not clash with other names in your import path. For instance, if the appname is the same as the script name, an importerror can occur because rather than importing the application modules it tries to load the modules from the dumpscript file itself.

Examples:

```
# Wrong
$ ./manage.py dumpscript appname > dumps/appname.py

# Right
$ ./manage.py dumpscript appname > dumps/appname_all.py

# Right
$ ./manage.py dumpscript appname.Somemodel > dumps/appname_somemodel.py
```

RunScript

synopsis Runs a script in the django context.

Introduction

The runscript command lets you run an arbritrary set of python commands within the django context. It offers the same usability and functionality as running a set of commands in shell accessed by:

```
$ python manage.py shell
```

Getting Started

This example assumes you have followed the tutorial for Django 1.8+, and created a polls app containing a `Question` model. We will create a script that deletes all of the questions from the database.

To get started create a scripts directory in your project root, next to manage.py:

```
$ mkdir scripts
$ touch scripts/__init__.py
```

Note: The *__init__.py* file is necessary so that the folder is picked up as a python package.

Next, create a python file with the name of the script you want to run within the scripts directory:

```
$ touch scripts/delete_all_questions.py
```

This file must implement a *run()* function. This is what gets called when you run the script. You can import any models or other parts of your django project to use in these scripts.

For example:

```
# scripts/delete_all_questions.py

from polls.models import Question

def run():
    # Fetch all questions
    questions = Question.objects.all()
    # Delete questions
    questions.delete()
```

Note: You can put a script inside a *scripts* folder in any of your apps too.

Usage

To run any script you use the command *runscript* with the name of the script that you want to run.

For example:

```
$ python manage.py runscript delete_all_questions
```

Note: The command first checks for scripts in your apps i.e. *app_name/scripts* folder and runs them before checking for and running scripts in the *project_root/scripts* folder. You can have multiple scripts with the same name and they will all be run sequentially.

Passing arguments

You can pass arguments from the command line to your script by passing a space separated list of values with `--script-args`. For example:

```
$ python manage.py runscript delete_all_questions --script-args staleonly
```

The list of argument values gets passed as arguments to your *run()* function. For example:

```
# scripts/delete_all_questions.py
from datetime import timedelta

from django.utils import timezone

from polls.models import Question

def run(*args):
    # Get all questions
    questions = Question.objects.all()
    if 'staleonly' in args:
        # Only get questions more than 100 days old
        questions = questions.filter(pub_date__lt=timezone.now() -␣
↪timedelta(days=100))
    # Delete questions
    questions.delete()
```

Debugging

If an exception occurs you will not get a traceback by default. To get a traceback specify `--traceback`. For example:

```
$ python manage.py runscript delete_all_questions --traceback
```

export_emails

synopsis export the email addresses for your users in one of many formats

Most Django sites include a registered user base. There are times when you would like to import these e-mail addresses into other systems (generic mail program, Gmail, Google Docs invites, give edit permissions, LinkedIn Group pre-approved listing, etc.). The export_emails command extension gives you this ability. Exported users can be filtered by Group name association.

Example Usage

```
# Export all the addresses in the '"First Last" <my@addr.com>;' format.
$ ./manage.py export_emails > addresses.txt
```

```
# Export users from the group 'Attendees' in the linked in pre-approve Group csv␣
↪format.
$ ./manage.py export_emails -g Attendees -f linkedin pycon08.csv
```

```
# Create a csv file importable by Gmail or Google Docs
$ ./manage.py export_emails --format=google google.csv
```

Supported Formats

address

This is the default basic text format. Each entry is on its own line in the format:

```
"First Last" <user@host.com>;
```

This can be used with all known mail programs (that I know about anyway).

google

A CSV (comma separated value) format which Google applications can import. This can be used to import directly into Gmail, a Gmail mailing group, Google Docs invite (to read), Google Docs grant edit permissions, Google Calendar invites, etc.

Only two columns are supplied. One for the person's name and one for the email address. This is also nice for importing into spreadsheets.

outlook

A CSV (comma separated value) format which Outlook can parse and import. Supplies all the columns that Outlook 'requires', but only the name and email address are supplied.

linkedin

A CSV (comma separated value) format which can be imported by LinkedIn Groups to pre-approve a list of people for joining the group.

This supplies 3 columns: first name, last name, and email address. This is the best generic csv file for importing into spreadsheets as well.

vcard

A vCard format which Apple Address Book can parse and import.

Field Extensions

synopsis Current Field Extensions

Current Database Model Field Extensions

- *AutoSlugField* - AutoSlugfield will automatically create a unique slug incrementing an appended number on the slug until it is unique. Inspired by SmileyChris' Unique Slugify snippet.

  AutoSlugField takes a *populate_from* argument that specifies which field, list of fields, or model method the slug will be populated from, for instance:

```
slug = AutoSlugField(populate_from=['title', 'description', 'get_author_name'])
```

  populate_from can traverse a ForeignKey relationship by using Django ORM syntax:

```
slug = AutoSlugField(populate_from=['related_model__title', 'related_model__get_
↪readable_name'])
```

- *RandomCharField* - AutoRandomCharField will automatically create a unique random character field with the specified length. By default upper/lower case and digits are included as possible characters. Given a length of 8 thats yields 3.4 million possible combinations. A 12 character field would yield about 2 billion. Below are some examples:

```
>>> RandomCharField(length=8, unique=True)
BVm9GEaE

>>> RandomCharField(length=4, include_alpha=False)
7097

>>> RandomCharField(length=12, include_punctuation=True)
k[ZS.TR,0LHO

>>> RandomCharField(length=12, lowercase=True, include_digits=False)
pzolbemetmok
```

- *CreationDateTimeField* - DateTimeField that will automatically set its date when the object is first saved to the database. Works in the same way as the auto_now_add keyword.
- *ModificationDateTimeField* - DateTimeField that will automatically set its date when an object is saved to the database. Works in the same way as the auto_now keyword. It is possible to preserve the current timestamp by setting update_modified to False:

```
>>> example = MyTimeStampedModel.objects.get(pk=1)

>>> print example.modified
datetime.datetime(2016, 3, 18, 10, 3, 39, 740349, tzinfo=<UTC>)

>>> example.save(update_modified=False)

>>> print example.modified
datetime.datetime(2016, 3, 18, 10, 3, 39, 740349, tzinfo=<UTC>)

>>> example.save()

>>> print example.modified
datetime.datetime(2016, 4, 8, 14, 25, 43, 123456, tzinfo=<UTC>)
```

  It is also possible to set the attribute directly on the model, for example when you don't use the TimeStamped-Model provided in this package, or when you are in a migration:

```
>>> example = MyCustomModel.objects.get(pk=1)

>>> print example.modified
datetime.datetime(2016, 3, 18, 10, 3, 39, 740349, tzinfo=<UTC>)

>>> example.update_modified=False

>>> example.save()

>>> print example.modified
datetime.datetime(2016, 3, 18, 10, 3, 39, 740349, tzinfo=<UTC>)
```

- *UUIDField* - UUIDField for Django, supports all uuid versions that are natively supported by the uuid python module.

  Deprecated since version 1.4.7: Django 1.8 features a native UUIDField. Django-Extensions will support *UUIDField* at the very least until Django 1.7 becomes unsupported.

- *PostgreSQLUUIDField* - UUIDField for Django, uses PostgreSQL uuid type.

  Deprecated since version 1.4.7: Django 1.8 features a native UUIDField. Django-Extensions will support *UUIDField* at the very least until Django 1.7 becomes unsupported.

- *EncryptedCharField* - CharField which transparently encrypts its value as it goes in and out of the database. Encryption is handled by Keyczar. To use this field you must have Keyczar installed, have generated a primary encryption key, and have `settings.ENCRYPTED_FIELD_KEYS_DIR` set to the full path of your keys directory.

- *EncryptedTextField* - CharField which transparently encrypts its value as it goes in and out of the database. Encryption is handled by Keyczar. To use this field you must have Keyczar installed, have generated a primary encryption key, and have `settings.ENCRYPTED_FIELD_KEYS_DIR` set to the full path of your keys directory.

- *ShortUUIDField* - CharField which transparently generates a UUID and pass it to base57. It result in shorter 22 characters values useful e.g. for concise, unambiguous URLS. It's possible to get shorter values with length parameter: they are not Universal Unique any more but probability of collision is still low

- *JSONField* - a generic TextField that neatly serializes/unserializes JSON objects seamlessly. Django 1.9 introduces a native JSONField for PostgreSQL, which is preferred for PostgreSQL users on Django 1.9 and above.

Graph models

synopsis Renders a graphical overview of your project or specified apps.

Creates a GraphViz dot file for the specified app names based on their models.py. You can pass multiple app names and they will all be combined into a single model. Output is usually directed to a dot file.

Several options are available: grouping models, including inheritance, excluding models and columns, and changing the layout when rendering to an output image.

With the latest revisions it's also possible to specify an output file if pygraphviz is installed and render directly to an image or other supported file-type.

Selecting a library

You need to select the library to generate the image. You can do so by passing the –pygraphviz or –pydot parameter, depending on which library you want to use.

When neither of the command line parameters are given the default is to try and load pygraphviz or pydot (in that order) to generate the image.

To install pygraphviz you usually need to run this command:

```
$ pip install pygraphviz
```

It is possible you can't install it because it needs some C extensions to build. In that case you can try other methods to install or you can use PyDot.

To install pydot you need to run this command:

```
$ pip install pyparsing==1.5.7
$ pip install pydot
```

Installation should be fast and easy. Remember to install this exact version of pyparsing, otherwise it's possible you get this error:

> Couldn't import dot_parser, loading of dot files will not be possible.

Default Settings

The option GRAPH_MODELS = { } can be used in the settings file to specify default options:

```
GRAPH_MODELS = {
  'all_applications': True,
  'group_models': True,
}
```

It uses the same names as on the command line only with the leading two dashes removed and the other dashes replaced by underscores.

Templates

Django templates are used to generate the dot code. This in turn can be drawn into a image by libraries like *pygraphviz* or *pydot*. You can extend or override the templates if needed.

Templates used:

- django_extensions/graph_models/digraph.dot
- django_extensions/graph_models/label.dot
- django_extensions/graph_models/relation.dot

Documentation on how to create dot files can be found here: http://www.graphviz.org/Documentation.php

Warning: Modifying Django's default templates behaviour might break *graph_models*

Please be aware that if you use any *template_loaders* or extensions that change the way templates are rendered that this can cause *graph_models* to fail.

An example of this is the Django app *django-template-minifier* this automatically removed the newlines before/after template tags even for none-html templates which leads to a mallformed file.

Example Usage

With *django-extensions* installed you can create a dot-file or an image by using the *graph_models* command:

```
# Create a dot file
$ ./manage.py graph_models -a > my_project.dot
```

```
# Create a PNG image file called my_project_visualized.png with application grouping
$ ./manage.py graph_models -a -g -o my_project_visualized.png

# Same example but with explicit selection of pygraphviz or pydot
$ ./manage.py graph_models --pygraphviz -a -g -o my_project_visualized.png
$ ./manage.py graph_models --pydot -a -g -o my_project_visualized.png
```

```
# Create a dot file for only the 'foo' and 'bar' applications of your project
$ ./manage.py graph_models foo bar > my_project.dot
```

```
# Create a graph for only certain models
$ ./manage.py graph_models -a -I Foo,Bar -o my_project_subsystem.png
```

```
# Create a excluding certain models
$ ./manage.py graph_models -a -X Foo,Bar -o my_project_sans_foo_bar.png
```

Jobs scheduling

synopsis Documentation on creating/using jobs in Django-extensions

JobsScheduling

This page is very much a Work In Progress

Creating jobs works much like management commands work in Django. Use create_jobs to make a 'jobs' directory inside of an application. After that create one python file per job.

Some simple examples are provided by the django_extensions.jobs package.

A job is a python script with a mandatory Job class which extends from HourlyJob, DailyJob, WeeklyJob or MonthlyJob. It has one method that must be implemented called 'execute', which is called when the job is run.

The following commands are related to jobs:

- create_jobs, create the directory structure for jobs
- runjob, run a single job
- runjobs, run all hourly/daily/weekly/monthly jobs

Use "runjob(s) -l" to list all jobs recognized.

Jobs do not run automatically !

You must either run a job manually specifying the exact time on which the command is to be run, or use crontab:

```
@hourly /path/to/my/project/manage.py runjobs hourly
```

```
@daily /path/to/my/project/manage.py runjobs daily
```

```
@weekly /path/to/my/project/manage.py runjobs weekly
```

```
@monthly /path/to/my/project/manage.py runjobs monthly
```

Model extensions

synopsis Current Model Extensions

Current Database Model Extensions

- *TimeStampedModel* - TimeStampedModel An abstract base class model that provides self-managed "created" and "modified" fields.
- *TitleDescriptionModel* - An abstract base class model that provides "title" and "description" fields.
- *TitleSlugDescriptionModel* - An abstract base class model that provides "title" and "description" fields and a self-managed "slug" field that populates from the title.

Namespace proposal

synopsis Namespace Proposal

Introduction

Please change / write your proposal for splitting django_extensions into namespaces here.

Proposal of a Namespace

Rough proposal for splitting into functional parts:

- django_extensions.commands (20% that everbody uses / production)
- django_extensions.commands.development (everything development)
- django_extensions.commands.extra (not fitting about category's?)
- django_extensions.db
- django_extensions.templates
- django_extensions.jobs

The db part should be okay where it is right now. It's only used when somebody explicitly imports:

```
from django_extensions.db.models import something
```

print_settings

synopsis Django managment command similar to `diffsettings` but shows *selected* active Django settings or *all* if no args passed.

Introduction

Django comes with a `diffsettings` command that shows how your project's settings differ from the Django defaults. Sometimes it is useful to just see the settings that are in effect for your project. This is particularly true if you have a more complex system for settings than just a single `settings.py` file. For example, you might have settings files that import other settings file, such as dev, test, and production settings files that source a base settings file.

This command also supports dumping the data in a few different formats.

More Info

The simplest way to run it is with no arguments:

```
$ python manage.py print_settings
```

Some variations:

```
$ python manage.py print_settings --format=json
$ python manage.py print_settings --format=yaml    # Requires PyYAML
$ python manage.py print_settings --format=pprint
$ python manage.py print_settings --format=text
$ python manage.py print_settings --format=value
```

Show just selected settings:

```
$ python manage.py print_settings DEBUG INSTALLED_APPS
$ python manage.py print_settings DEBUG INSTALLED_APPS --format=pprint
$ python manage.py print_settings INSTALLED_APPS --format=value
```

For more info, take a look at the built-in help:

```
$ python manage.py print_settings --help
Usage: manage.py print_settings [options]

Print the active Django settings.

Options:
  -v VERBOSITY, --verbosity=VERBOSITY
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings=SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath=PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Print traceback on exception
  --format=FORMAT       Specifies output format.
  --indent=INDENT       Specifies indent level for JSON and YAML
```

```
  --version             show program's version number and exit
  -h, --help            show this help message and exit
```

RunProfileServer

We recommend that before you start profiling any language or framework you learn enough about it so that you feel comfortable with digging into its internals.

Without sufficient knowledge it will not only be (very) hard but you're likely to make wrong assumptions (and fixes). As a rule of thumb, clean, well written code will help you a lot more than overzealous micro-optimizations will.

This document is work in progress. If you feel you can help with better/clearer or additional information about profiling Django please leave a comment.

Introduction

runprofileserver starts Django's runserver command with hotshot/profiling tools enabled. It will save .prof files containing the profiling information into the –prof-path directory. Note that for each request made one profile data file is saved.

By default the profile-data-files are saved in /tmp use the –prof-path option to specify your own target directory. Saving the data in a meaningful directory structure helps to keep your profile data organized and keeps /tmp uncluttered. (Yes this probably malfunctions systems such as Windows where /tmp does not exist)

To define profile filenames use –prof-file option. Default format is "{path}.{duration:06d}ms.{time}" (Python Format Specification is used).

Examples:

- "{time}-{path}-{duration}ms" - to order profile-data-files by request time
- "{duration:06d}ms.{path}.{time}" - to order by request duration

gather_profile_stats.py

Django comes packed with a tool to aggregate these different prof files into one aggregated profile file. This tool is called *gather_profile_stats.py* and is located inside the *bin* directory of your Django distribution.

Profiler choice

runprofileserver supports two profilers: *hotshot* and *cProfile*. Both come with the standard Python library but *cProfile* is more recent and may not be available on all systems. For this reason, *hotshot* is the default profiler.

However, *hotshot* is not maintained anymore and using *cProfile* is usually the recommended way. If it is available on your system, you can use it with the option `--use-cprofile`.

Example:

```
$ mkdir /tmp/my-profile-data
$ ./manage.py runprofileserver --use-cprofile --prof-path=/tmp/my-profile-data
```

If you used the default profiler but are not able to open the profiling results with the `pstats` module or with your profiling GUI of choice because of an error "*ValueError: bad marshal data (unknown type code)*", try using *cProfile* instead.

KCacheGrind

Recent versions of *runprofileserver* have an option to save the profile data into a KCacheGrind compatible format. So you can use the excellent KCacheGrind tool for analyzing the profile data.

Example:

```
$ mkdir /tmp/my-profile-data
$ ./manage.py runprofileserver --kcachegrind --prof-path=/tmp/my-profile-data
Validating models...
0 errors found

Django version X.Y.Z, using settings 'complete_project.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[13/Nov/2008 06:29:38] "GET / HTTP/1.1" 200 41107
[13/Nov/2008 06:29:39] "GET /site_media/base.css?743 HTTP/1.1" 200 17227
[13/Nov/2008 06:29:39] "GET /site_media/logo.png HTTP/1.1" 200 3474
[13/Nov/2008 06:29:39] "GET /site_media/jquery.js HTTP/1.1" 200 31033
[13/Nov/2008 06:29:39] "GET /site_media/heading.png HTTP/1.1" 200 247
[13/Nov/2008 06:29:39] "GET /site_media/base.js HTTP/1.1" 200 751
<ctrl-c>
$ kcachegrind /tmp/my-profile-data/root.12574391.592.prof
```

Here is a screenshot of how the above commands might look in KCacheGrind:

> http://trbs.net/media/misc/django-runprofileserver-kcachegrind-full.jpg

Links

- http://code.djangoproject.com/wiki/ProfilingDjango
- http://www.rkblog.rk.edu.pl/w/p/django-profiling-hotshot-and-kcachegrind/
- http://code.djangoproject.com/browser/django/trunk/django/bin/profiling/gather_profile_stats.py
- http://www.oluyede.org/blog/2007/03/07/profiling-django/
- http://simonwillison.net/2008/May/22/debugging/

RunServerPlus

> **synopsis** RunServerPlus-typical runserver with Werkzeug debugger baked in

Introduction

This item requires that you have the *Werkzeug WSGI utilities* installed. Included with Werkzeug is a kick ass debugger that renders nice debugging tracebacks and adds an AJAX based debugger (which allows code execution in the context of the traceback's frames). Additionally it provides a nice access view to the source code.

Getting Started

To get started we just use the *runserver_plus* command instead of the normal *runserver* command:

```
$ python manage.py runserver_plus

* Running on http://127.0.0.1:8000/
* Restarting with reloader...

Validating models...
0 errors found

Django version X.Y.Z, using settings 'screencasts.settings'
Development server is running at http://127.0.0.1:8000/
Using the Werkzeug debugger (http://werkzeug.pocoo.org/)
Quit the server with CONTROL-C.
```

Note: all normal runserver options apply. In other words, if you need to change the port number or the host information, you can do so like you would normally.

Usage

Instead of the default Django traceback page, the Werkzeug traceback page will be shown when an exception occurs.

Along with the typical traceback information we have a couple of options. These options appear when hovering over a particular traceback line. Notice that two buttons appear to the right:

The options are:

View Source

This displays the source underneath the traceback:

Being able to view the source file is handy because it provides more context information around the error. The actual traceback areas are highlighted so they are easy to spot.

One awkward aspect of the UI is that the page is not scrolled to the bottom. At first I thought nothing was happening because of this.

Interactive Debugging Console

Clicking on this button opens up a new pane under the traceback line you're on. This is the money shot:

An ajax based console appears in the pane and you can start debugging. Notice in the screenshot above I did a *print environ* to see what was in the environment parameter coming into the function.

WARNING: This should *never* be used in any kind of production environment. Not even for a quick problem check. I cannot emphasize this enough. The interactive debugger allows you to evaluate python code right against the server. You've been warned.

SSL

runserver_plus also supports SSL, so that you can easily debug bugs that pop up when https is used. To use SSL simply provide a file name for certificates; a key and certificate file will be automatically generated:

```
$ python manage.py runserver_plus --cert cert
Validating models...
0 errors found
```

```
Django version X.Y.Z, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Using the Werkzeug debugger (http://werkzeug.pocoo.org/)
Quit the server with CONTROL-C.
 * Running on https://127.0.0.1:8000/
 * Restarting with reloader
Validating models...
0 errors found

Django version X.Y.Z, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Using the Werkzeug debugger (http://werkzeug.pocoo.org/)
Quit the server with CONTROL-C.
```

After running this command, your web application can be accessed through https://127.0.0.1:8000.

You will also find that two files are created in the current working directory: a key file and a certificate file. If you run the above command again, these certificate files will be reused so that you do not have to keep accepting the self-generated certificates from your browser every time. You can also provide a specific file for the certificate to be used if you already have one:

```
$ python manage.py runserver_plus --cert /tmp/cert
```

Note that you need the OpenSSL library to use SSL, and Werkzeug 0.9 or later if you want to reuse existing certificates.

To install OpenSSL:

```
$ pip install pyOpenSSL
```

Configuration

The *RUNSERVERPLUS_SERVER_ADDRESS_PORT* setting can be configured to specify which address and port the development server should bind to.

If you find yourself frequently starting the server with:

```
$ python manage.py runserver_plus 0.0.0.0:8000
```

You can use settings to automatically default your development to an address/port:

```
RUNSERVERPLUS_SERVER_ADDRESS_PORT = '0.0.0.0:8000'
```

To ensure Werkzeug can log to the console, you may need to add the following to your settings:

```
LOGGING = {
    ...
    'handlers': {
        ...
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        ...
        'werkzeug': {
            'handlers': ['console'],
```

```
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

IO Calls and CPU Usage

As noted in gh625 *runserver_plus* can be seen to use a lot of CPU and generate many I/O when idle.

This is due to the way Werkzeug has implemented the auto reload capability. It supports two ways of doing auto reloading either via *stat polling* or *file system events*.

The *stat polling* approach is pretty brute force and continously issues *stat* system calls which causes the CPU and IO load.

If possible try to install the Watchdog package, this should automatically cause Werkzeug to use *file system events* whenever possible.

You can read more about this in Werkzeug documentation

You can also increase the poll interval when using *stat polling* from the default of 1 second. This will decrease the CPU load at the expense of file edits taking longer to pick up.

This can be set two ways, in the django settings file:

    RUNSERVERPLUS_POLLER_RELOADER_INTERVAL = 5

or as a commad line argument:

    $ python manage.py runserver_plus –reloader-interval 5

Debugger PIN

> The following text about the debugger PIN is taken verbatim from the Werkzeug documentation.
>
> —http://werkzeug.pocoo.org/docs/0.11/debug/#debugger-pin

Starting with Werkzeug 0.11 the debugger is additionally protected by a PIN. This is a security helper to make it less likely for the debugger to be exploited in production as it has happened to people to keep the debugger active. The PIN based authentication is enabled by default.

When the debugger comes up, on first usage it will prompt for a PIN that is printed to the command line. The PIN is generated in a stable way that is specific to the project. In some situations it might be not possible to generate a stable PIN between restarts in which case an explicit PIN can be provided through the environment variable WERKZEUG_DEBUG_PIN. This can be set to a number and will become the PIN. This variable can also be set to the value off to disable the PIN check entirely.

If the PIN is entered too many times incorrectly the server needs to be restarted.

This feature is not supposed to entirely secure the debugger. It's intended to make it harder for an attacker to exploit the debugger. Never enable the debugger in production.

sync_s3

synopsis sync your MEDIA_ROOT and STATIC_ROOT folders to S3

Django command that scans all files in your settings.MEDIA_ROOT and settings.STATIC_ROOT folders, then uploads them to S3 with the same directory structure.

This command can optionally do the following but it is off by default:

- gzip compress any CSS and Javascript files it finds and adds the appropriate 'Content-Encoding' header.
- set a far future 'Expires' header for optimal caching.
- upload only media or static files.
- use any other provider compatible with Amazon S3.
- set other than 'public-read' ACL.

Example Usage

```
# Upload files to S3 into the bucket 'mybucket'
$ ./manage.py sync_s3 mybucket
```

```
# Upload files to S3 into the bucket 'mybucket' and enable gzipping CSS/JS files and␣
↪setting of a far future expires header
$ ./manage.py sync_s3 mybucket --gzip --expires
```

```
# Upload only media files to S3 into the bucket 'mybucket'
$ ./manage.py sync_s3 mybucket  --media-only  # or --static-only
```

```
# Upload only media files to a S3 compatible provider into the bucket 'mybucket' and␣
↪set private file ACLs
$ ./manage.py sync_s3 mybucket  --media-only  --s3host=cs.example.com --acl=private
```

Required libraries and settings

This management command requires the boto library and was tested with version 1.4c:

https://github.com/boto/boto

It also requires an account with Amazon Web Services (AWS) and the AWS S3 keys. Bucket name is required and cannot be empty. The keys and bucket name are added to your settings.py file, for example:

```
# settings.py
AWS_ACCESS_KEY_ID = ''
AWS_SECRET_ACCESS_KEY = ''
AWS_BUCKET_NAME = 'bucket'
```

Optional settings

It is possible to customize sync_s3 directly from django settings file, for example:

```
# settings.py
AWS_S3_HOST = 'cs.example.com'
AWS_DEFAULT_ACL = 'private'
SYNC_S3_PREFIX = 'some_prefix'
FILTER_LIST = 'dir1, dir2'
```

```
AWS_CLOUDFRONT_DISTRIBUTION = 'E27LVI50CSW06W'
SYNC_S3_RENAME_GZIP_EXT = '.gz'
```

sqldiff

synopsis Prints the ALTER TABLE statements for the given appnames.

Django command that scans all models for the given appnames and compares their database schema with the real database tables.

It indicates how columns in the database are different from the SQL that would be generated by Django. This command is not a database migration tool, though it might certainly be of help during migrations. Its purpose is to show the current differences as a way to check or debug your models compared to the real database tables and columns.

Supported Databases

Currently the following databases are supported:

- PostgreSQL
- Sqlite3
- MySQL
- Oracle

Patches to support other databases are welcome! :-)

Exit Codes

Exit status is 0 if inputs are the same, 1 if different, 2 if trouble.

Example Usage

```
# View SQL differences for all installed applications
$ ./manage.py sqldiff -a
```

```
# View SQL differences for all installed applications using text instead of SQL
$ ./manage.py sqldiff -a -t
```

sqlcreate

synopsis Helps you setup your database(s) more easily

Introduction

Stop creating databases by hand. Your settings.py file already contains the correct information, so DRY.

Usage

$ python manage.py sqlcreate [–router=<routername>] | <my_database_shell_command>

It will spit out SQL which you can review (if you want). Ultimately you want to pipe it into the database shell command of your choice.

If there were a good way to ensure that the user in the database settings had the proper permissions, we could submit the commands straight to the database. However, due to the nature of this portion of the project setup, that will never happen.

Example

PostgreSQL

$ ./manage.py sqlcreate [–router=<routername>] | psql -U <db_administrator> -W

MySQL

$ ./manage.py sqlcreate [–router=<routername>] | mysql -u <db_administrator> -p

Known Issues

- CREATE DATABASE is not SQL standard so might not work everywhere.
- When using fallback user is not created and password is not set. But it does try to do a GRANT to the database user.
- Missing options for tablespaces, etc.

sqldsn

synopsis Prints Data Source Name connection string on stdout

Supported Databases

Currently the following databases are supported:

- PostgreSQL (psycopg2 or postgis)
- Sqlite3
- MySQL

Patches to support other databases are welcome! :-)

Exit Codes

Exit status is 0

Example Usage

```
# Prints the DSN for the default database
$ ./manage.py sqldsn
```

```
# Prints the DSN for all databases
$ ./manage.py sqldsn --all
```

```
# Print the DSN for database named 'slave'
$ ./manage.py sqldsn --router=slave
```

```
# Print all DSN styles available for the default database
$ ./manage.py sqldsn --style=all
```

```
# Create .pgpass file for default database by using the quiet option
$ ./manage.py sqldsn -q --style=pgpass > .pgpass
```

validate_templates

synopsis Checks templates on syntax or compile errors.

Options

verbosity

A higher verbosity level will print out all the files that are processed instead of only the ones that contain errors.

break

Do not continue scanning other templates after the first failure.

includes

Use -i (can be used multiple times) to add directories to the TEMPLATE DIRS.

Settings

VALIDATE_TEMPLATES_EXTRA_TEMPLATE_DIRS

You can use *VALIDATE_TEMPLATES_EXTRA_TEMPLATE_DIRS* to include a number of template dirs by default directly from the settings file. This can be useful for situations where TEMPLATE DIRS is dynamically generated or switched in middleware, or when you have other template dirs for external applications like celery, and you want to check those as well.

Usage Example

./manage.py validate_templates

CHAPTER 5

Indices and tables

- search