# South Basics

### Configuring your Django installation

Now you’ve installed South system-wide, you’ll need to configure Django to use it. Doing so is simple; just edit your settings.py and add 'south' to the end of INSTALLED\_APPS.

If Django doesn’t seem to pick this up, check that you’re not overriding INSTALLED\_APPS elsewhere, and that you can run import south from inside ./manage.py shell with no errors.

Once South is added in, you’ll need to run ./manage.py syncdb to make the South migration-tracking tables (South doesn’t use migrations for its own models, for various reasons).

**The First Migration**

South has several ways of creating migrations; some are automatic, some are manual. As a basic user, you’ll probably use the two automatic ways - --auto and --initial.

--auto looks at the previous migration, works out what’s changed, and creates a migration which applies the differences - for example, if you add a field to a model, --auto will notice this, and make a migration which creates a new column for that field on its model’s table.

However, you’ll notice that --auto needs a previous migration - our new app doesn’t have one. Instead, in this case, we need to use --initial, which will create tables and indexes for all of the models in the app; it’s what you use first, much like syncdb, and --auto is then used afterwards for each change.

$ ./manage.py schemamigration <App> --initial

Creating migrations directory at '/home/andrew/Programs/litret/<App>/migrations'...

Creating \_\_init\_\_.py in '/home/andrew/Programs/litret/<App>/migrations'...

+ Added model <App>.Knight

Created 0001\_initial.py. You can now apply this migration with: ./manage.py migrate <App>

As you can see, that’s created a migrations directory for us, and made a new migration inside it. All we need to do now is apply our new migration:

$ ./manage.py migrate <App>

Running migrations for <App>:

- Migrating forwards to 0001\_initial.

> <App>:0001\_initial

- Loading initial data for <App>.

### Changing the model

So far, we’ve done nothing that syncdb couldn’t accomplish; time to change that (or rather, our model). Let’s add another field to our model:

from django.db import models

class Knight(models.Model):

name = models.CharField(max\_length=100)

of\_the\_round\_table = models.BooleanField()

dances\_whenever\_able = models.BooleanField()

Now, if we weren’t using migrations, making this new column appear on our southtut\_knight table would be annoying at best. However, with South, we need only do two, quick steps: make a migration for the change, then apply it.

First, make the new migration, using the –auto feature:

$ ./manage.py schemamigration southtut --auto

+ Added field dances\_whenever\_able on southtut.Knight

Created 0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able.py. You can now apply this migration with: ./manage.py migrate southtut

(Notice that South has automatically picked a name for this migration; you can instead give migrations custom names by providing it as another argument)

Now, apply it:

$ ./manage.py migrate southtut

Running migrations for southtut:

- Migrating forwards to 0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able.

> southtut:0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able

- Loading initial data for southtut.

With that, our new column is created; again, go and check, you’ll be able to add Knights who can dance whenever they’re able.

### Part 2: Advanced Changes

Now you’ve done a simple change to the model, let’s look at some of the more advanced changes you can do with South.

**Defaults**

Firstly, let’s deal with more tricky column types. In the previous part, we added a BooleanField to the table - this is easy for a database to handle, as it has a default value (of False) specified, so that’s the value that gets used for the column in all of the existing rows.

However, some columns don’t have a default defined. If the column is nullable - that is, null=True - then the existing rows will have NULL in the new column. Otherwise, if you’ve given no default, but the column is NOT NULL (i.e. null=False, the default), there’s no value the database can put in the new column, and so you won’t be able to reliably add the column [[1]](http://south.readthedocs.org/en/latest/tutorial/part2.html" \l "id2).

|  |  |
| --- | --- |
| [[1]](http://south.readthedocs.org/en/latest/tutorial/part2.html" \l "id1) | Some database backends will let you add the column anyway if the table is empty, while some will refuse outright in this scenario. |

If South detects such a situation, it will pop up and ask you what to do; let’s make it do so.

First, change your model to add a new field that has no default, but is also not nullable:

from django.db import models

class Knight(models.Model):

name = models.CharField(max\_length=100)

of\_the\_round\_table = models.BooleanField()

dances\_whenever\_able = models.BooleanField()

shrubberies = models.IntegerField(null=False)

Now, let’s try and get South to automatically generate a migration for that:

./manage.py schemamigration southtut --auto

? The field 'Knight.shrubberies' does not have a default specified, yet is NOT NULL.

? Since you are adding or removing this field, you MUST specify a default

? value to use for existing rows. Would you like to:

? 1. Quit now, and add a default to the field in models.py

? 2. Specify a one-off value to use for existing columns now

? Please select a choice:

South presents you with two options; if you select choice one, the command will quit without doing anything, and you should edit your models.py and add a default to the new field.

If you select choice two, you’ll get a Python prompt, where you should enter the default value you want to use for this migration. The default you enter will only ever be used for the currently-existing rows - this is a good option if you don’t want the field on your model to have a default value.

We’ll select choice two, and use 0 as our default (it is an IntegerField, after all):

? Please select a choice: 2

? Please enter Python code for your one-off default value.

? The datetime module is available, so you can do e.g. datetime.date.today()

>>> 0

+ Added field shrubberies on southtut.Knight

Created 0003\_auto\_\_add\_field\_knight\_shrubberies.py. You can now apply this migration with: ./manage.py migrate southtut

If you look at the generated migration, you’ll see that there’s a default specified for the new field, so your database won’t cry. Finish off by running the migration:

$ ./manage.py migrate southtut

Running migrations for southtut:

- Migrating forwards to 0003\_auto\_\_add\_field\_knight\_shrubberies.

> southtut:0003\_auto\_\_add\_field\_knight\_shrubberies

- Loading initial data for southtut.

**Uniques**

As well as detecting new fields (and also ones you’ve removed), South also detects most changes to fields, including changing their unique attributes.

First, let’s make our Knights have unique names:

from django.db import models

class Knight(models.Model):

name = models.CharField(max\_length=100, unique=True)

of\_the\_round\_table = models.BooleanField()

dances\_whenever\_able = models.BooleanField()

shrubberies = models.IntegerField(null=False)

Run the automatic migration creator:

$ ./manage.py schemamigration --auto southtut

+ Added unique constraint for ['name'] on southtut.Knight

Created 0004\_auto\_\_add\_unique\_knight\_name.py. You can now apply this migration with: ./manage.py migrate southtut

As you can see, it’s detected the change in unique; you can now apply it:

$ ./manage.py migrate southtut

Running migrations for southtut:

- Migrating forwards to 0004\_auto\_\_add\_unique\_knight\_name.

> southtut:0004\_auto\_\_add\_unique\_knight\_name

- Loading initial data for southtut.

South also detects changes to unique\_together in your model’s Meta in the same way.

**ManyToMany fields**

South should automatically detect ManyToMany fields; when you add the field, South will create the table the ManyToMany represents, and when you remove the field, the table will be deleted.

The one exception to this is when you have a ‘through model’ (i.e. you’re using the through= option) - since the table for the model is already created when the model is detected, South does nothing with these types of ManyToMany fields.

**Custom fields**

If you’ve looked closely at the migration files, you’ll see that South stores field definitions by storing their class, and the arguments that need to be passed to the field’s constructor.

Since Python offers no way to get the arguments used in a class’ constructor directly, South uses something called the *model introspector* to work out what arguments fields were passed. This knows what variables the arguments are stored into on the field, and using this knowledge, can reconstruct the arguments directly.

Because custom fields (either those written by you, or included with third-party apps) are all different, South can’t work out how to get their arguments without extra help, so if you try to add, change or remove custom fields, South will bail out and say that you need to give it rules for your custom fields; this topic is covered in detail in *[Custom Fields](http://south.readthedocs.org/en/latest/customfields.html" \l "custom-fields)*.

### Part 3: Advanced Commands and Data Migrations

### Iteratively working on a migration

Sometimes, you’ll find that you’ve made model changes that need to be further refined. Say you define this model:

class Group(models.Model):

name = models.TextField(verbose\_name="Name")

facebook\_page\_\_id = models.CharField(max\_length=255)

and you’ve created and applied this migration:

./manage.py schemamigration southtut --auto

./manage.py migrate southtut

You then notice two things: One, name should really be a CharField, not a TextField; and facebook\_page\_\_id contains double underscores where there should be a single one. You can fix these issues in your model, and then run:

./manage.py schemamigration southtut --auto --update

+ Added model southtut.Group

Migration to be updated, 0026\_auto\_\_add\_group, is already applied, rolling it back now...

previous\_migration: 0025\_auto\_\_foo (applied: 2012-05-25 21:20:47)

Running migrations for southtut:

- Migrating backwards to just after 0025\_auto\_\_foo.

< partner:0026\_auto\_\_add\_group

Updated 0026\_auto\_\_add\_group.py. You can now apply this migration with: ./manage.py migrate southtut

What happened here is that South removed the most recent migration, which created the model, but included the mistakes that were made, and replaced it with a new migration that includes the latest corrections made to the model.

It also noticed that the migration had already been applied, and automatically rolled it back for you. You can now apply the latest version of the migration to create the correct version of the model:

./manage.py migrate southtut

You may repeat this process as often as required to iron out any issues and come up with the final database changes required; which you can then publish, neatly packed into a single migration.

### Listing current migrations

It can be very useful to know what migrations you currently have applied, and which ones are available. For this reason, there’s ./manage.py migrate --list.

Run against our project from before, we get:

$ ./manage.py migrate --list

southtut

(\*) 0001\_initial

(\*) 0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able

(\*) 0003\_auto\_\_add\_field\_knight\_shrubberies

(\*) 0004\_auto\_\_add\_unique\_knight\_name

The output has an asterisk (\*) next to a migration name if it has been applied, and an empty space ( ) if not [[1]](http://south.readthedocs.org/en/latest/tutorial/part3.html" \l "id2).

If you have a lot of apps or migrations, you can also specify an app name to show just the migrations from that app.

|  |  |
| --- | --- |
| [[1]](http://south.readthedocs.org/en/latest/tutorial/part3.html" \l "id1) | An interesting side effect of this is that you can run the command ./manage.py migrate --list |grep -v "\*" to see which migrations are unapplied, and need running. |

### Data migrations

The previous parts have only covered schema migrations - migrations which change the layout of your columns and indexes. There’s also another kind of migration, the so-called data migration.

Data migrations are used to change the data stored in your database to match a new schema, or feature. For example, if you’ve been storing passwords in plain text [[2]](http://south.readthedocs.org/en/latest/tutorial/part3.html" \l "id4), and you’re moving to salted and hashed passwords, you might have these three steps (where each step corresponds to a migration):

* Create two new columns, password\_salt and password\_hash (a schema migration).
* Using the contents of the old password column, calculate salts and hashes for each user (a data migration)
* Remove the old password column (a schema migration).

|  |  |
| --- | --- |
| [[2]](http://south.readthedocs.org/en/latest/tutorial/part3.html" \l "id3) | If you’re actually storing passwords in plaintext, please convert. Now. |

The first and last migrations you already know how to do; make the relevant changes in the models.py file, and run ./manage.py schemamigration --auto myapp. Remember that you need to add the two columns separately to deleting the old column, as otherwise the old column won’t be around for us to get data out of, and you’ll have lost all your users’ passwords [[3]](http://south.readthedocs.org/en/latest/tutorial/part3.html" \l "id6).

|  |  |
| --- | --- |
| [[3]](http://south.readthedocs.org/en/latest/tutorial/part3.html" \l "id5) | Always, always, backup your database before doing any kind of potentially destructive migration. One time, it will go wrong. |

Let’s follow a real example. Make a new app, and call it southtut2. Add it to INSTALLED\_APPS, and then give it this model:

from django.db import models

class User(models.Model):

username = models.CharField(max\_length=255)

password = models.CharField(max\_length=60)

name = models.TextField()

Make an initial migration for it, apply it, and then add a record:

$ ./manage.py schemamigration --initial southtut2

Creating migrations directory at '/home/andrew/Programs/litret/southtut2/migrations'...

Creating \_\_init\_\_.py in '/home/andrew/Programs/litret/southtut2/migrations'...

+ Added model southtut2.User

Created 0001\_initial.py. You can now apply this migration with: ./manage.py migrate southtut2

$ ./manage.py migrate southtut2

Running migrations for southtut2:

- Migrating forwards to 0001\_initial.

> southtut2:0001\_initial

- Loading initial data for southtut2.

$ ./manage.py shell

In [1]: from southtut2.models import User

In [2]: User.objects.create(username="andrew", password="ihopetheycantseethis", name="Andrew Godwin")

Out[2]: <User: User object>

In [3]: User.objects.get(id=1).password

Out[3]: u'ihopetheycantseethis'

As you can see, the password is clearly visible, which isn’t good. Let’s move to password hashing, while keeping everyone’s password valid. Firstly, modify the model so it looks like this:

from django.db import models

import sha

class User(models.Model):

username = models.CharField(max\_length=255)

password = models.CharField(max\_length=60)

password\_salt = models.CharField(max\_length=8, null=True)

password\_hash = models.CharField(max\_length=40, null=True)

name = models.TextField()

def check\_password(self, password):

return sha.sha(self.password\_salt + password).hexdigest() == self.password\_hash

Make a schema migration that will create our two new columns (notice that they’ve both been added as null=True; once they have data, we’ll alter them to be null=False):

$ ./manage.py schemamigration southtut2 --auto

+ Added field password\_salt on southtut2.User

+ Added field password\_hash on southtut2.User

Created 0002\_auto\_\_add\_field\_user\_password\_salt\_\_add\_field\_user\_password\_hash.py. You can now apply this migration with: ./manage.py migrate southtut2

Now, the second migration is more interesting. Firstly, we need to create a skeleton data migration (unlike schema migrations, South can’t write these for you):

$ ./manage.py datamigration southtut2 hash\_passwords

Created 0003\_hash\_passwords.py.

If you open up the file, you’ll see that South has made the shell of a migration; the models definitions are there, the forwards() and backwards() functions are there, but there’s no code in either. We’ll write some code to port the passwords over in the forwards function:

def forwards(self, orm):

import random, sha, string

for user in orm.User.objects.all():

user.password\_salt = "".join([random.choice(string.letters) for i in range(8)])

user.password\_hash = sha.sha(user.password\_salt + user.password).hexdigest()

user.save()

Notice that we use orm.User to access the User model - this gives us the version of User from when this migration was created, so if we want to run the migration in future, it won’t get a completely different, new, User model.

If you want to access models from other apps in your data migration, use a syntax like orm['contenttypes.ContentType']. Models will be available if you can somehow get to them via ForeignKey or ManyToMany traversal from your app’s models; if you want to freeze other models, simply pass --freeze appname on the datamigration command line.

We should also raise an error in the backwards() method, since this process is by its very nature irreversible:

def backwards(self, orm):

raise RuntimeError("Cannot reverse this migration.")

That looks good. Finally, remove the password field from your model, and run schemamigration one last time to make a migration to remove that field:

$ ./manage.py schemamigration southtut2 --auto

? The field 'User.password' does not have a default specified, yet is NOT NULL.

? Since you are adding or removing this field, you MUST specify a default

? value to use for existing rows. Would you like to:

? 1. Quit now, and add a default to the field in models.py

? 2. Specify a one-off value to use for existing columns now

? Please select a choice: 2

? Please enter Python code for your one-off default value.

? The datetime module is available, so you can do e.g. datetime.date.today()

>>> ""

- Deleted field password on southtut2.User

Created 0004\_auto\_\_del\_field\_user\_password.py. You can now apply this migration with: ./manage.py migrate southtut2

Notice that South is asking for a default value for password; if you were to reverse this migration, it tries to re-add the password column, and thus needs either a default value or for the field to be null=True. Here, I’ve fed it the empty string, as that’s a reasonable default in this case.

Finally, let’s apply all three migrations:

$ ./manage.py migrate southtut2

Running migrations for southtut2:

- Migrating forwards to 0004\_auto\_\_del\_field\_user\_password.

> southtut2:0002\_auto\_\_add\_field\_user\_password\_salt\_\_add\_field\_user\_password\_hash

> southtut2:0003\_hash\_passwords

> southtut2:0004\_auto\_\_del\_field\_user\_password

- Loading initial data for southtut2.

Looks good - we’ve added the new columns, migrated the passwords over, and then deleted the old column. Let’s check our data was preserved:

$ ./manage.py shell

In [1]: from southtut2.models import User

In [2]: User.objects.get(id=1).check\_password("ihopetheycantseethis")

Out[2]: True

In [3]: User.objects.get(id=1).check\_password("fakepass")

Out[3]: False

That looks like a successful data migration!

You can do a lot more with this inside a data migration; any model can be available to you. The only caveat is that you won’t have access to any custom methods or managers on your models, as they’re not preserved as part of the freezing process (there’s no way to do this generally); you’ll have to copy any code you want into the migration itself. Feel free to make them methods on the Migration class; South ignores everything apart from forwards and backwards.

# First steps with Celery 3.0.1

Celery is a simple, flexible and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It’s a task queue with focus on real-time processing, while also supporting task scheduling.

### Configuring your Django project to use Celery

You need four simple steps to use celery with your Django project.

1. Install the django-celery library:
2. $ pip install django-celery
3. Add the following lines to settings.py:
4. import djcelery
5. djcelery.setup\_loader()
6. Add djcelery to INSTALLED\_APPS.
7. Create the celery database tables.

If you are using [south](http://pypi.python.org/pypi/South/) for schema migrations, you’ll want to:

$ python manage.py migrate djcelery

For those who are not using south, a normal syncdb will work:

$ python manage.py syncdb

By default Celery uses [RabbitMQ](http://www.rabbitmq.com/) as the broker, but there are several alternatives to choose from, see [Choosing a Broker](http://docs.celeryproject.org/en/latest/getting-started/first-steps-with-celery.html" \l "celerytut-broker).

All settings mentioned in the Celery documentation should be added to your Django project’s settings.py module. For example you can configure the [BROKER\_URL](http://docs.celeryproject.org/en/latest/configuration.html" \l "std:setting-BROKER_URL) setting to specify what broker to use:

BROKER\_URL = 'amqp://guest:guest@localhost:5672/'

That’s it.

### Special note for mod\_wsgi users

If you’re using mod\_wsgi to deploy your Django application you need to include the following in your .wsgi module:

import djcelery

djcelery.setup\_loader()

### Defining and calling tasks

Tasks are defined by wrapping functions in the @task decorator. It is a common practice to put these in their own module named tasks.py, and the worker will automatically go through the apps in INSTALLED\_APPS to import these modules.

For a simple demonstration create a new Django app called celerytest. To create this app you need to be in the directory of your Django project where manage.py is located and execute:

$ python manage.py startapp celerytest

Next you have to add the new app to INSTALLED\_APPS so that your Django project recognizes it. This setting is a tuple/list so just append celerytest as a new element at the end

INSTALLED\_APPS = (

...,

'djcelery',

'celerytest',

)

After the new app has been created and added to INSTALLED\_APPS, you can define your tasks by creating a new file called celerytest/tasks.py:

from celery import task

@task()

def add(x, y):

return x + y

Our example task is pretty pointless, it just returns the sum of two arguments, but it will do for demonstration, and it is referred to in many parts of the Celery documentation.

Relative Imports

You have to be consistent in how you import the task module, e.g. if you have project.app in INSTALLED\_APPS then you also need to import the tasks from project.app or else the names of the tasks will be different.

In a production environment you will want to run the worker in the background as a daemon - see [Running the worker as a daemon](http://docs.celeryproject.org/en/latest/tutorials/daemonizing.html" \l "daemonizing) - but for testing and development it is useful to be able to start a worker instance by using the celery worker manage command, much as you would use Django’s runserver:

$ python manage.py celery worker --loglevel=info

For a complete listing of the command line options available, use the help command:

$ python manage.py celery help

### Calling our task

Now that the worker is running, open up a new python interactive shell with python manage.py shell to actually call the task you defined:

>>> from celerytest.tasks import add

>>> add.delay(2, 2)

Note that if you open a regular python shell by simply running python you will need to import your Django application’s settings by running:

# Replace 'myproject' with your project's name

>>> from myproject import settings

The delay method used above is a handy shortcut to the apply\_async method which enables you to have greater control of the task execution. To read more about calling tasks, including specifying the time at which the task should be processed see [Calling Tasks](http://docs.celeryproject.org/en/latest/userguide/calling.html" \l "guide-calling).

Note

Tasks need to be stored in a real module, they can’t be defined in the python shell or IPython/bpython. This is because the worker server must be able to import the task function.

The task should now be processed by the worker you started earlier, and you can verify that by looking at the worker’s console output.

Calling a task returns an [AsyncResult](http://docs.celeryproject.org/en/latest/reference/celery.result.html" \l "celery.result.AsyncResult) instance, which can be used to check the state of the task, wait for the task to finish or get its return value (or if the task failed, the exception and traceback).

By default django-celery stores this state in the Django database. You may consider choosing an alternate result backend or disabling states alltogether (see [Result Backends](http://docs.celeryproject.org/en/latest/userguide/tasks.html" \l "task-result-backends)).

To demonstrate how the results work call the task again, but this time keep the result instance returned:

>>> result = add.delay(4, 4)

>>> result.ready() # returns True if the task has finished processing.

False

>>> result.result # task is not ready, so no return value yet.

None

>>> result.get() # Waits until the task is done and returns the retval.

8

>>> result.result # direct access to result, doesn't re-raise errors.

8

>>> result.successful() # returns True if the task didn't end in failure.

True

If the task raises an exception, the return value of result.successful() will be False, and result.result will contain the exception instance raised by the task.

# Using django-tagging in Django Project

Install django-tagging application inside your virtual environment.

pip install django-tagging

After installing the application, add tagging inside the INSTALLED\_APPS of your project settings.

settings.py

pip install django-tagging

|  |  |
| --- | --- |
| 1  2  3  4  5 | INSTALLED\_APPS = (      ...      'tagging',      ...  ) |

Inside your model, add a TagField which contains all your tags.

models.py

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ...  from tagging.fields import TagField  ...    class Session(models.Model):      ...      tags = TagField()      ... |

Make sure to run syncdb to update your database.

Inside your template:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {% load tagging\_tags %}        {% tag\_cloud\_for\_model session\_app.Session as tags with steps=20 min\_count=1 distribution=log %}      {% for tag in tags %}      <font style="font-size:{{ tag.font\_size|add:"10" }}pt">          <a href="" title="{{ tag.count }}">{{ tag.name }}</a>      </font>      {% endfor %} |

This will list all the tags in the model session\_app.Session along with their ‘weighting’. You should change the model to the one you are using in your project in the form .. In this example, the weighting varies by twenty degrees, corresponding to the steps=20 declaration.