A Peek Behind the Curtains: How Django Auto-Generates the Initial Setup

Who Am I?

- Name: Marcus Willock
- Handle: crazcalm (Github and Twitter)
- About Me: I am a python dev that dreams of getting more sleep #WorksInTheCityButLivesInJersey

Few Assumptions

- I assume you all know what Django is?
- I assume that at some point in time you have gone through a Django tutorial.
- Note: If you do not fall under these assumptions, you will still understand the talk just fine.

Django: The Beginning

Writing your first Django app, part 1

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic poll application.

It'll consist of two parts:

- A public site that lets people view polls and vote in them.
- An admin site that lets you add, change, and delete polls.

We'll assume you have Django installed already. You can tell Django is installed and which version by running the following command in a shell prompt (indicated by the $ prefix):

```
$ python -m django --version
```

If Django is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named django".

Creating a project

If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django project – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, **cd** into a directory where you'd like to store your code, then run the following command:

```
$ django-admin startproject mysite
```

This will create a **mysite** directory in your current directory. If it didn't work, see Problems running django-admin.

Let's look at what **`startproject`** created:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

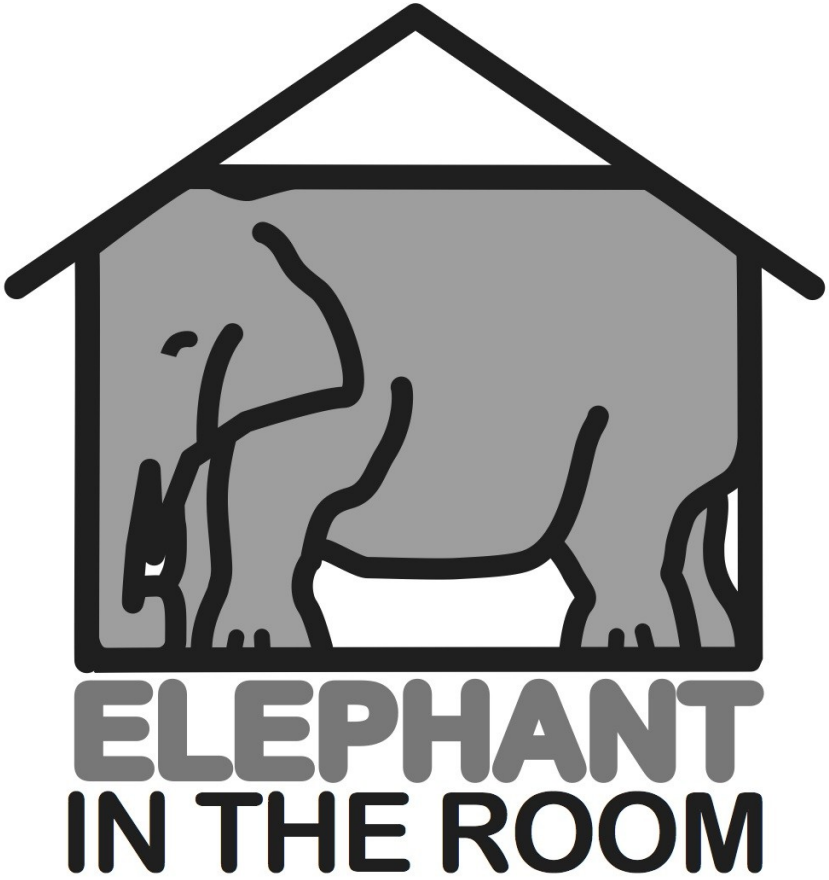
Quick Review:

```
$ django-admin startproject mysite
```

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

In this talk, we are going to figure out how the django-admin command creates that directory and those files.

But First!



Back to the matter at hand

```
$ django-admin startproject mysite
```

Django Admin Source Code:

```
1   #!/usr/bin/env python
2   from django.core import management
3
4   if __name__ == "__main__":
5       management.execute_from_command_line()
```

django.core.management.__init__.py

```
377
378 def execute_from_command_line(argv=None):
379     """Run a ManagementUtility."""
380     utility = ManagementUtility(argv)
381     utility.execute()
```

Same file...

```
151  class ManagementUtility:
152      """
153      Encapsulate the logic of the django-admin and manage.py utilities.
154      """
155      def __init__(self, argv=None):
156          self.argv = argv or sys.argv[:]
157          self.prog_name = os.path.basename(self.argv[0])
158          if self.prog_name == '__main__.py':
159              self.prog_name = 'python -m django'
160          self.settings_exception = None
```

Variable Values:

- sys.argv = [path to django-admin, startproject, mysite]
- self.prog_name = django-admin

django.core.management.__init__.py

```
377
378 def execute_from_command_line(argv=None):
379     """Run a ManagementUtility."""
380     utility = ManagementUtility(argv)
381     utility.execute()
```

Execute Method:

```python
    def execute(self):
        """
        Given the command-line arguments, figure out which subcommand is being
        run, create a parser appropriate to that command, and run it.
        """
        try:
            subcommand = self.argv[1]
        except IndexError:
            subcommand = 'help'  # Display help if no arguments were given.

        # Preprocess options to extract --settings and --pythonpath.
        # These options could affect the commands that are available, so they
        # must be processed early.
        parser = CommandParser(usage='%(prog)s subcommand [options] [args]', add_help=False, allow_abbrev=False)
        parser.add_argument('--settings')
        parser.add_argument('--pythonpath')
        parser.add_argument('args', nargs='*')  # catch-all
        try:
            options, args = parser.parse_known_args(self.argv[2:])
            handle_default_options(options)
        except CommandError:
            pass  # Ignore any option errors at this point.

        try:
            settings.INSTALLED_APPS
        except ImproperlyConfigured as exc:
            self.settings_exception = exc
        except ImportError as exc:
            self.settings_exception = exc

        if settings.configured:
            # Start the auto-reloading dev server even if the code is broken.
            # The hardcoded condition is a code smell but we can't rely on a
            # flag on the command class because we haven't located it yet.
            if subcommand == 'runserver' and '--noreload' not in self.argv:
                try:
                    autoreload.check_errors(django.setup)()
                except Exception:
                    # The exception will be raised later in the child process
                    # started by the autoreloader. Pretend it didn't happen by
                    # loading an empty list of applications.
                    apps.all_models = defaultdict(OrderedDict)
                    apps.app_configs = OrderedDict()
                    apps.apps_ready = apps.models_ready = apps.ready = True

                    # Remove options not compatible with the built-in runserver
                    # (e.g. options for the contrib.staticfiles' runserver).
                    # Changes here require manually testing as described in
                    # #27522.
                    _parser = self.fetch_command('runserver').create_parser('django', 'runserver')
                    _options, _args = _parser.parse_known_args(self.argv[2:])
                    for _arg in _args:
                        self.argv.remove(_arg)

            # In all other cases, django.setup() is required to succeed.
            else:
                django.setup()

        self.autocomplete()

        if subcommand == 'help':
            if '--commands' in args:
                sys.stdout.write(self.main_help_text(commands_only=True) + '\n')
            elif not options.args:
                sys.stdout.write(self.main_help_text() + '\n')
            else:
                self.fetch_command(options.args[0]).print_help(self.prog_name, options.args[0])
        # Special-cases: We want 'django-admin --version' and
        # 'django-admin --help' to work, for backwards compatibility.
        elif subcommand == 'version' or self.argv[1:] == ['--version']:
            sys.stdout.write(django.get_version() + '\n')
        elif self.argv[1:] in (['--help'], ['-h']):
            sys.stdout.write(self.main_help_text() + '\n')
        else:
            self.fetch_command(subcommand).run_from_argv(self.argv)
```

- 75 Lines of code
- But… Most of it does not apply to us!

Continued.

```
301     def execute(self):
302         """
303         Given the command-line arguments, figure out which subcommand is being
304         run, create a parser appropriate to that command, and run it.
305         """
306         try:
307             subcommand = self.argv[1]
308         except IndexError:
309             subcommand = 'help'  # Display help if no arguments were given.
```

subcommand = “startproject”

Last line of execute method

```
374         else:
375             self.fetch_command(subcommand).run_from_argv(self.argv)
```

- Subcommand = “startproject”
- Self.argv = [path to django-admin, startproject, mysite]

fetch_command Method

```python
    def fetch_command(self, subcommand):
        """
        Try to fetch the given subcommand, printing a message with the
        appropriate command called from the command line (usually
        "django-admin" or "manage.py") if it can't be found.
        """
        # Get commands outside of try block to prevent swallowing exceptions
        commands = get_commands()
        try:
            app_name = commands[subcommand]
        except KeyError:
            if os.environ.get('DJANGO_SETTINGS_MODULE'):
                # If `subcommand` is missing due to misconfigured settings, the
                # following line will retrigger an ImproperlyConfigured exception
                # (get_commands() swallows the original one) so the user is
                # informed about it.
                settings.INSTALLED_APPS
            else:
                sys.stderr.write("No Django settings specified.\n")
            possible_matches = get_close_matches(subcommand, commands)
            sys.stderr.write('Unknown command: %r' % subcommand)
            if possible_matches:
                sys.stderr.write(' Did you mean %s?' % possible_matches[0])
            sys.stderr.write("\nType '%s help' for usage.\n" % self.prog_name)
            sys.exit(1)
        if isinstance(app_name, BaseCommand):
            # If the command is already loaded, use it directly.
            klass = app_name
        else:
            klass = load_command_class(app_name, subcommand)
        return klass
```

- About 30 lines
- We do not care about most of theses lines, but here is where it gets interesting!

```
195     def fetch_command(self, subcommand):
196         """
197         Try to fetch the given subcommand, printing a message with the
198         appropriate command called from the command line (usually
199         "django-admin" or "manage.py") if it can't be found.
200         """
201         # Get commands outside of try block to prevent swallowing exceptions
202         commands = get_commands()
203         try:
204             app_name = commands[subcommand]
```

get_commands?
commands[subcommand]?

What is this MAGIC?!

```
41  def get_commands():
42      """
43      Return a dictionary mapping command names to their callback applications.
44
45      Look for a management.commands package in django.core, and in each
46      installed application -- if a commands package exists, register all
47      commands in that package.
48
49      Core commands are always included. If a settings module has been
50      specified, also include user-defined commands.
51
52      The dictionary is in the format {command_name: app_name}. Key-value
53      pairs from this dictionary can then be used in calls to
54      load_command_class(app_name, command_name)
```

get_commands Source code

```
63      commands = {name: 'django.core' for name in find_commands(__path__[0])}
64
65      if not settings.configured:
66          return commands
67
68      for app_config in reversed(list(apps.get_app_configs())):
69          path = os.path.join(app_config.path, 'management')
70          commands.update({name: app_config.name for name in find_commands(path)})
71
72      return commands
```

===> find_commands(__path__[0]) <===

We have struck GOLD!

```
20  def find_commands(management_dir):
21      """
22      Given a path to a management directory, return a list of all the command
23      names that are available.
24      """
25      command_dir = os.path.join(management_dir, 'commands')
26      return [name for _, name, is_pkg in pkgutil.iter_modules([command_dir])
27              if not is_pkg and not name.startswith('_')]
```

command_dir = python_path/site-packages/django/core/management/commands

pope1ni and **timgraham** Fixed #30159 -- Removed unneeded use of OrderedDict. ··· 1 Latest commit `24b82cd` 2 days ago

..

check.py	Removed parser.add_arguments() arguments that match the defaults.	7 months ago
compilemessages.py	Fixed #29973 -- Added compilemessages --ignore option.	9 days ago
createcachetable.py	Removed parser.add_arguments() arguments that match the defaults.	7 months ago
dbshell.py	Removed parser.add_arguments() arguments that match the defaults.	7 months ago
diffsettings.py	Fixed #30057 -- Fixed diffsettings ignoring custom configured settings.	25 days ago
dumpdata.py	Fixed #30159 -- Removed unneeded use of OrderedDict.	5 hours ago
flush.py	Removed parser.add_arguments() arguments that match the defaults.	7 months ago
inspectdb.py	Fixed #30159 -- Removed unneeded use of OrderedDict.	5 hours ago

Ready to see the startproject.py source code?!

```python
from django.core.management.templates import TemplateCommand

from ..utils import get_random_secret_key


class Command(TemplateCommand):
    help = (
        "Creates a Django project directory structure for the given project "
        "name in the current directory or optionally in the given directory."
    )
    missing_args_message = "You must provide a project name."

    def handle(self, **options):
        project_name = options.pop('name')
        target = options.pop('directory')

        # Create a random SECRET_KEY to put it in the main settings.
        options['secret_key'] = get_random_secret_key()

        super().handle('project', project_name, target, **options)
```

The Inheritance Tree:

```
6   class Command(TemplateCommand):
```

```
21   class TemplateCommand(BaseCommand):
```

This inheritance tree seems to lead to something much bigger than we were searching for...

The BaseCommand Class

- The BaseCommand class is located in django/core/management/base/py

- For us, the most important part of this class is the doc string where is explains the work flow.

Work Flow:

1. ``django-admin`` or ``manage.py`` loads the command class and calls its ``run_from_argv()`` method.

2. The ``run_from_argv()`` method calls ``create_parser()`` to get an ``ArgumentParser`` for the arguments, parses them, performs any environment changes requested by options like ``pythonpath``, and then calls the ``execute()`` method, passing the parsed arguments.

3. The ``execute()`` method attempts to carry out the command by calling the ``handle()`` method with the parsed arguments; any output produced by ``handle()`` will be printed to standard output and, if the command is intended to produce a block of SQL statements, will be wrapped in ``BEGIN`` and ``COMMIT``.

Lets take a few steps back to the fetch_command function

```
    def fetch_command(self, subcommand):
        """
        Try to fetch the given subcommand, printing a message with the
        appropriate command called from the command line (usually
        "django-admin" or "manage.py") if it can't be found.
        """
        # Get commands outside of try block to prevent swallowing exceptions
        commands = get_commands()
        try:
            app_name = commands[subcommand]
```

fetch_command found the command that we wanted and returned it.

One More Step Back

Last line of execute method

```
374         else:
375             self.fetch_command(subcommand).run_from_argv(self.argv)
```

- Subcommand = “startproject”
- Self.argv = [path to django-admin, startproject, mysite]

We have now made it full circle!

According to the BaseCommand

- fetch_command(subcommand).run_from_argv(self.argv)
- Which will call an `execute()` Method
- Which will call an `handle()` Method
- And bring us to startproject.py::Command::handle()
  - See Source Code in the following slide

```python
from django.core.management.templates import TemplateCommand

from ..utils import get_random_secret_key


class Command(TemplateCommand):
    help = (
        "Creates a Django project directory structure for the given project "
        "name in the current directory or optionally in the given directory."
    )
    missing_args_message = "You must provide a project name."

    def handle(self, **options):
        project_name = options.pop('name')      ==> “mysite”
        target = options.pop('directory')       ==> None

        # Create a random SECRET_KEY to put it in the main settings.
        options['secret_key'] = get_random_secret_key()

        super().handle('project', project_name, target, **options)
```

This is where the work is done

```
20
21  class TemplateCommand(BaseCommand):
22      """
23      Copy either a Django application layout template or a Django project
24      layout template into the specified directory.
25
```

- The handle method copies template files from django/conf/project_template, fills in the needed information with the project name and writes all those files to disk.

In Summary

- `django-admin startproject mysite` looks up the subcommand (“startproject”) from a directory filled with subcommand files.
- It then calls a handle method which loads up all the template files and passes in “mysite” as the project name.
- Then it writes everything to disk.

The End