2.13.9 Writing the code for your ticket

Next we'll be adding the `make_toast()` function.

Navigate to the `django/` folder and open the `shortcuts.py` file. At the bottom, add:

```
def make_toast():
    return "toast"
```

Now we need to make sure that the test we wrote earlier passes, so we can see whether the code we added is working correctly. Again, navigate to the Django `tests/` directory and run:

```
$ ./runtests.py shortcuts
```

Everything should pass. If it doesn't, make sure you correctly added the function to the correct file.

2.13.10 Running Django's test suite for the second time

Once you've verified that your changes and test are working correctly, it's a good idea to run the entire Django test suite to verify that your change hasn't introduced any bugs into other areas of Django. While successfully passing the entire test suite doesn't guarantee your code is bug free, it does help identify many bugs and regressions that might otherwise go unnoticed.

To run the entire Django test suite, `cd` into the Django `tests/` directory and run:

```
$ ./runtests.py
```

2.13.11 Writing Documentation

This is a new feature, so it should be documented. Open the file `docs/topics/http/shortcuts.txt` and add the following at the end of the file:

```
``make_toast()``
================

.. function:: make_toast()

.. versionadded:: 2.2

Returns ``'toast'``.
```

Since this new feature will be in an upcoming release it is also added to the release notes for the next version of Django. Open the release notes for the latest version in `docs/releases/`, which at time of writing is `2.2.txt`. Add a note under the "Minor Features" header:

```
:mod:`django.shortcuts`
~~~~~~~~~~~~~~~~~~~~~~~

* The new :func:`django.shortcuts.make_toast` function returns ``'toast'``.
```

For more information on writing documentation, including an explanation of what the `versionadded` bit is all about, see Writing documentation. That page also includes an explanation of how to build a copy of the documentation locally, so you can preview the HTML that will be generated.

2.13.12 Previewing your changes

Now it's time to review the changes made in the branch. To stage all the changes ready for commit, run:

```
$ git add --all
```

Then display the differences between your current copy of Django (with your changes) and the revision that you initially checked out earlier in the tutorial with:

```
$ git diff --cached
```

Use the arrow keys to move up and down.

```
diff --git a/django/shortcuts.py b/django/shortcuts.py
index 7ab1df0e9d..8dde9e28d9 100644
--- a/django/shortcuts.py
+++ b/django/shortcuts.py
@@ -156,3 +156,7 @@ def resolve_url(to, *args, **kwargs):

     # Finally, fall back and assume it's a URL
     return to
+
+
+def make_toast():
+    return 'toast'
diff --git a/docs/releases/2.2.txt b/docs/releases/2.2.txt
index 7d85d30c4a..81518187b3 100644
--- a/docs/releases/2.2.txt
+++ b/docs/releases/2.2.txt
@@ -40,6 +40,11 @@ database constraints. Constraints are added to models using the
 Minor features
 --------------
```

(continues on next page)

(continued from previous page)

```
+:mod:`django.shortcuts`
+~~~~~~~~~~~~~~~~~~~~~~
+
+* The new :func:`django.shortcuts.make_toast` function returns ``'toast'``.
+
 :mod:`django.contrib.admin`
 ~~~~~~~~~~~~~~~~~~~~~~~~~~

diff --git a/docs/topics/http/shortcuts.txt b/docs/topics/http/shortcuts.txt
index 7b3a3a2c00..711bf6bb6d 100644
--- a/docs/topics/http/shortcuts.txt
+++ b/docs/topics/http/shortcuts.txt
@@ -271,3 +271,12 @@ This example is equivalent to::
         my_objects = list(MyModel.objects.filter(published=True))
         if not my_objects:
             raise Http404("No MyModel matches the given query.")
+
+``make_toast()``
+================
+
+.. function:: make_toast()
+
+.. versionadded:: 2.2
+
+Returns ``'toast'``.
diff --git a/tests/shortcuts/test_make_toast.py b/tests/shortcuts/test_make_toast.py
new file mode 100644
index 0000000000..6f4c627b6e
--- /dev/null
+++ b/tests/shortcuts/test_make_toast.py
@@ -0,0 +1,7 @@
+from django.shortcuts import make_toast
+from django.test import SimpleTestCase
+
+
+class MakeToastTests(SimpleTestCase):
+    def test_make_toast(self):
+        self.assertEqual(make_toast(), 'toast')
```

When you're done previewing the changes, hit the `q` key to return to the command line. If the diff looked okay, it's time to commit the changes.

2.13.13 Committing the changes

To commit the changes:

```
$ git commit
```

This opens up a text editor to type the commit message. Follow the commit message guidelines and write a message like:

```
Fixed #99999 -- Added a shortcut function to make toast.
```

2.13.14 Pushing the commit and making a pull request

After committing the changes, send it to your fork on GitHub (substitute "ticket_99999" with the name of your branch if it's different):

```
$ git push origin ticket_99999
```

You can create a pull request by visiting the Django GitHub page. You'll see your branch under "Your recently pushed branches". Click "Compare & pull request" next to it.

Please don't do it for this tutorial, but on the next page that displays a preview of the changes, you would click "Create pull request".

2.13.15 Next steps

Congratulations, you've learned how to make a pull request to Django! Details of more advanced techniques you may need are in Working with Git and GitHub.

Now you can put those skills to good use by helping to improve Django's codebase.

More information for new contributors

Before you get too into contributing to Django, there's a little more information on contributing that you should probably take a look at:

- You should make sure to read Django's documentation on claiming tickets and submitting pull requests. It covers Trac etiquette, how to claim tickets for yourself, expected coding style (both for code and docs), and many other important details.
- First time contributors should also read Django's documentation for first time contributors. It has lots of good advice for those of us who are new to helping out with Django.
- After those, if you're still hungry for more information about contributing, you can always browse through the rest of Django's documentation on contributing. It contains a ton of useful information and should be your first source for answering any questions you might have.

Finding your first real ticket

Once you've looked through some of that information, you'll be ready to go out and find a ticket of your own to contribute to. Pay special attention to tickets with the "easy pickings" criterion. These tickets are often much simpler in nature and are great for first time contributors. Once you're familiar with contributing to Django, you can start working on more difficult and complicated tickets.

If you just want to get started already (and nobody would blame you!), try taking a look at the list of easy tickets without a branch and the easy tickets that have branches which need improvement. If you're familiar with writing tests, you can also look at the list of easy tickets that need tests. Remember to follow the guidelines about claiming tickets that were mentioned in the link to Django's documentation on claiming tickets and submitting branches.

What's next after creating a pull request?

After a ticket has a branch, it needs to be reviewed by a second set of eyes. After submitting a pull request, update the ticket metadata by setting the flags on the ticket to say "has patch", "doesn't need tests", etc, so others can find it for review. Contributing doesn't necessarily always mean writing code from scratch. Reviewing open pull requests is also a very helpful contribution. See Triaging tickets for details.

See also

If you're new to Python, you might want to start by getting an idea of what the language is like. Django is 100% Python, so if you've got minimal comfort with Python you'll probably get a lot more out of Django.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend referring the official Python documentation, which provides comprehensive and authoritative information about the language, as well as links to other resources such as a list of books about Python.

CHAPTER

THREE

USING DJANGO

Introductions to all the key parts of Django you'll need to know:

3.1 How to install Django

This document will get you up and running with Django.

3.1.1 Install Python

Django is a Python web framework. See What Python version can I use with Django? for details.

Get the latest version of Python at https://www.python.org/downloads/ or with your operating system's package manager.

> **Python on Windows**
>
> If you are just starting with Django and using Windows, you may find How to install Django on Windows useful.

3.1.2 Install Apache and `mod_wsgi`

If you just want to experiment with Django, skip ahead to the next section; Django includes a lightweight web server you can use for testing, so you won't need to set up Apache until you're ready to deploy Django in production.

If you want to use Django on a production site, use Apache with mod_wsgi. mod_wsgi operates in one of two modes: embedded mode or daemon mode. In embedded mode, mod_wsgi is similar to mod_perl – it embeds Python within Apache and loads Python code into memory when the server starts. Code stays in memory throughout the life of an Apache process, which leads to significant performance gains over other server arrangements. In daemon mode, mod_wsgi spawns an independent daemon process that handles requests. The daemon process can run as a different user than the web server, possibly leading to improved security. The daemon process can be restarted without restarting the entire Apache web server, possibly making refreshing your codebase more seamless. Consult the mod_wsgi documentation to determine which

mode is right for your setup. Make sure you have Apache installed with the mod_wsgi module activated. Django will work with any version of Apache that supports mod_wsgi.

See How to use Django with mod_wsgi for information on how to configure mod_wsgi once you have it installed.

If you can't use mod_wsgi for some reason, fear not: Django supports many other deployment options. One is uWSGI; it works very well with nginx. Additionally, Django follows the WSGI spec (PEP 3333), which allows it to run on a variety of server platforms.

3.1.3 Get your database running

If you plan to use Django's database API functionality, you'll need to make sure a database server is running. Django supports many different database servers and is officially supported with PostgreSQL, MariaDB, MySQL, Oracle and SQLite.

If you are developing a small project or something you don't plan to deploy in a production environment, SQLite is generally the best option as it doesn't require running a separate server. However, SQLite has many differences from other databases, so if you are working on something substantial, it's recommended to develop with the same database that you plan on using in production.

In addition to the officially supported databases, there are backends provided by 3rd parties that allow you to use other databases with Django.

To use another database other than SQLite, you'll need to make sure that the appropriate Python database bindings are installed:

- If you're using PostgreSQL, you'll need the psycopg or psycopg2 package. Refer to the PostgreSQL notes for further details.
- If you're using MySQL or MariaDB, you'll need a DB API driver like `mysqlclient`. See notes for the MySQL backend for details.
- If you're using SQLite you might want to read the SQLite backend notes.
- If you're using Oracle, you'll need to install oracledb, but please read the notes for the Oracle backend for details regarding supported versions of both Oracle and `oracledb`.
- If you're using an unofficial 3rd party backend, please consult the documentation provided for any additional requirements.

And ensure that the following keys in the `'default'` item of the *`DATABASES`* dictionary match your database connection settings:

- *`ENGINE`* – Either `'django.db.backends.sqlite3'`, `'django.db.backends.postgresql'`, `'django.db.backends.mysql'`, or `'django.db.backends.oracle'`. Other backends are also available.
- *`NAME`* – The name of your database. If you're using SQLite, the database will be a file on your computer. In that case, `NAME` should be the full absolute path, including the filename of that file. You don't need to

create anything beforehand; the database file will be created automatically when needed. The default value, `BASE_DIR / 'db.sqlite3'`, will store the file in your project directory.

> **For databases other than SQLite**
>
> If you are not using SQLite as your database, additional settings such as *USER*, *PASSWORD*, and *HOST* must be added. For more details, see the reference documentation for *DATABASES*.
>
> Also, make sure that you've created the database by this point. Do that with "`CREATE DATABASE database_name;`" within your database's interactive prompt.

If you plan to use Django's `manage.py migrate` command to automatically create database tables for your models (after first installing Django and creating a project), you'll need to ensure that Django has permission to create and alter tables in the database you're using; if you plan to manually create the tables, you can grant Django `SELECT`, `INSERT`, `UPDATE` and `DELETE` permissions. After creating a database user with these permissions, you'll specify the details in your project's settings file, see *DATABASES* for details.

If you're using Django's testing framework to test database queries, Django will need permission to create a test database.

3.1.4 Install the Django code

Installation instructions are slightly different depending on whether you're installing a distribution-specific package, downloading the latest official release, or fetching the latest development version.

Installing an official release with `pip`

This is the recommended way to install Django.

1. Install pip. The easiest is to use the standalone pip installer. If your distribution already has `pip` installed, you might need to update it if it's outdated. If it's outdated, you'll know because installation won't work.
2. Take a look at venv. This tool provides isolated Python environments, which are more practical than installing packages systemwide. It also allows installing packages without administrator privileges. The contributing tutorial walks through how to create a virtual environment.
3. After you've created and activated a virtual environment, enter the command:

```
$ python -m pip install Django
```

Installing a distribution-specific package

Check the distribution specific notes to see if your platform/distribution provides official Django packages/installers. Distribution-provided packages will typically allow for automatic installation of dependencies and supported upgrade paths; however, these packages will rarely contain the latest release of Django.

Installing the development version

> **Tracking Django development**
>
> If you decide to use the latest development version of Django, you'll want to pay close attention to the development timeline, and you'll want to keep an eye on the release notes for the upcoming release. This will help you stay on top of any new features you might want to use, as well as any changes you'll need to make to your code when updating your copy of Django. (For stable releases, any necessary changes are documented in the release notes.)

If you'd like to be able to update your Django code occasionally with the latest bug fixes and improvements, follow these instructions:

1. Make sure that you have Git installed and that you can run its commands from a shell. (Enter `git help` at a shell prompt to test this.)
2. Check out Django's main development branch like so:

   ```
   $ git clone https://github.com/django/django.git
   ```

   This will create a directory `django` in your current directory.
3. Make sure that the Python interpreter can load Django's code. The most convenient way to do this is to use a virtual environment and pip. The contributing tutorial walks through how to create a virtual environment.
4. After setting up and activating the virtual environment, run the following command:

   ```
   $ python -m pip install -e django/
   ```

   This will make Django's code importable, and will also make the `django-admin` utility command available. In other words, you're all set!

When you want to update your copy of the Django source code, run the command `git pull` from within the `django` directory. When you do this, Git will download any changes.

3.2 Models and databases

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

3.2.1 Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses *`django.db.models.Model`*.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API; see Making queries.

Quick example

This example model defines a `Person`, which has a `first_name` and `last_name`:

```
from django.db import models


class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and `last_name` are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above `Person` model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Some technical notes:

- The name of the table, `myapp_person`, is automatically derived from some model metadata but can be overridden. See Table names for more details.
- An `id` field is added automatically, but this behavior can be overridden. See Automatic primary key fields.

- The `CREATE TABLE` SQL in this example is formatted using PostgreSQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your settings file.

Using models

Once you have defined your models, you need to tell Django you're going to use those models. Do this by editing your settings file and changing the *INSTALLED_APPS* setting to add the name of the module that contains your `models.py`.

For example, if the models for your application live in the module `myapp.models` (the package structure that is created for an application by the *manage.py startapp* script), *INSTALLED_APPS* should read, in part:

```
INSTALLED_APPS = [
    # ...
    "myapp",
    # ...
]
```

When you add new apps to *INSTALLED_APPS*, be sure to run *manage.py migrate*, optionally making migrations for them first with *manage.py makemigrations*.

Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attributes. Be careful not to choose field names that conflict with the models API like `clean`, `save`, or `delete`.

Example:

```
from django.db import models


class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)


class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Field types

Each field in your model should be an instance of the appropriate *Field* class. Django uses the field class types to determine a few things:

- The column type, which tells the database what kind of data to store (e.g. `INTEGER`, `VARCHAR`, `TEXT`).
- The default HTML widget to use when rendering a form field (e.g. `<input type="text">`, `<select>`).
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

Django ships with dozens of built-in field types; you can find the complete list in the model field reference. You can easily write your own fields if Django's built-in ones don't do the trick; see How to create custom model fields.

Field options

Each field takes a certain set of field-specific arguments (documented in the model field reference). For example, *CharField* (and its subclasses) require a *max_length* argument which specifies the size of the `VARCHAR` database field used to store the data.

There's also a set of common arguments available to all field types. All are optional. They're fully explained in the reference, but here's a quick summary of the most often-used ones:

null

If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

blank

If `True`, the field is allowed to be blank. Default is `False`.

Note that this is different than *null*. *null* is purely database-related, whereas *blank* is validation-related. If a field has *blank=True*, form validation will allow entry of an empty value. If a field has *blank=False*, the field will be required.

choices

A sequence of 2-value tuples, a mapping, an enumeration type, or a callable (that expects no arguments and returns any of the previous formats), to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = [
    ("FR", "Freshman"),
    ("SO", "Sophomore"),
    ("JR", "Junior"),
    ("SR", "Senior"),
    ("GR", "Graduate"),
]
```

> **Note**
>
> A new migration is created each time the order of `choices` changes.

The first element in each tuple is the value that will be stored in the database. The second element is displayed by the field's form widget.

Given a model instance, the display value for a field with `choices` can be accessed using the *`get_FOO_display()`* method. For example:

```
from django.db import models


class Person(models.Model):
    SHIRT_SIZES = {
        "S": "Small",
        "M": "Medium",
        "L": "Large",
    }
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

You can also use enumeration classes to define `choices` in a concise way:

```
from django.db import models


class Runner(models.Model):
    MedalType = models.TextChoices("MedalType", "GOLD SILVER BRONZE")
    name = models.CharField(max_length=60)
    medal = models.CharField(blank=True, choices=MedalType, max_length=10)
```

Further examples are available in the model field reference.

default

The default value for the field. This can be a value or a callable object. If callable it will be called every

time a new object is created.

db_default

The database-computed default value for the field. This can be a literal value or a database function.

If both `db_default` and *Field.default* are set, `default` will take precedence when creating instances in Python code. `db_default` will still be set at the database level and will be used when inserting rows outside of the ORM or when adding a new field in a migration.

help_text

Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

primary_key

If `True`, this field is the primary key for the model.

If you don't specify *primary_key=True* for any fields in your model, Django will automatically add a field to hold the primary key, so you don't need to set *primary_key=True* on any of your fields unless you want to override the default primary-key behavior. For more, see Automatic primary key fields.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one. For example:

```
from django.db import models



class Fruit(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

```
>>> fruit = Fruit.objects.create(name="Apple")
>>> fruit.name = "Pear"
>>> fruit.save()
>>> Fruit.objects.values_list("name", flat=True)
<QuerySet ['Apple', 'Pear']>
```

unique

If `True`, this field must be unique throughout the table.

Again, these are just short descriptions of the most common field options. Full details can be found in the common model field option reference.

Automatic primary key fields

By default, Django gives each model an auto-incrementing primary key with the type specified per app in *AppConfig.default_auto_field* or globally in the *DEFAULT_AUTO_FIELD* setting. For example:

```
id = models.BigAutoField(primary_key=True)
```

If you'd like to specify a custom primary key, specify *primary_key=True* on one of your fields. If Django sees you've explicitly set *Field.primary_key*, it won't add the automatic `id` column.

Each model requires exactly one field to have *primary_key=True* (either explicitly declared or automatically added).

Verbose field names

Each field type, except for *ForeignKey*, *ManyToManyField* and *OneToOneField*, takes an optional first positional argument – a verbose name. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

In this example, the verbose name is `"person's first name"`:

```
first_name = models.CharField("person's first name", max_length=30)
```

In this example, the verbose name is `"first name"`:

```
first_name = models.CharField(max_length=30)
```

ForeignKey, *ManyToManyField* and *OneToOneField* require the first argument to be a model class, so use the *verbose_name* keyword argument:

```
poll = models.ForeignKey(
    Poll,
    on_delete=models.CASCADE,
    verbose_name="the related poll",
)
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(
    Place,
    on_delete=models.CASCADE,
    verbose_name="related place",
)
```

The convention is not to capitalize the first letter of the *verbose_name*. Django will automatically capitalize the first letter where it needs to.

Relationships

Clearly, the power of relational databases lies in relating tables to each other. Django offers ways to define the three most common types of database relationships: many-to-one, many-to-many and one-to-one.

Many-to-one relationships

To define a many-to-one relationship, use *django.db.models.ForeignKey*. You use it just like any other *Field* type: by including it as a class attribute of your model.

ForeignKey requires a positional argument: the class to which the model is related.

For example, if a `Car` model has a `Manufacturer` – that is, a `Manufacturer` makes multiple cars but each `Car` only has one `Manufacturer` – use the following definitions:

```
from django.db import models


class Manufacturer(models.Model):
    # ...
    pass


class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

You can also create recursive relationships (an object with a many-to-one relationship to itself) and relationships to models not yet defined; see the model field reference for details.

It's suggested, but not required, that the name of a *ForeignKey* field (`manufacturer` in the example above) be the name of the model, lowercase. You can call the field whatever you want. For example:

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(
        Manufacturer,
        on_delete=models.CASCADE,
    )
    # ...
```

See also

ForeignKey fields accept a number of extra arguments which are explained in the model field reference. These options help define how the relationship should work; all are optional.

For details on accessing backwards-related objects, see the Following relationships backward example.

For sample code, see the Many-to-one relationship model example.

Many-to-many relationships

To define a many-to-many relationship, use *ManyToManyField*. You use it just like any other *Field* type: by including it as a class attribute of your model.

ManyToManyField requires a positional argument: the class to which the model is related.

For example, if a `Pizza` has multiple `Topping` objects – that is, a `Topping` can be on multiple pizzas and each `Pizza` has multiple toppings – here's how you'd represent that:

```
from django.db import models


class Topping(models.Model):
    # ...
    pass


class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

As with *ForeignKey*, you can also create recursive relationships (an object with a many-to-many relationship to itself) and relationships to models not yet defined.

It's suggested, but not required, that the name of a *ManyToManyField* (`toppings` in the example above) be a plural describing the set of related model objects.

It doesn't matter which model has the *ManyToManyField*, but you should only put it in one of the models – not both.

Generally, *ManyToManyField* instances should go in the object that's going to be edited on a form. In the above example, `toppings` is in `Pizza` (rather than `Topping` having a `pizzas` *ManyToManyField*) because it's more natural to think about a pizza having toppings than a topping being on multiple pizzas. The way it's set up above, the `Pizza` form would let users select the toppings.

See also

See the Many-to-many relationship model example for a full example.

ManyToManyField fields also accept a number of extra arguments which are explained in the model field

reference. These options help define how the relationship should work; all are optional.

Extra fields on many-to-many relationships

When you're only dealing with many-to-many relationships such as mixing and matching pizzas and toppings, a standard *ManyToManyField* is all you need. However, sometimes you may need to associate data with the relationship between two models.

For example, consider the case of an application tracking the musical groups which musicians belong to. There is a many-to-many relationship between a person and the groups of which they are a member, so you could use a *ManyToManyField* to represent this relationship. However, there is a lot of detail about the membership that you might want to collect, such as the date at which the person joined the group.

For these situations, Django allows you to specify the model that will be used to govern the many-to-many relationship. You can then put extra fields on the intermediate model. The intermediate model is associated with the *ManyToManyField* using the *through* argument to point to the model that will act as an intermediary. For our musician example, the code would look something like this:

```
from django.db import models


class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name


class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through="Membership")

    def __str__(self):
        return self.name


class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)

    class Meta:
```

(continues on next page)

(continued from previous page)

```
        constraints = [
            models.UniqueConstraint(
                fields=["person", "group"], name="unique_person_group"
            )
        ]
```

When you set up the intermediary model, you explicitly specify foreign keys to the models that are involved in the many-to-many relationship. This explicit declaration defines how the two models are related.

If you don't want multiple associations between the same instances, add a *UniqueConstraint* including the `from` and `to` fields. Django's automatically generated many-to-many tables include such a constraint.

There are a few restrictions on the intermediate model:

- Your intermediate model must contain one - and only one - foreign key to the source model (this would be `Group` in our example), or you must explicitly specify the foreign keys Django should use for the relationship using *ManyToManyField.through_fields*. If you have more than one foreign key and `through_fields` is not specified, a validation error will be raised. A similar restriction applies to the foreign key to the target model (this would be `Person` in our example).
- For a model which has a many-to-many relationship to itself through an intermediary model, two foreign keys to the same model are permitted, but they will be treated as the two (different) sides of the many-to-many relationship. If *through_fields* is not specified, the first foreign key will be taken to represent the source side of the `ManyToManyField`, while the second will be taken to represent the target side. If there are more than two foreign keys though, you must specify *through_fields* to explicitly indicate which foreign keys to use, otherwise a validation error will be raised.

Now that you have set up your *ManyToManyField* to use your intermediary model (`Membership`, in this case), you're ready to start creating some many-to-many relationships. You do this by creating instances of the intermediate model:

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(
...     person=ringo,
...     group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.",
... )
>>> m1.save()
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>]>
```

(continues on next page)

(continued from previous page)

```
>>> ringo.group_set.all()
<QuerySet [<Group: The Beatles>]>
>>> m2 = Membership.objects.create(
...     person=paul,
...     group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.",
... )
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>]>
```

You can also use *add()*, *create()*, or *set()* to create relationships, as long as you specify `through_defaults` for any required fields:

```
>>> beatles.members.add(john, through_defaults={"date_joined": date(1960, 8, 1)})
>>> beatles.members.create(
...     name="George Harrison", through_defaults={"date_joined": date(1960, 8, 1)}
... )
>>> beatles.members.set(
...     [john, paul, ringo, george], through_defaults={"date_joined": date(1960, 8, 1)}
... )
```

You may prefer to create instances of the intermediate model directly.

If the custom through table defined by the intermediate model does not enforce uniqueness on the `(model1, model2)` pair, allowing multiple values, the *remove()* call will remove all intermediate model instances:

```
>>> Membership.objects.create(
...     person=ringo,
...     group=beatles,
...     date_joined=date(1968, 9, 4),
...     invite_reason="You've been gone for a month and we miss you.",
... )
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>, <Person: Ringo Starr>]>
>>> # This deletes both of the intermediate model instances for Ringo Starr
>>> beatles.members.remove(ringo)
>>> beatles.members.all()
<QuerySet [<Person: Paul McCartney>]>
```

The *clear()* method can be used to remove all many-to-many relationships for an instance:

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
<QuerySet []>
```

Once you have established the many-to-many relationships, you can issue queries. Just as with normal many-to-many relationships, you can query using the attributes of the many-to-many-related model:

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith="Paul")
<QuerySet [<Group: The Beatles>]>
```

As you are using an intermediate model, you can also query on its attributes:

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name="The Beatles", membership__date_joined__gt=date(1961, 1, 1)
... )
<QuerySet [<Person: Ringo Starr]>
```

If you need to access a membership's information you may do so by directly querying the `Membership` model:

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

Another way to access the same information is by querying the many-to-many reverse relationship from a `Person` object:

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

One-to-one relationships

To define a one-to-one relationship, use *OneToOneField*. You use it just like any other `Field` type: by including it as a class attribute of your model.

This is most useful on the primary key of an object when that object "extends" another object in some way.

OneToOneField requires a positional argument: the class to which the model is related.

For example, if you were building a database of "places", you would build pretty standard stuff such as address, phone number, etc. in the database. Then, if you wanted to build a database of restaurants on top of the places, instead of repeating yourself and replicating those fields in the `Restaurant` model, you could make `Restaurant` have a *OneToOneField* to `Place` (because a restaurant "is a" place; in fact, to handle this you'd typically use inheritance, which involves an implicit one-to-one relation).

As with *ForeignKey*, a recursive relationship can be defined and references to as-yet undefined models can be made.

> See also
>
> See the One-to-one relationship model example for a full example.

OneToOneField fields also accept an optional *parent_link* argument.

OneToOneField classes used to automatically become the primary key on a model. This is no longer true (although you can manually pass in the *primary_key* argument if you like). Thus, it's now possible to have multiple fields of type *OneToOneField* on a single model.

Models across files

It's perfectly OK to relate a model to one from another app. To do this, import the related model at the top of the file where your model is defined. Then, refer to the other model class wherever needed. For example:

```
from django.db import models
from geography.models import ZipCode


class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(
        ZipCode,
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )
```

Alternatively, you can use a lazy reference to the related model, specified as a string in the format `"app_label.ModelName"`. This does not require the related model to be imported. For example:

```
from django.db import models


class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(
        "geography.ZipCode",
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )
```

See lazy relationships for more details.

Field name restrictions

Django places some restrictions on model field names:

1. A field name cannot be a Python reserved word, because that would result in a Python syntax error. For example:

   ```
   class Example(models.Model):
       pass = models.IntegerField() # 'pass' is a reserved word!
   ```

2. A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works. For example:

   ```
   class Example(models.Model):
       foo__bar = models.IntegerField()  # 'foo__bar' has two underscores!
   ```

3. A field name cannot end with an underscore, for similar reasons.

4. A field name cannot be `check`, as this would override the check framework's `Model.check()` method.

These limitations can be worked around, though, because your field name doesn't necessarily have to match your database column name. See the *db_column* option.

SQL reserved words, such as `join`, `where` or `select`, are allowed as model field names, because Django escapes all database table names and column names in every underlying SQL query. It uses the quoting syntax of your particular database engine.

Custom field types

If one of the existing model fields cannot be used to fit your purposes, or if you wish to take advantage of some less common database column types, you can create your own field class. Full coverage of creating your own fields is provided in How to create custom model fields.

`Meta` options

Give your model metadata by using an inner `class Meta`, like so:

```
from django.db import models


class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

Model metadata is "anything that's not a field", such as ordering options (*`ordering`*), database table name (*`db_table`*), or human-readable singular and plural names (*`verbose_name`* and *`verbose_name_plural`*). None are required, and adding `class Meta` to a model is completely optional.

A complete list of all possible `Meta` options can be found in the model option reference.

Model attributes

`objects`

The most important attribute of a model is the *`Manager`*. It's the interface through which database query operations are provided to Django models and is used to retrieve the instances from the database. If no custom `Manager` is defined, the default name is *`objects`*. Managers are only accessible via model classes, not the model instances.

Model methods

Define custom methods on a model to add custom "row-level" functionality to your objects. Whereas *`Manager`* methods are intended to do "table-wide" things, model methods should act on a particular model instance.

This is a valuable technique for keeping business logic in one place – the model.

For example, this model has a few custom methods:

```
from django.db import models
```

(continues on next page)

(continued from previous page)

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime

        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    @property
    def full_name(self):
        "Returns the person's full name."
        return f"{self.first_name} {self.last_name}"
```

The last method in this example is a property.

The model instance reference has a complete list of methods automatically given to each model. You can override most of these – see overriding predefined model methods, below – but there are a couple that you'll almost always want to define:

__str__()

A Python "magic method" that returns a string representation of any object. This is what Python and Django will use whenever a model instance needs to be coerced and displayed as a plain string. Most notably, this happens when you display an object in an interactive console or in the admin.

You'll always want to define this method; the default isn't very helpful at all.

get_absolute_url()

This tells Django how to calculate the URL for an object. Django uses this in its admin interface, and any time it needs to figure out a URL for an object.

Any object that has a URL that uniquely identifies it should define this method.

Overriding predefined model methods

There's another set of model methods that encapsulate a bunch of database behavior that you'll want to customize. In particular you'll often want to change the way *save()* and *delete()* work.

You're free to override these methods (and any other model method) to alter behavior.

A classic use-case for overriding the built-in methods is if you want something to happen whenever you save an object. For example (see *save()* for documentation of the parameters it accepts):

```
from django.db import models


class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, **kwargs):
        do_something()
        super().save(**kwargs)  # Call the "real" save() method.
        do_something_else()
```

You can also prevent saving:

```
from django.db import models


class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, **kwargs):
        if self.name == "Yoko Ono's blog":
            return  # Yoko shall never have her own blog!
        else:
            super().save(**kwargs)  # Call the "real" save() method.
```

It's important to remember to call the superclass method – that's that `super().save(**kwargs)` business – to ensure that the object still gets saved into the database. If you forget to call the superclass method, the default behavior won't happen and the database won't get touched.

It's also important that you pass through the arguments that can be passed to the model method – that's what the `**kwargs` bit does. Django will, from time to time, extend the capabilities of built-in model methods, adding new keyword arguments. If you use `**kwargs` in your method definitions, you are guaranteed that your code will automatically support those arguments when they are added.

If you wish to update a field value in the *save()* method, you may also want to have this field added to the `update_fields` keyword argument. This will ensure the field is saved when `update_fields` is specified. For example:

```
from django.db import models
from django.utils.text import slugify


class Blog(models.Model):
    name = models.CharField(max_length=100)
    slug = models.TextField()

    def save(self, **kwargs):
        self.slug = slugify(self.name)
        if (
            update_fields := kwargs.get("update_fields")
        ) is not None and "name" in update_fields:
            kwargs["update_fields"] = {"slug"}.union(update_fields)
        super().save(**kwargs)
```

See Specifying which fields to save for more details.

> **Overridden model methods are not called on bulk operations**
>
> Note that the *delete()* method for an object is not necessarily called when deleting objects in bulk using a QuerySet or as a result of a *cascading delete*. To ensure customized delete logic gets executed, you can use *pre_delete* and/or *post_delete* signals.
>
> Unfortunately, there isn't a workaround when *creating* or *updating* objects in bulk, since none of *save()*, *pre_save*, and *post_save* are called.

Executing custom SQL

Another common pattern is writing custom SQL statements in model methods and module-level methods. For more details on using raw SQL, see the documentation on using raw SQL.

Model inheritance

Model inheritance in Django works almost identically to the way normal class inheritance works in Python, but the basics at the beginning of the page should still be followed. That means the base class should subclass *django.db.models.Model*.

The only decision you have to make is whether you want the parent models to be models in their own right (with their own database tables), or if the parents are just holders of common information that will only be

visible through the child models.

There are three styles of inheritance that are possible in Django.

1. Often, you will just want to use the parent class to hold information that you don't want to have to type out for each child model. This class isn't going to ever be used in isolation, so Abstract base classes are what you're after.
2. If you're subclassing an existing model (perhaps something from another application entirely) and want each model to have its own database table, Multi-table inheritance is the way to go.
3. Finally, if you only want to modify the Python-level behavior of a model, without changing the models fields in any way, you can use Proxy models.

Abstract base classes

Abstract base classes are useful when you want to put some common information into a number of other models. You write your base class and put `abstract=True` in the Meta class. This model will then not be used to create any database table. Instead, when it is used as a base class for other models, its fields will be added to those of the child class.

An example:

```
from django.db import models


class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True


class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

The `Student` model will have three fields: `name`, `age` and `home_group`. The `CommonInfo` model cannot be used as a normal Django model, since it is an abstract base class. It does not generate a database table or have a manager, and cannot be instantiated or saved directly.

Fields inherited from abstract base classes can be overridden with another field or value, or be removed with `None`.

For many uses, this type of model inheritance will be exactly what you want. It provides a way to factor out common information at the Python level, while still only creating one database table per child model at the database level.

`Meta` inheritance

When an abstract base class is created, Django makes any Meta inner class you declared in the base class available as an attribute. If a child class does not declare its own Meta class, it will inherit the parent's Meta. If the child wants to extend the parent's Meta class, it can subclass it. For example:

```
from django.db import models


class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ["name"]


class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = "student_info"
```

Django does make one adjustment to the Meta class of an abstract base class: before installing the Meta attribute, it sets `abstract=False`. This means that children of abstract base classes don't automatically become abstract classes themselves. To make an abstract base class that inherits from another abstract base class, you need to explicitly set `abstract=True` on the child.

Some attributes won't make sense to include in the Meta class of an abstract base class. For example, including `db_table` would mean that all the child classes (the ones that don't specify their own Meta) would use the same database table, which is almost certainly not what you want.

Due to the way Python inheritance works, if a child class inherits from multiple abstract base classes, only the Meta options from the first listed class will be inherited by default. To inherit Meta options from multiple abstract base classes, you must explicitly declare the Meta inheritance. For example:

```
from django.db import models


class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True
```

(continues on next page)

(continued from previous page)

```
        ordering = ["name"]


class Unmanaged(models.Model):
    class Meta:
        abstract = True
        managed = False


class Student(CommonInfo, Unmanaged):
    home_group = models.CharField(max_length=5)

    class Meta(CommonInfo.Meta, Unmanaged.Meta):
        pass
```

Be careful with `related_name` and `related_query_name`

If you are using *`related_name`* or *`related_query_name`* on a `ForeignKey` or `ManyToManyField`, you must always specify a unique reverse name and query name for the field. This would normally cause a problem in abstract base classes, since the fields on this class are included into each of the child classes, with exactly the same values for the attributes (including *`related_name`* and *`related_query_name`*) each time.

To work around this problem, when you are using *`related_name`* or *`related_query_name`* in an abstract base class (only), part of the value should contain `'%(app_label)s'` and `'%(class)s'`.

- `'%(class)s'` is replaced by the lowercased name of the child class that the field is used in.
- `'%(app_label)s'` is replaced by the lowercased name of the app the child class is contained within. Each installed application name must be unique and the model class names within each app must also be unique, therefore the resulting name will end up being different.

For example, given an app `common/models.py`:

```
from django.db import models


class Base(models.Model):
    m2m = models.ManyToManyField(
        OtherModel,
        related_name="%(app_label)s_%(class)s_related",
        related_query_name="%(app_label)s_%(class)ss",
    )
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        abstract = True


class ChildA(Base):
    pass


class ChildB(Base):
    pass
```

Along with another app `rare/models.py`:

```
from common.models import Base


class ChildB(Base):
    pass
```

The reverse name of the `common.ChildA.m2m` field will be `common_childa_related` and the reverse query name will be `common_childas`. The reverse name of the `common.ChildB.m2m` field will be `common_childb_related` and the reverse query name will be `common_childbs`. Finally, the reverse name of the `rare.ChildB.m2m` field will be `rare_childb_related` and the reverse query name will be `rare_childbs`. It's up to you how you use the `'%(class)s'` and `'%(app_label)s'` portion to construct your related name or related query name but if you forget to use it, Django will raise errors when you perform system checks (or run *migrate*).

If you don't specify a *related_name* attribute for a field in an abstract base class, the default reverse name will be the name of the child class followed by `'_set'`, just as it normally would be if you'd declared the field directly on the child class. For example, in the above code, if the *related_name* attribute was omitted, the reverse name for the `m2m` field would be `childa_set` in the `ChildA` case and `childb_set` for the `ChildB` field.

Multi-table inheritance

The second type of model inheritance supported by Django is when each model in the hierarchy is a model all by itself. Each model corresponds to its own database table and can be queried and created individually. The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created *OneToOneField*). For example:

```
from django.db import models
```

(continues on next page)

(continued from previous page)

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)


class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

All of the fields of `Place` will also be available in `Restaurant`, although the data will reside in a different database table. So these are both possible:

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

If you have a `Place` that is also a `Restaurant`, you can get from the `Place` object to the `Restaurant` object by using the lowercase version of the model name:

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

However, if `p` in the above example was not a `Restaurant` (it had been created directly as a `Place` object or was the parent of some other class), referring to `p.restaurant` would raise a `Restaurant.DoesNotExist` exception.

The automatically-created *`OneToOneField`* on `Restaurant` that links it to `Place` looks like this:

```
place_ptr = models.OneToOneField(
    Place,
    on_delete=models.CASCADE,
    parent_link=True,
    primary_key=True,
)
```

You can override that field by declaring your own *`OneToOneField`* with *`parent_link=True`* on `Restaurant`.

`Meta` and multi-table inheritance

In the multi-table inheritance situation, it doesn't make sense for a child class to inherit from its parent's Meta class. All the Meta options have already been applied to the parent class and applying them again would normally only lead to contradictory behavior (this is in contrast with the abstract base class case, where the base class doesn't exist in its own right).

So a child model does not have access to its parent's Meta class. However, there are a few limited cases where the child inherits behavior from the parent: if the child does not specify an *`ordering`* attribute or a *`get_latest_by`* attribute, it will inherit these from its parent.

If the parent has an ordering and you don't want the child to have any natural ordering, you can explicitly disable it:

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

Inheritance and reverse relations

Because multi-table inheritance uses an implicit *`OneToOneField`* to link the child and the parent, it's possible to move from the parent down to the child, as in the above example. However, this uses up the name that is the default *`related_name`* value for *`ForeignKey`* and *`ManyToManyField`* relations. If you are putting those types of relations on a subclass of the parent model, you must specify the *`related_name`* attribute on each such field. If you forget, Django will raise a validation error.

For example, using the above `Place` class again, let's create another subclass with a *`ManyToManyField`*:

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

This results in the error:

```
Reverse query name for 'Supplier.customers' clashes with reverse query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition for
'Supplier.customers' or 'Supplier.place_ptr'.
```

Adding `related_name` to the `customers` field as follows would resolve the error: `models.ManyToManyField(Place, related_name='provider')`.

Specifying the parent link field

As mentioned, Django will automatically create a *OneToOneField* linking your child class back to any non-abstract parent models. If you want to control the name of the attribute linking back to the parent, you can create your own *OneToOneField* and set *parent_link=True* to indicate that your field is the link back to the parent class.

Proxy models

When using multi-table inheritance, a new database table is created for each subclass of a model. This is usually the desired behavior, since the subclass needs a place to store any additional data fields that are not present on the base class. Sometimes, however, you only want to change the Python behavior of a model – perhaps to change the default manager, or add a new method.

This is what proxy model inheritance is for: creating a proxy for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model. The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original.

Proxy models are declared like normal models. You tell Django that it's a proxy model by setting the *proxy* attribute of the `Meta` class to `True`.

For example, suppose you want to add a method to the `Person` model. You can do it like this:

```
from django.db import models


class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)


class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

The `MyPerson` class operates on the same database table as its parent `Person` class. In particular, any new instances of `Person` will also be accessible through `MyPerson`, and vice-versa:

```
>>> p = Person.objects.create(first_name="foobar")
```

(continues on next page)

(continued from previous page)

```
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

You could also use a proxy model to define a different default ordering on a model. You might not always want to order the `Person` model, but regularly order by the `last_name` attribute when you use the proxy:

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

Now normal `Person` queries will be unordered and `OrderedPerson` queries will be ordered by `last_name`.

Proxy models inherit `Meta` attributes in the same way as regular models.

`QuerySets` still return the model that was requested

There is no way to have Django return, say, a `MyPerson` object whenever you query for `Person` objects. A queryset for `Person` objects will return those types of objects. The whole point of proxy objects is that code relying on the original `Person` will use those and your own code can use the extensions you included (that no other code is relying on anyway). It is not a way to replace the `Person` (or any other) model everywhere with something of your own creation.

Base class restrictions

A proxy model must inherit from exactly one non-abstract model class. You can't inherit from multiple non-abstract models as the proxy model doesn't provide any connection between the rows in the different database tables. A proxy model can inherit from any number of abstract model classes, providing they do not define any model fields. A proxy model may also inherit from any number of proxy models that share a common non-abstract parent class.

Proxy model managers

If you don't specify any model managers on a proxy model, it inherits the managers from its model parents. If you define a manager on the proxy model, it will become the default, although any managers defined on the parent classes will still be available.

Continuing our example from above, you could change the default manager used when you query the `Person` model like this:

```
from django.db import models
```

(continues on next page)

(continued from previous page)

```
class NewManager(models.Manager):
    # ...
    pass


class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

If you wanted to add a new manager to the Proxy, without replacing the existing default, you can use the techniques described in the custom manager documentation: create a base class containing the new managers and inherit that after the primary base class:

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True


class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

You probably won't need to do this very often, but, when you do, it's possible.

Differences between proxy inheritance and unmanaged models

Proxy model inheritance might look fairly similar to creating an unmanaged model, using the *managed* attribute on a model's `Meta` class.

With careful setting of *Meta.db_table* you could create an unmanaged model that shadows an existing model and adds Python methods to it. However, that would be very repetitive and fragile as you need to keep both copies synchronized if you make any changes.

On the other hand, proxy models are intended to behave exactly like the model they are proxying for. They are always in sync with the parent model since they directly inherit its fields and managers.

The general rules are:

1. If you are mirroring an existing model or database table and don't want all the original database table

columns, use `Meta.managed=False`. That option is normally useful for modeling database views and tables not under the control of Django.

2. If you are wanting to change the Python-only behavior of a model, but keep all the same fields as in the original, use `Meta.proxy=True`. This sets things up so that the proxy model is an exact copy of the storage structure of the original model when data is saved.

Multiple inheritance

Just as with Python's subclassing, it's possible for a Django model to inherit from multiple parent models. Keep in mind that normal Python name resolution rules apply. The first base class that a particular name (e.g. Meta) appears in will be the one that is used; for example, this means that if multiple parents contain a Meta class, only the first one is going to be used, and all others will be ignored.

Generally, you won't need to inherit from multiple parents. The main use-case where this is useful is for "mix-in" classes: adding a particular extra field or method to every class that inherits the mix-in. Try to keep your inheritance hierarchies as simple and straightforward as possible so that you won't have to struggle to work out where a particular piece of information is coming from.

Note that inheriting from multiple models that have a common `id` primary key field will raise an error. To properly use multiple inheritance, you can use an explicit *AutoField* in the base models:

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...


class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...


class BookReview(Book, Article):
    pass
```

Or use a common ancestor to hold the *AutoField*. This requires using an explicit *OneToOneField* from each parent model to the common ancestor to avoid a clash between the fields that are automatically generated and inherited by the child:

```
class Piece(models.Model):
    pass


class Article(Piece):
```

(continues on next page)

(continued from previous page)

```
    article_piece = models.OneToOneField(
        Piece, on_delete=models.CASCADE, parent_link=True
    )
    ...


class Book(Piece):
    book_piece = models.OneToOneField(Piece, on_delete=models.CASCADE, parent_link=True)
    ...


class BookReview(Book, Article):
    pass
```

Field name "hiding" is not permitted

In normal Python class inheritance, it is permissible for a child class to override any attribute from the parent class. In Django, this isn't usually permitted for model fields. If a non-abstract model base class has a field called `author`, you can't create another model field or define an attribute called `author` in any class that inherits from that base class.

This restriction doesn't apply to model fields inherited from an abstract model. Such fields may be overridden with another field or value, or be removed by setting `field_name = None`.

> **Warning**
>
> Model managers are inherited from abstract base classes. Overriding an inherited field which is referenced by an inherited *`Manager`* may cause subtle bugs. See custom managers and model inheritance.

> **Note**
>
> Some fields define extra attributes on the model, e.g. a *`ForeignKey`* defines an extra attribute with `_id` appended to the field name, as well as `related_name` and `related_query_name` on the foreign model.
>
> These extra attributes cannot be overridden unless the field that defines it is changed or removed so that it no longer defines the extra attribute.

Overriding fields in a parent model leads to difficulties in areas such as initializing new instances (specifying which field is being initialized in `Model.__init__`) and serialization. These are features which normal Python class inheritance doesn't have to deal with in quite the same way, so the difference between Django model inheritance and Python class inheritance isn't arbitrary.

This restriction only applies to attributes which are *Field* instances. Normal Python attributes can be overridden if you wish. It also only applies to the name of the attribute as Python sees it: if you are manually specifying the database column name, you can have the same column name appearing in both a child and an ancestor model for multi-table inheritance (they are columns in two different database tables).

Django will raise a *FieldError* if you override any model field in any ancestor model.

Note that because of the way fields are resolved during class definition, model fields inherited from multiple abstract parent models are resolved in a strict depth-first order. This contrasts with standard Python MRO, which is resolved breadth-first in cases of diamond shaped inheritance. This difference only affects complex model hierarchies, which (as per the advice above) you should try to avoid.

Organizing models in a package

The *manage.py startapp* command creates an application structure that includes a `models.py` file. If you have many models, organizing them in separate files may be useful.

To do so, create a `models` package. Remove `models.py` and create a `myapp/models/` directory with an `__init__.py` file and the files to store your models. You must import the models in the `__init__.py` file.

For example, if you had `organic.py` and `synthetic.py` in the `models` directory:

Listing 1: `myapp/models/__init__.py`

```
from .organic import Person
from .synthetic import Robot
```

Explicitly importing each model rather than using `from .models import *` has the advantages of not cluttering the namespace, making code more readable, and keeping code analysis tools useful.

See also

The Models Reference
: Covers all the model related APIs including model fields, related objects, and `QuerySet`.

3.2.2 Making queries

Once you've created your data models, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. This document explains how to use this API. Refer to the data model reference for full details of all the various model lookup options.

Throughout this guide (and in the reference), we'll refer to the following models, which comprise a blog application:

```
from datetime import date
```

(continues on next page)

(continued from previous page)

```
from django.db import models


class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name


class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):
        return self.name


class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField(default=date.today)
    authors = models.ManyToManyField(Author)
    number_of_comments = models.IntegerField(default=0)
    number_of_pingbacks = models.IntegerField(default=0)
    rating = models.IntegerField(default=5)

    def __str__(self):
        return self.headline
```

Creating objects

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call *save()* to save it to the database.

Assuming models live in a `models.py` file inside a `blog` Django app, here is an example:

```
>>> from blog.models import Blog
>>> b = Blog(name="Beatles Blog", tagline="All the latest Beatles news.")
>>> b.save()
```

This performs an `INSERT` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call *save()*.

The *save()* method has no return value.

See also

save() takes a number of advanced options not described here. See the documentation for *save()* for complete details.

To create and save an object in a single step, use the *create()* method.

Saving changes to objects

To save changes to an object that's already in the database, use *save()*.

Given a `Blog` instance `b5` that has already been saved to the database, this example changes its name and updates its record in the database:

```
>>> b5.name = "New name"
>>> b5.save()
```

This performs an `UPDATE` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call *save()*.

Saving `ForeignKey` and `ManyToManyField` fields

Updating a *ForeignKey* field works exactly the same way as saving a normal field – assign an object of the right type to the field in question. This example updates the `blog` attribute of an `Entry` instance `entry`, assuming appropriate instances of `Entry` and `Blog` are already saved to the database (so we can retrieve them below):

```
>>> from blog.models import Blog, Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

Updating a *ManyToManyField* works a little differently – use the *add()* method on the field to add a record to the relation. This example adds the `Author` instance `joe` to the `entry` object:

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

To add multiple records to a *ManyToManyField* in one go, include multiple arguments in the call to *add()*, like this:

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django will complain if you try to assign or add an object of the wrong type.

Retrieving objects

To retrieve objects from your database, construct a *QuerySet* via a *Manager* on your model class.

A *QuerySet* represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters. In SQL terms, a *QuerySet* equates to a `SELECT` statement, and a filter is a limiting clause such as `WHERE` or `LIMIT`.

You get a *QuerySet* by using your model's *Manager*. Each model has at least one *Manager*, and it's called *objects* by default. Access it directly via the model class, like so:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name="Foo", tagline="Bar")
>>> b.objects
Traceback:
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

> **Note**
>
> A `Manager` is accessible only via model classes, rather than from model instances, to enforce a separation between "table-level" operations and "record-level" operations.

The *Manager* is the main source of querysets for a model. For example, `Blog.objects.all()` returns a *QuerySet* that contains all `Blog` objects in the database.

Retrieving all objects

The simplest way to retrieve objects from a table is to get all of them. To do this, use the *all()* method on a *Manager*:

```
>>> all_entries = Entry.objects.all()
```

The *all()* method returns a *QuerySet* of all the objects in the database.

Retrieving specific objects with filters

The *QuerySet* returned by *all()* describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial *QuerySet*, adding filter conditions. The two most common ways to refine a *QuerySet* are:

filter(kwargs)**
: Returns a new *QuerySet* containing objects that match the given lookup parameters.

exclude(kwargs)**
: Returns a new *QuerySet* containing objects that do not match the given lookup parameters.

The lookup parameters (`**kwargs` in the above function definitions) should be in the format described in Field lookups below.

For example, to get a *QuerySet* of blog entries from the year 2006, use *filter()* like so:

```
Entry.objects.filter(pub_date__year=2006)
```

With the default manager class, it is the same as:

```
Entry.objects.all().filter(pub_date__year=2006)
```

Chaining filters

The result of refining a *QuerySet* is itself a *QuerySet*, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(headline__startswith="What").exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(pub_date__gte=datetime.date(2005, 1, 30))
```

This takes the initial *QuerySet* of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a *QuerySet* containing all entries with a headline that starts with "What", that were published between January 30, 2005, and the current day.

Filtered QuerySets are unique

Each time you refine a *QuerySet*, you get a brand-new *QuerySet* that is in no way bound to the previous *QuerySet*. Each refinement creates a separate and distinct *QuerySet* that can be stored, used and reused.

Example:

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

These three querysets are separate. The first is a base *QuerySet* containing all entries that contain a headline starting with "What". The second is a subset of the first, with an additional criteria that excludes records whose `pub_date` is today or in the future. The third is a subset of the first, with an additional criteria that selects only the records whose `pub_date` is today or in the future. The initial *QuerySet* (`q1`) is unaffected by the refinement process.

QuerySets are lazy

`QuerySet` objects are lazy – the act of creating a *QuerySet* doesn't involve any database activity. You can stack filters together all day long, and Django won't actually run the query until the *QuerySet* is evaluated. Take a look at this example:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (`print(q)`). In general, the results of a *QuerySet* aren't fetched from the database until you "ask" for them. When you do, the *QuerySet* is evaluated by accessing the database. For more details on exactly when evaluation takes place, see When QuerySets are evaluated.

Retrieving a single object with `get()`

filter() will always give you a *QuerySet*, even if only a single object matches the query - in this case, it will be a *QuerySet* containing a single element.

If you know there is only one object that matches your query, you can use the *get()* method on a *Manager* which returns the object directly:

```
>>> one_entry = Entry.objects.get(pk=1)
```

You can use any query expression with *get()*, just like with *filter()* - again, see Field lookups below.

Note that there is a difference between using *get()*, and using *filter()* with a slice of `[0]`. If there are no results that match the query, *get()* will raise a `DoesNotExist` exception. This exception is an attribute of the model class that the query is being performed on - so in the code above, if there is no `Entry` object with a primary key of 1, Django will raise `Entry.DoesNotExist`.

Similarly, Django will complain if more than one item matches the *get()* query. In this case, it will raise *MultipleObjectsReturned*, which again is an attribute of the model class itself.

Other `QuerySet` methods

Most of the time you'll use *all()*, *get()*, *filter()* and *exclude()* when you need to look up objects from the database. However, that's far from all there is; see the QuerySet API Reference for a complete list of all the various *QuerySet* methods.

Limiting `QuerySets`

Use a subset of Python's array-slicing syntax to limit your *QuerySet* to a certain number of results. This is the equivalent of SQL's `LIMIT` and `OFFSET` clauses.

For example, this returns the first 5 objects (`LIMIT 5`):

```
>>> Entry.objects.all()[:5]
```

This returns the sixth through tenth objects (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all()[5:10]
```

Negative indexing (i.e. `Entry.objects.all()[-1]`) is not supported.

Generally, slicing a *QuerySet* returns a new *QuerySet* – it doesn't evaluate the query. An exception is if you use the "step" parameter of Python slice syntax. For example, this would actually execute the query in order to return a list of every second object of the first 10:

```
>>> Entry.objects.all()[:10:2]
```

Further filtering or ordering of a sliced queryset is prohibited due to the ambiguous nature of how that might work.

To retrieve a single object rather than a list (e.g. `SELECT foo FROM bar LIMIT 1`), use an index instead of a slice. For example, this returns the first `Entry` in the database, after ordering entries alphabetically by headline:

```
>>> Entry.objects.order_by("headline")[0]
```

This is roughly equivalent to:

```
>>> Entry.objects.order_by("headline")[0:1].get()
```

Note, however, that the first of these will raise `IndexError` while the second will raise `DoesNotExist` if no objects match the given criteria. See *`get()`* for more details.

Field lookups

Field lookups are how you specify the meat of an SQL `WHERE` clause. They're specified as keyword arguments to the *`QuerySet`* methods *`filter()`*, *`exclude()`* and *`get()`*.

Basic lookups keyword arguments take the form `field__lookuptype=value`. (That's a double-underscore). For example:

```
>>> Entry.objects.filter(pub_date__lte="2006-01-01")
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

> **How this is possible**
>
> Python has the ability to define functions that accept arbitrary name-value arguments whose names and values are evaluated at runtime. For more information, see Keyword Arguments in the official Python tutorial.

The field specified in a lookup has to be the name of a model field. There's one exception though, in case of a *`ForeignKey`* you can specify the field name suffixed with `_id`. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key. For example:

```
>>> Entry.objects.filter(blog_id=4)
```

If you pass an invalid keyword argument, a lookup function will raise `TypeError`.

The database API supports about two dozen lookup types; a complete reference can be found in the field lookup reference. To give you a taste of what's available, here's some of the more common lookups you'll probably use:

`exact`

An "exact" match. For example:

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
```

Would generate SQL along these lines:

```
SELECT ... WHERE headline = 'Cat bites dog';
```

If you don't provide a lookup type – that is, if your keyword argument doesn't contain a double underscore – the lookup type is assumed to be `exact`.

For example, the following two statements are equivalent:

```
>>> Blog.objects.get(id__exact=14)  # Explicit form
>>> Blog.objects.get(id=14)  # __exact is implied
```

This is for convenience, because `exact` lookups are the common case.

iexact

A case-insensitive match. So, the query:

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

Would match a `Blog` titled `"Beatles Blog"`, `"beatles blog"`, or even `"BeAtlES blOG"`.

contains

Case-sensitive containment test. For example:

```
Entry.objects.get(headline__contains="Lennon")
```

Roughly translates to this SQL:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Note this will match the headline `'Today Lennon honored'` but not `'today lennon honored'`.

There's also a case-insensitive version, *icontains*.

startswith, *endswith*

Starts-with and ends-with search, respectively. There are also case-insensitive versions called *istartswith* and *iendswith*.

Again, this only scratches the surface. A complete reference can be found in the field lookup reference.

Lookups that span relationships

Django offers a powerful and intuitive way to "follow" relationships in lookups, taking care of the SQL `JOIN`s for you automatically, behind the scenes. To span a relationship, use the field name of related fields across models, separated by double underscores, until you get to the field you want.

This example retrieves all `Entry` objects with a `Blog` whose `name` is `'Beatles Blog'`:

```
>>> Entry.objects.filter(blog__name="Beatles Blog")
```

This spanning can be as deep as you'd like.

It works backwards, too. While it *can be customized*, by default you refer to a "reverse" relationship in a lookup using the lowercase name of the model.

This example retrieves all `Blog` objects which have at least one `Entry` whose `headline` contains `'Lennon'`:

```
>>> Blog.objects.filter(entry__headline__contains="Lennon")
```

If you are filtering across multiple relationships and one of the intermediate models doesn't have a value that meets the filter condition, Django will treat it as if there is an empty (all values are `NULL`), but valid, object there. All this means is that no error will be raised. For example, in this filter:

```
Blog.objects.filter(entry__authors__name="Lennon")
```

(if there was a related `Author` model), if there was no `author` associated with an entry, it would be treated as if there was also no `name` attached, rather than raising an error because of the missing `author`. Usually this is exactly what you want to have happen. The only case where it might be confusing is if you are using *isnull*. Thus:

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

will return `Blog` objects that have an empty `name` on the `author` and also those which have an empty `author` on the `entry`. If you don't want those latter objects, you could write:

```
Blog.objects.filter(entry__authors__isnull=False, entry__authors__name__isnull=True)
```

Spanning multi-valued relationships

When spanning a *ManyToManyField* or a reverse *ForeignKey* (such as from `Blog` to `Entry`), filtering on multiple attributes raises the question of whether to require each attribute to coincide in the same related object. We might seek blogs that have an entry from 2008 with "Lennon" in its headline, or we might seek blogs that merely have any entry from 2008 as well as some newer or older entry with "Lennon" in its headline.

To select all blogs containing at least one entry from 2008 having "Lennon" in its headline (the same entry satisfying both conditions), we would write:

```
Blog.objects.filter(entry__headline__contains="Lennon", entry__pub_date__year=2008)
```

Otherwise, to perform a more permissive query selecting any blogs with merely some entry with "Lennon" in its headline and some entry from 2008, we would write:

```
Blog.objects.filter(entry__headline__contains="Lennon").filter(
    entry__pub_date__year=2008
)
```

Suppose there is only one blog that has both entries containing "Lennon" and entries from 2008, but that none of the entries from 2008 contained "Lennon". The first query would not return any blogs, but the second query would return that one blog. (This is because the entries selected by the second filter may or may not be the same as the entries in the first filter. We are filtering the `Blog` items with each filter statement, not the `Entry` items.) In short, if each condition needs to match the same related object, then each should be contained in a single *`filter()`* call.

> **Note**
>
> As the second (more permissive) query chains multiple filters, it performs multiple joins to the primary model, potentially yielding duplicates.

```
>>> from datetime import date
>>> beatles = Blog.objects.create(name="Beatles Blog")
>>> pop = Blog.objects.create(name="Pop Music Blog")
>>> Entry.objects.create(
...     blog=beatles,
...     headline="New Lennon Biography",
...     pub_date=date(2008, 6, 1),
... )
<Entry: New Lennon Biography>
>>> Entry.objects.create(
...     blog=beatles,
...     headline="New Lennon Biography in Paperback",
...     pub_date=date(2009, 6, 1),
... )
<Entry: New Lennon Biography in Paperback>
>>> Entry.objects.create(
...     blog=pop,
...     headline="Best Albums of 2008",
...     pub_date=date(2008, 12, 15),
... )
<Entry: Best Albums of 2008>
>>> Entry.objects.create(
...     blog=pop,
...     headline="Lennon Would Have Loved Hip Hop",
...     pub_date=date(2020, 4, 1),
... )
<Entry: Lennon Would Have Loved Hip Hop>
>>> Blog.objects.filter(
...     entry__headline__contains="Lennon",
...     entry__pub_date__year=2008,
```

```
... )
<QuerySet [<Blog: Beatles Blog>]>
>>> Blog.objects.filter(
...     entry__headline__contains="Lennon",
... ).filter(
...     entry__pub_date__year=2008,
... )
<QuerySet [<Blog: Beatles Blog>, <Blog: Beatles Blog>, <Blog: Pop Music Blog]>
```

Note

The behavior of *filter()* for queries that span multi-value relationships, as described above, is not implemented equivalently for *exclude()*. Instead, the conditions in a single *exclude()* call will not necessarily refer to the same item.

For example, the following query would exclude blogs that contain both entries with "Lennon" in the headline and entries published in 2008:

```
Blog.objects.exclude(
    entry__headline__contains="Lennon",
    entry__pub_date__year=2008,
)
```

However, unlike the behavior when using *filter()*, this will not limit blogs based on entries that satisfy both conditions. In order to do that, i.e. to select all blogs that do not contain entries published with "Lennon" that were published in 2008, you need to make two queries:

```
Blog.objects.exclude(
    entry__in=Entry.objects.filter(
        headline__contains="Lennon",
        pub_date__year=2008,
    ),
)
```

Filters can reference fields on the model

In the examples given so far, we have constructed filters that compare the value of a model field with a constant. But what if you want to compare the value of a model field with another field on the same model?

Django provides *F expressions* to allow such comparisons. Instances of `F()` act as a reference to a model field within a query. These references can then be used in query filters to compare the values of two different fields on the same model instance.

For example, to find a list of all blog entries that have had more comments than pingbacks, we construct an `F()` object to reference the pingback count, and use that `F()` object in the query:

```
>>> from django.db.models import F
>>> Entry.objects.filter(number_of_comments__gt=F("number_of_pingbacks"))
```

Django supports the use of addition, subtraction, multiplication, division, modulo, and power arithmetic with `F()` objects, both with constants and with other `F()` objects. To find all the blog entries with more than twice as many comments as pingbacks, we modify the query:

```
>>> Entry.objects.filter(number_of_comments__gt=F("number_of_pingbacks") * 2)
```

To find all the entries where the rating of the entry is less than the sum of the pingback count and comment count, we would issue the query:

```
>>> Entry.objects.filter(rating__lt=F("number_of_comments") + F("number_of_pingbacks"))
```

You can also use the double underscore notation to span relationships in an `F()` object. An `F()` object with a double underscore will introduce any joins needed to access the related object. For example, to retrieve all the entries where the author's name is the same as the blog name, we could issue the query:

```
>>> Entry.objects.filter(authors__name=F("blog__name"))
```

For date and date/time fields, you can add or subtract a `timedelta` object. The following would return all entries that were modified more than 3 days after they were published:

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F("pub_date") + timedelta(days=3))
```

The `F()` objects support bitwise operations by `.bitand()`, `.bitor()`, `.bitxor()`, `.bitrightshift()`, and `.bitleftshift()`. For example:

```
>>> F("somefield").bitand(16)
```

> **Oracle**
>
> Oracle doesn't support bitwise XOR operation.

Expressions can reference transforms

Django supports using transforms in expressions.

For example, to find all `Entry` objects published in the same year as they were last modified:

```
>>> from django.db.models import F
>>> Entry.objects.filter(pub_date__year=F("mod_date__year"))
```

To find the earliest year an entry was published, we can issue the query:

```
>>> from django.db.models import Min
>>> Entry.objects.aggregate(first_published_year=Min("pub_date__year"))
```

This example finds the value of the highest rated entry and the total number of comments on all entries for each year:

```
>>> from django.db.models import OuterRef, Subquery, Sum
>>> Entry.objects.values("pub_date__year").annotate(
...     top_rating=Subquery(
...         Entry.objects.filter(
...             pub_date__year=OuterRef("pub_date__year"),
...         )
...         .order_by("-rating")
...         .values("rating")[:1]
...     ),
...     total_comments=Sum("number_of_comments"),
... )
```

The `pk` lookup shortcut

For convenience, Django provides a `pk` lookup shortcut, which stands for "primary key".

In the example `Blog` model, the primary key is the `id` field, so these three statements are equivalent:

```
>>> Blog.objects.get(id__exact=14)  # Explicit form
>>> Blog.objects.get(id=14)  # __exact is implied
>>> Blog.objects.get(pk=14)  # pk implies id__exact
```

The use of `pk` isn't limited to `__exact` queries – any query term can be combined with `pk` to perform a query on the primary key of a model:

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1, 4, 7])

# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

`pk` lookups also work across joins. For example, these three statements are equivalent:

```
>>> Entry.objects.filter(blog__id__exact=3)  # Explicit form
>>> Entry.objects.filter(blog__id=3)  # __exact is implied
>>> Entry.objects.filter(blog__pk=3)  # __pk implies __id__exact
```

Escaping percent signs and underscores in LIKE statements

The field lookups that equate to LIKE SQL statements (`iexact`, `contains`, `icontains`, `startswith`, `istartswith`, `endswith` and `iendswith`) will automatically escape the two special characters used in LIKE statements – the percent sign and the underscore. (In a LIKE statement, the percent sign signifies a multiple-character wildcard and the underscore signifies a single-character wildcard.)

This means things should work intuitively, so the abstraction doesn't leak. For example, to retrieve all the entries that contain a percent sign, use the percent sign as any other character:

```
>>> Entry.objects.filter(headline__contains="%")
```

Django takes care of the quoting for you; the resulting SQL will look something like this:

```
SELECT ... WHERE headline LIKE '%\%%';
```

Same goes for underscores. Both percentage signs and underscores are handled for you transparently.

Caching and QuerySets

Each *QuerySet* contains a cache to minimize database access. Understanding how it works will allow you to write the most efficient code.

In a newly created *QuerySet*, the cache is empty. The first time a *QuerySet* is evaluated – and, hence, a database query happens – Django saves the query results in the *QuerySet*'s cache and returns the results that have been explicitly requested (e.g., the next element, if the *QuerySet* is being iterated over). Subsequent evaluations of the *QuerySet* reuse the cached results.

Keep this caching behavior in mind, because it may bite you if you don't use your *QuerySet*s correctly. For example, the following will create two *QuerySet*s, evaluate them, and throw them away:

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

That means the same database query will be executed twice, effectively doubling your database load. Also, there's a possibility the two lists may not include the same database records, because an `Entry` may have been added or deleted in the split second between the two requests.

To avoid this problem, save the *QuerySet* and reuse it:

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset])  # Evaluate the query set.
>>> print([p.pub_date for p in queryset])  # Reuse the cache from the evaluation.
```

When QuerySets are not cached

Querysets do not always cache their results. When evaluating only part of the queryset, the cache is checked, but if it is not populated then the items returned by the subsequent query are not cached. Specifically, this means that limiting the queryset using an array slice or an index will not populate the cache.

For example, repeatedly getting a certain index in a queryset object will query the database each time:

```
>>> queryset = Entry.objects.all()
>>> print(queryset[5])  # Queries the database
>>> print(queryset[5])  # Queries the database again
```

However, if the entire queryset has already been evaluated, the cache will be checked instead:

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset]  # Queries the database
>>> print(queryset[5])  # Uses cache
>>> print(queryset[5])  # Uses cache
```

Here are some examples of other actions that will result in the entire queryset being evaluated and therefore populate the cache:

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

> **Note**
>
> Simply printing the queryset will not populate the cache. This is because the call to `__repr__()` only returns a slice of the entire queryset.

Asynchronous queries

If you are writing asynchronous views or code, you cannot use the ORM for queries in quite the way we have described above, as you cannot call blocking synchronous code from asynchronous code - it will block up the event loop (or, more likely, Django will notice and raise a `SynchronousOnlyOperation` to stop that from happening).

Fortunately, you can do many queries using Django's asynchronous query APIs. Every method that might block - such as `get()` or `delete()` - has an asynchronous variant (`aget()` or `adelete()`), and when you iterate over results, you can use asynchronous iteration (`async for`) instead.

Query iteration

The default way of iterating over a query - with `for` - will result in a blocking database query behind the scenes as Django loads the results at iteration time. To fix this, you can swap to `async for`:

```
async for entry in Authors.objects.filter(name__startswith="A"):
    ...
```

Be aware that you also can't do other things that might iterate over the queryset, such as wrapping `list()` around it to force its evaluation (you can use `async for` in a comprehension, if you want it).

Because `QuerySet` methods like `filter()` and `exclude()` do not actually run the query - they set up the queryset to run when it's iterated over - you can use those freely in asynchronous code. For a guide to which methods can keep being used like this, and which have asynchronous versions, read the next section.

`QuerySet` and manager methods

Some methods on managers and querysets - like `get()` and `first()` - force execution of the queryset and are blocking. Some, like `filter()` and `exclude()`, don't force execution and so are safe to run from asynchronous code. But how are you supposed to tell the difference?

While you could poke around and see if there is an `a`-prefixed version of the method (for example, we have `aget()` but not `afilter()`), there is a more logical way - look up what kind of method it is in the QuerySet reference.

In there, you'll find the methods on QuerySets grouped into two sections:

- Methods that return new querysets: These are the non-blocking ones, and don't have asynchronous versions. You're free to use these in any situation, though read the notes on `defer()` and `only()` before you use them.
- Methods that do not return querysets: These are the blocking ones, and have asynchronous versions - the asynchronous name for each is noted in its documentation, though our standard pattern is to add an `a` prefix.

Using this distinction, you can work out when you need to use asynchronous versions, and when you don't. For example, here's a valid asynchronous query:

```
user = await User.objects.filter(username=my_input).afirst()
```

`filter()` returns a queryset, and so it's fine to keep chaining it inside an asynchronous environment, whereas `first()` evaluates and returns a model instance - thus, we change to `afirst()`, and use `await` at the front of the whole expression in order to call it in an asynchronous-friendly way.

> **Note**
>
> If you forget to put the `await` part in, you may see errors like "coroutine object has no attribute x" or "<coroutine . . .>" strings in place of your model instances. If you ever see these, you are missing an `await` somewhere to turn that coroutine into a real value.

Transactions

Transactions are not currently supported with asynchronous queries and updates. You will find that trying to use one raises `SynchronousOnlyOperation`.

If you wish to use a transaction, we suggest you write your ORM code inside a separate, synchronous function and then call that using `sync_to_async` - see Asynchronous support for more.

Querying `JSONField`

Lookups implementation is different in *`JSONField`*, mainly due to the existence of key transformations. To demonstrate, we will use the following example model:

```
from django.db import models


class Dog(models.Model):
    name = models.CharField(max_length=200)
    data = models.JSONField(null=True)

    def __str__(self):
        return self.name
```

Storing and querying for `None`

As with other fields, storing `None` as the field's value will store it as SQL `NULL`. While not recommended, it is possible to store JSON scalar `null` instead of SQL `NULL` by using *`Value(None, JSONField())`*.

Whichever of the values is stored, when retrieved from the database, the Python representation of the JSON scalar `null` is the same as SQL `NULL`, i.e. `None`. Therefore, it can be hard to distinguish between them.

This only applies to `None` as the top-level value of the field. If `None` is inside a `list` or `dict`, it will always be interpreted as JSON `null`.

When querying, `None` value will always be interpreted as JSON `null`. To query for SQL `NULL`, use *`isnull`*:

```
>>> Dog.objects.create(name="Max", data=None)  # SQL NULL.
<Dog: Max>
```

(continues on next page)

(continued from previous page)

```
>>> Dog.objects.create(name="Archie", data=Value(None, JSONField()))  # JSON null.
<Dog: Archie>
>>> Dog.objects.filter(data=None)
<QuerySet [<Dog: Archie>]>
>>> Dog.objects.filter(data=Value(None, JSONField()))
<QuerySet [<Dog: Archie>]>
>>> Dog.objects.filter(data__isnull=True)
<QuerySet [<Dog: Max>]>
>>> Dog.objects.filter(data__isnull=False)
<QuerySet [<Dog: Archie>]>
```

Unless you are sure you wish to work with SQL `NULL` values, consider setting `null=False` and providing a suitable default for empty values, such as `default=dict`.

> **Note**
>
> Storing JSON scalar `null` does not violate *`null=False`*.

Key, index, and path transforms

To query based on a given dictionary key, use that key as the lookup name:

```
>>> Dog.objects.create(
...     name="Rufus",
...     data={
...         "breed": "labrador",
...         "owner": {
...             "name": "Bob",
...             "other_pets": [
...                 {
...                     "name": "Fishy",
...                 }
...             ],
...         },
...     },
... )
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": None})
<Dog: Meg>
>>> Dog.objects.filter(data__breed="collie")
```

(continues on next page)

(continued from previous page)

```
<QuerySet [<Dog: Meg>]>
```

Multiple keys can be chained together to form a path lookup:

```
>>> Dog.objects.filter(data__owner__name="Bob")
<QuerySet [<Dog: Rufus>]>
```

If the key is an integer, it will be interpreted as an index transform in an array:

```
>>> Dog.objects.filter(data__owner__other_pets__0__name="Fishy")
<QuerySet [<Dog: Rufus>]>
```

If the key you wish to query by clashes with the name of another lookup, use the *contains* lookup instead.

To query for missing keys, use the `isnull` lookup:

```
>>> Dog.objects.create(name="Shep", data={"breed": "collie"})
<Dog: Shep>
>>> Dog.objects.filter(data__owner__isnull=True)
<QuerySet [<Dog: Shep>]>
```

> **Note**
>
> The lookup examples given above implicitly use the *exact* lookup. Key, index, and path transforms can also be chained with: *icontains*, *endswith*, *iendswith*, *iexact*, *regex*, *iregex*, *startswith*, *istartswith*, *lt*, *lte*, *gt*, and *gte*, as well as with Containment and key lookups.

`KT()` expressions

class `KT`(lookup)

Represents the text value of a key, index, or path transform of *JSONField*. You can use the double underscore notation in `lookup` to chain dictionary key and index transforms.

For example:

```
>>> from django.db.models.fields.json import KT
>>> Dog.objects.create(
...     name="Shep",
...     data={
...         "owner": {"name": "Bob"},
...         "breed": ["collie", "lhasa apso"],
...     },
```

(continues on next page)

(continued from previous page)

```
... )
<Dog: Shep>
>>> Dog.objects.annotate(
...     first_breed=KT("data__breed__1"), owner_name=KT("data__owner__name")
... ).filter(first_breed__startswith="lhasa", owner_name="Bob")
<QuerySet [<Dog: Shep>]>
```

Note

Due to the way in which key-path queries work, *exclude()* and *filter()* are not guaranteed to produce exhaustive sets. If you want to include objects that do not have the path, add the `isnull` lookup.

Warning

Since any string could be a key in a JSON object, any lookup other than those listed below will be interpreted as a key lookup. No errors are raised. Be extra careful for typing mistakes, and always check your queries work as you intend.

MariaDB and Oracle users

Using *order_by()* on key, index, or path transforms will sort the objects using the string representation of the values. This is because MariaDB and Oracle Database do not provide a function that converts JSON values into their equivalent SQL values.

Oracle users

On Oracle Database, using `None` as the lookup value in an *exclude()* query will return objects that do not have `null` as the value at the given path, including objects that do not have the path. On other database backends, the query will return objects that have the path and the value is not `null`.

PostgreSQL users

On PostgreSQL, if only one key or index is used, the SQL operator `->` is used. If multiple operators are used then the `#>` operator is used.

> **SQLite users**
>
> On SQLite, `"true"`, `"false"`, and `"null"` string values will always be interpreted as `True`, `False`, and JSON `null` respectively.

Containment and key lookups

`contains`

The *`contains`* lookup is overridden on `JSONField`. The returned objects are those where the given `dict` of key-value pairs are all contained in the top-level of the field. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador", "owner": "Bob"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.create(name="Fred", data={})
<Dog: Fred>
>>> Dog.objects.create(
...     name="Merry", data={"breed": "pekingese", "tricks": ["fetch", "dance"]}
... )
>>> Dog.objects.filter(data__contains={"owner": "Bob"})
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
>>> Dog.objects.filter(data__contains={"breed": "collie"})
<QuerySet [<Dog: Meg>]>
>>> Dog.objects.filter(data__contains={"tricks": ["dance"]})
<QuerySet [<Dog: Merry>]>
```

> **Oracle and SQLite**
>
> `contains` is not supported on Oracle and SQLite.

`contained_by`

This is the inverse of the *`contains`* lookup - the objects returned will be those where the key-value pairs on the object are a subset of those in the value passed. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador", "owner": "Bob"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
```

(continues on next page)

(continued from previous page)

```
>>> Dog.objects.create(name="Fred", data={})
<Dog: Fred>
>>> Dog.objects.create(
...     name="Merry", data={"breed": "pekingese", "tricks": ["fetch", "dance"]}
... )
>>> Dog.objects.filter(data__contained_by={"breed": "collie", "owner": "Bob"})
<QuerySet [<Dog: Meg>, <Dog: Fred>]>
>>> Dog.objects.filter(data__contained_by={"breed": "collie"})
<QuerySet [<Dog: Fred>]>
>>> Dog.objects.filter(
...     data__contained_by={"breed": "pekingese", "tricks": ["dance", "fetch", "hug"]}
... )
<QuerySet [<Dog: Merry>, <Dog: Fred>]>
```

Oracle and SQLite

`contained_by` is not supported on Oracle and SQLite.

has_key

Returns objects where the given key is in the top-level of the data. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.filter(data__has_key="owner")
<QuerySet [<Dog: Meg>]>
```

has_keys

Returns objects where all of the given keys are in the top-level of the data. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"breed": "collie", "owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.filter(data__has_keys=["breed", "owner"])
<QuerySet [<Dog: Meg>]>
```

has_any_keys

Returns objects where any of the given keys are in the top-level of the data. For example:

```
>>> Dog.objects.create(name="Rufus", data={"breed": "labrador"})
<Dog: Rufus>
>>> Dog.objects.create(name="Meg", data={"owner": "Bob"})
<Dog: Meg>
>>> Dog.objects.filter(data__has_any_keys=["owner", "breed"])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

Complex lookups with `Q` objects

Keyword argument queries – in *`filter()`*, etc. – are "AND"ed together. If you need to execute more complex queries (for example, queries with `OR` statements), you can use *`Q objects`*.

A *`Q object`*(`django.db.models.Q`) is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in "Field lookups" above.

For example, this `Q` object encapsulates a single `LIKE` query:

```
from django.db.models import Q

Q(question__startswith="What")
```

`Q` objects can be combined using the `&`, `|`, and `^` operators. When an operator is used on two `Q` objects, it yields a new `Q` object.

For example, this statement yields a single `Q` object that represents the "OR" of two `"question__startswith"` queries:

```
Q(question__startswith="Who") | Q(question__startswith="What")
```

This is equivalent to the following SQL `WHERE` clause:

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

You can compose statements of arbitrary complexity by combining `Q` objects with the `&`, `|`, and `^` operators and use parenthetical grouping. Also, `Q` objects can be negated using the `~` operator, allowing for combined lookups that combine both a normal query and a negated (`NOT`) query:

```
Q(question__startswith="Who") | ~Q(pub_date__year=2005)
```

Each lookup function that takes keyword-arguments (e.g. *`filter()`*, *`exclude()`*, *`get()`*) can also be passed one or more `Q` objects as positional (not-named) arguments. If you provide multiple `Q` object arguments to a lookup function, the arguments will be "AND"ed together. For example:

```
Poll.objects.get(
    Q(question__startswith="Who"),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
)
```

... roughly translates into the SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Lookup functions can mix the use of `Q` objects and keyword arguments. All arguments provided to a lookup function (be they keyword arguments or `Q` objects) are "AND"ed together. However, if a `Q` object is provided, it must precede the definition of any keyword arguments. For example:

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith="Who",
)
```

... would be a valid query, equivalent to the previous example; but:

```
# INVALID QUERY
Poll.objects.get(
    question__startswith="Who",
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
)
```

... would not be valid.

See also

The OR lookups examples in Django's unit tests show some possible uses of `Q`.

Comparing objects

To compare two model instances, use the standard Python comparison operator, the double equals sign: `==`. Behind the scenes, that compares the primary key values of two models.

Using the `Entry` example above, the following two statements are equivalent:

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

If a model's primary key isn't called `id`, no problem. Comparisons will always use the primary key, whatever it's called. For example, if a model's primary key field is called `name`, these two statements are equivalent:

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

Deleting objects

The delete method, conveniently, is named *delete()*. This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type. Example:

```
>>> e.delete()
(1, {'blog.Entry': 1})
```

You can also delete objects in bulk. Every *QuerySet* has a *delete()* method, which deletes all members of that *QuerySet*.

For example, this deletes all `Entry` objects with a `pub_date` year of 2005:

```
>>> Entry.objects.filter(pub_date__year=2005).delete()
(5, {'webapp.Entry': 5})
```

Keep in mind that this will, whenever possible, be executed purely in SQL, and so the `delete()` methods of individual object instances will not necessarily be called during the process. If you've provided a custom `delete()` method on a model class and want to ensure that it is called, you will need to "manually" delete instances of that model (e.g., by iterating over a *QuerySet* and calling `delete()` on each object individually) rather than using the bulk *delete()* method of a *QuerySet*.

When Django deletes an object, by default it emulates the behavior of the SQL constraint `ON DELETE CASCADE` – in other words, any objects which had foreign keys pointing at the object to be deleted will be deleted along with it. For example:

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

This cascade behavior is customizable via the *on_delete* argument to the *ForeignKey*.

Note that *delete()* is the only *QuerySet* method that is not exposed on a *Manager* itself. This is a safety mechanism to prevent you from accidentally requesting `Entry.objects.delete()`, and deleting all the entries. If you do want to delete all the objects, then you have to explicitly request a complete query set:

```
Entry.objects.all().delete()
```

Copying model instances

Although there is no built-in method for copying model instances, it is possible to easily create new instance with all fields' values copied. In the simplest case, you can set `pk` to `None` and `_state.adding` to `True`. Using our blog example:

```
blog = Blog(name="My blog", tagline="Blogging is easy")
blog.save()  # blog.pk == 1

blog.pk = None
blog._state.adding = True
blog.save()  # blog.pk == 2
```

Things get more complicated if you use inheritance. Consider a subclass of `Blog`:

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)


django_blog = ThemeBlog(name="Django", tagline="Django is easy", theme="python")
django_blog.save()  # django_blog.pk == 3
```

Due to how inheritance works, you have to set both `pk` and `id` to `None`, and `_state.adding` to `True`:

```
django_blog.pk = None
django_blog.id = None
django_blog._state.adding = True
django_blog.save()  # django_blog.pk == 4
```

This process doesn't copy relations that aren't part of the model's database table. For example, `Entry` has a `ManyToManyField` to `Author`. After duplicating an entry, you must set the many-to-many relations for the new entry:

```
entry = Entry.objects.all()[0]  # some previous entry
old_authors = entry.authors.all()
entry.pk = None
entry._state.adding = True
entry.save()
entry.authors.set(old_authors)
```

For a `OneToOneField`, you must duplicate the related object and assign it to the new object's field to avoid violating the one-to-one unique constraint. For example, assuming `entry` is already duplicated as above:

```
detail = EntryDetail.objects.all()[0]
detail.pk = None
detail._state.adding = True
detail.entry = entry
detail.save()
```

Updating multiple objects at once

Sometimes you want to set a field to a particular value for all the objects in a *QuerySet*. You can do this with the *update()* method. For example:

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline="Everything is the same")
```

You can only set non-relation fields and *ForeignKey* fields using this method. To update a non-relation field, provide the new value as a constant. To update *ForeignKey* fields, set the new value to be the new model instance you want to point to. For example:

```
>>> b = Blog.objects.get(pk=1)

# Change every Entry so that it belongs to this Blog.
>>> Entry.objects.update(blog=b)
```

The `update()` method is applied instantly and returns the number of rows matched by the query (which may not be equal to the number of rows updated if some rows already have the new value). The only restriction on the *QuerySet* being updated is that it can only access one database table: the model's main table. You can filter based on related fields, but you can only update columns in the model's main table. Example:

```
>>> b = Blog.objects.get(pk=1)

# Update all the headlines belonging to this Blog.
>>> Entry.objects.filter(blog=b).update(headline="Everything is the same")
```

Be aware that the `update()` method is converted directly to an SQL statement. It is a bulk operation for direct updates. It doesn't run any *save()* methods on your models, or emit the `pre_save` or `post_save` signals (which are a consequence of calling *save()*), or honor the *auto_now* field option. If you want to save every item in a *QuerySet* and make sure that the *save()* method is called on each instance, you don't need any special function to handle that. Loop over them and call *save()*:

```
for item in my_queryset:
    item.save()
```

Calls to update can also use *F expressions* to update one field based on the value of another field in the

model. This is especially useful for incrementing counters based upon their current value. For example, to increment the pingback count for every entry in the blog:

```
>>> Entry.objects.update(number_of_pingbacks=F("number_of_pingbacks") + 1)
```

However, unlike `F()` objects in filter and exclude clauses, you can't introduce joins when you use `F()` objects in an update – you can only reference fields local to the model being updated. If you attempt to introduce a join with an `F()` object, a `FieldError` will be raised:

```
# This will raise a FieldError
>>> Entry.objects.update(headline=F("blog__name"))
```

Related objects

When you define a relationship in a model (i.e., a *ForeignKey*, *OneToOneField*, or *ManyToManyField*), instances of that model will have a convenient API to access the related object(s).

Using the models at the top of this page, for example, an `Entry` object `e` can get its associated `Blog` object by accessing the `blog` attribute: `e.blog`.

(Behind the scenes, this functionality is implemented by Python descriptors. This shouldn't really matter to you, but we point it out here for the curious.)

Django also creates API accessors for the "other" side of the relationship – the link from the related model to the model that defines the relationship. For example, a `Blog` object `b` has access to a list of all related `Entry` objects via the `entry_set` attribute: `b.entry_set.all()`.

All examples in this section use the sample `Blog`, `Author` and `Entry` models defined at the top of this page.

One-to-many relationships

Forward

If a model has a *ForeignKey*, instances of that model will have access to the related (foreign) object via an attribute of the model.

Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog  # Returns the related Blog object.
```

You can get and set via a foreign-key attribute. As you may expect, changes to the foreign key aren't saved to the database until you call *save()*. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

If a *ForeignKey* field has `null=True` set (i.e., it allows `NULL` values), you can assign `None` to remove the relation. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save()  # "UPDATE blog_entry SET blog_id = NULL ...;"
```

Forward access to one-to-many relationships is cached the first time the related object is accessed. Subsequent accesses to the foreign key on the same object instance are cached. Example:

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog)  # Hits the database to retrieve the associated Blog.
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
```

Note that the *select_related() QuerySet* method recursively prepopulates the cache of all one-to-many relationships ahead of time. Example:

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
```

Following relationships "backward"

If a model has a *ForeignKey*, instances of the foreign-key model will have access to a *Manager* that returns all instances of the first model. By default, this *Manager* is named `FOO_set`, where `FOO` is the source model name, lowercased. This *Manager* returns `QuerySet` instances, which can be filtered and manipulated as described in the "Retrieving objects" section above.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all()  # Returns all Entry objects related to Blog.

# b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains="Lennon")
>>> b.entry_set.count()
```

You can override the `FOO_set` name by setting the *related_name* parameter in the *ForeignKey* definition. For example, if the `Entry` model was altered to `blog = ForeignKey(Blog, on_delete=models.CASCADE, related_name='entries')`, the above example code would look like this:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all()  # Returns all Entry objects related to Blog.
```

(continues on next page)

(continued from previous page)

```
# b.entries is a Manager that returns ``QuerySet`` instances.
>>> b.entries.filter(headline__contains="Lennon")
>>> b.entries.count()
```

Using a custom reverse manager

By default the *RelatedManager* used for reverse relations is a subclass of the default manager for that model. If you would like to specify a different manager for a given query you can use the following syntax:

```
from django.db import models


class Entry(models.Model):
    # ...
    objects = models.Manager()  # Default Manager
    entries = EntryManager()  # Custom Manager


b = Blog.objects.get(id=1)
b.entry_set(manager="entries").all()
```

If `EntryManager` performed default filtering in its `get_queryset()` method, that filtering would apply to the `all()` call.

Specifying a custom reverse manager also enables you to call its custom methods:

```
b.entry_set(manager="entries").is_published()
```

Interaction with prefetching

When calling *prefetch_related()* with a reverse relation, the default manager will be used. If you want to prefetch related objects using a custom reverse manager, use *Prefetch()*. For example:

```
from django.db.models import Prefetch

prefetch_manager = Prefetch("entry_set", queryset=Entry.entries.all())
Blog.objects.prefetch_related(prefetch_manager)
```

Additional methods to handle related objects

In addition to the *QuerySet* methods defined in "Retrieving objects" above, the *ForeignKey Manager* has additional methods used to handle the set of related objects. A synopsis of each is below, and complete details can be found in the related objects reference.

`add(obj1, obj2, ...)`
: Adds the specified model objects to the related object set.

`create(**kwargs)`
: Creates a new object, saves it and puts it in the related object set. Returns the newly created object.

`remove(obj1, obj2, ...)`
: Removes the specified model objects from the related object set.

`clear()`
: Removes all objects from the related object set.

`set(objs)`
: Replace the set of related objects.

To assign the members of a related set, use the `set()` method with an iterable of object instances. For example, if `e1` and `e2` are `Entry` instances:

```
b = Blog.objects.get(id=1)
b.entry_set.set([e1, e2])
```

If the `clear()` method is available, any preexisting objects will be removed from the `entry_set` before all objects in the iterable (in this case, a list) are added to the set. If the `clear()` method is not available, all objects in the iterable will be added without removing any existing elements.

Each "reverse" operation described in this section has an immediate effect on the database. Every addition, creation and deletion is immediately and automatically saved to the database.

Many-to-many relationships

Both ends of a many-to-many relationship get automatic API access to the other end. The API works similar to a "backward" one-to-many relationship, above.

One difference is in the attribute naming: The model that defines the *ManyToManyField* uses the attribute name of that field itself, whereas the "reverse" model uses the lowercased model name of the original model, plus `'_set'` (just like reverse one-to-many relationships).

An example makes this easier to understand:

```
e = Entry.objects.get(id=3)
e.authors.all()  # Returns all Author objects for this Entry.
e.authors.count()
```

(continues on next page)

(continued from previous page)

```
e.authors.filter(name__contains="John")

a = Author.objects.get(id=5)
a.entry_set.all()  # Returns all Entry objects for this Author.
```

Like *ForeignKey*, *ManyToManyField* can specify *related_name*. In the above example, if the *ManyToManyField* in `Entry` had specified `related_name='entries'`, then each `Author` instance would have an `entries` attribute instead of `entry_set`.

Another difference from one-to-many relationships is that in addition to model instances, the `add()`, `set()`, and `remove()` methods on many-to-many relationships accept primary key values. For example, if `e1` and `e2` are `Entry` instances, then these `set()` calls work identically:

```
a = Author.objects.get(id=5)
a.entry_set.set([e1, e2])
a.entry_set.set([e1.pk, e2.pk])
```

One-to-one relationships

One-to-one relationships are very similar to many-to-one relationships. If you define a *OneToOneField* on your model, instances of that model will have access to the related object via an attribute of the model.

For example:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()


ed = EntryDetail.objects.get(id=2)
ed.entry  # Returns the related Entry object.
```

The difference comes in "reverse" queries. The related model in a one-to-one relationship also has access to a *Manager* object, but that *Manager* represents a single object, rather than a collection of objects:

```
e = Entry.objects.get(id=2)
e.entrydetail  # returns the related EntryDetail object
```

If no object has been assigned to this relationship, Django will raise a `DoesNotExist` exception.

Instances can be assigned to the reverse relationship in the same way as you would assign the forward relationship:

```
e.entrydetail = ed
```

How are the backward relationships possible?

Other object-relational mappers require you to define relationships on both sides. The Django developers believe this is a violation of the DRY (Don't Repeat Yourself) principle, so Django only requires you to define the relationship on one end.

But how is this possible, given that a model class doesn't know which other model classes are related to it until those other model classes are loaded?

The answer lies in the *app registry*. When Django starts, it imports each application listed in *INSTALLED_APPS*, and then the `models` module inside each application. Whenever a new model class is created, Django adds backward-relationships to any related models. If the related models haven't been imported yet, Django keeps tracks of the relationships and adds them when the related models eventually are imported.

For this reason, it's particularly important that all the models you're using be defined in applications listed in *INSTALLED_APPS*. Otherwise, backwards relations may not work properly.

Queries over related objects

Queries involving related objects follow the same rules as queries involving normal value fields. When specifying the value for a query to match, you may use either an object instance itself, or the primary key value for the object.

For example, if you have a Blog object `b` with `id=5`, the following three queries would be identical:

```
Entry.objects.filter(blog=b)  # Query using object instance
Entry.objects.filter(blog=b.id)  # Query using id from instance
Entry.objects.filter(blog=5)  # Query using id directly
```

Falling back to raw SQL

If you find yourself needing to write an SQL query that is too complex for Django's database-mapper to handle, you can fall back on writing SQL by hand. Django has a couple of options for writing raw SQL queries; see Performing raw SQL queries.

Finally, it's important to note that the Django database layer is merely an interface to your database. You can access your database via other tools, programming languages or database frameworks; there's nothing Django-specific about your database.

3.2.3 Aggregation

The topic guide on Django's database-abstraction API described the way that you can use Django queries that create, retrieve, update and delete individual objects. However, sometimes you will need to retrieve values that are derived by summarizing or aggregating a collection of objects. This topic guide describes the ways that aggregate values can be generated and returned using Django queries.

Throughout this guide, we'll refer to the following models. These models are used to track the inventory for a series of online bookstores:

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()


class Publisher(models.Model):
    name = models.CharField(max_length=300)


class Book(models.Model):
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    pubdate = models.DateField()


class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
```

Cheat sheet

In a hurry? Here's how to do common aggregate queries, assuming the models above:

```
# Total number of books.
>>> Book.objects.count()
2452
```

(continues on next page)

(continued from previous page)

```
# Total number of books with publisher=BaloneyPress
>>> Book.objects.filter(publisher__name="BaloneyPress").count()
73

# Average price across all books, provide default to be returned instead
# of None if no books exist.
>>> from django.db.models import Avg
>>> Book.objects.aggregate(Avg("price", default=0))
{'price__avg': 34.35}

# Max price across all books, provide default to be returned instead of
# None if no books exist.
>>> from django.db.models import Max
>>> Book.objects.aggregate(Max("price", default=0))
{'price__max': Decimal('81.20')}

# Difference between the highest priced book and the average price of all books.
>>> from django.db.models import FloatField
>>> Book.objects.aggregate(
...     price_diff=Max("price", output_field=FloatField()) - Avg("price")
... )
{'price_diff': 46.85}

# All the following queries involve traversing the Book<->Publisher
# foreign key relationship backwards.

# Each publisher, each with a count of books as a "num_books" attribute.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count("book"))
>>> pubs
<QuerySet [<Publisher: BaloneyPress>, <Publisher: SalamiPress>, ...]>
>>> pubs[0].num_books
73

# Each publisher, with a separate count of books with a rating above and below 5
>>> from django.db.models import Q
>>> above_5 = Count("book", filter=Q(book__rating__gt=5))
>>> below_5 = Count("book", filter=Q(book__rating__lte=5))
>>> pubs = Publisher.objects.annotate(below_5=below_5).annotate(above_5=above_5)
```

(continues on next page)

(continued from previous page)

```
>>> pubs[0].above_5
23
>>> pubs[0].below_5
12


# The top 5 publishers, in order by number of books.
>>> pubs = Publisher.objects.annotate(num_books=Count("book")).order_by("-num_books")[:5]
>>> pubs[0].num_books
1323
```

Generating aggregates over a `QuerySet`

Django provides two ways to generate aggregates. The first way is to generate summary values over an entire `QuerySet`. For example, say you wanted to calculate the average price of all books available for sale. Django's query syntax provides a means for describing the set of all books:

```
>>> Book.objects.all()
```

What we need is a way to calculate summary values over the objects that belong to this `QuerySet`. This is done by appending an `aggregate()` clause onto the `QuerySet`:

```
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg("price"))
{'price__avg': 34.35}
```

The `all()` is redundant in this example, so this could be simplified to:

```
>>> Book.objects.aggregate(Avg("price"))
{'price__avg': 34.35}
```

The argument to the `aggregate()` clause describes the aggregate value that we want to compute - in this case, the average of the `price` field on the `Book` model. A list of the aggregate functions that are available can be found in the QuerySet reference.

`aggregate()` is a terminal clause for a `QuerySet` that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function. If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause:

```
>>> Book.objects.aggregate(average_price=Avg("price"))
{'average_price': 34.35}
```

If you want to generate more than one aggregate, you add another argument to the `aggregate()` clause. So, if we also wanted to know the maximum and minimum price of all books, we would issue the query:

```
>>> from django.db.models import Avg, Max, Min
>>> Book.objects.aggregate(Avg("price"), Max("price"), Min("price"))
{'price__avg': 34.35, 'price__max': Decimal('81.20'), 'price__min': Decimal('12.99')}
```

Generating aggregates for each item in a `QuerySet`

The second way to generate summary values is to generate an independent summary for each object in a *`QuerySet`*. For example, if you are retrieving a list of books, you may want to know how many authors contributed to each book. Each Book has a many-to-many relationship with the Author; we want to summarize this relationship for each book in the `QuerySet`.

Per-object summaries can be generated using the *`annotate()`* clause. When an `annotate()` clause is specified, each object in the `QuerySet` will be annotated with the specified values.

The syntax for these annotations is identical to that used for the *`aggregate()`* clause. Each argument to `annotate()` describes an aggregate that is to be calculated. For example, to annotate books with the number of authors:

```
# Build an annotated queryset
>>> from django.db.models import Count
>>> q = Book.objects.annotate(Count("authors"))
# Interrogate the first object in the queryset
>>> q[0]
<Book: The Definitive Guide to Django>
>>> q[0].authors__count
2
# Interrogate the second object in the queryset
>>> q[1]
<Book: Practical Django Projects>
>>> q[1].authors__count
1
```

As with `aggregate()`, the name for the annotation is automatically derived from the name of the aggregate function and the name of the field being aggregated. You can override this default name by providing an alias when you specify the annotation:

```
>>> q = Book.objects.annotate(num_authors=Count("authors"))
>>> q[0].num_authors
2
>>> q[1].num_authors
```

(continues on next page)

(continued from previous page)

```
1
```

Unlike `aggregate()`, `annotate()` is not a terminal clause. The output of the `annotate()` clause is a `QuerySet`; this `QuerySet` can be modified using any other `QuerySet` operation, including `filter()`, `order_by()`, or even additional calls to `annotate()`.

Combining multiple aggregations

Combining multiple aggregations with `annotate()` will yield the wrong results because joins are used instead of subqueries:

```
>>> book = Book.objects.first()
>>> book.authors.count()
2
>>> book.store_set.count()
3
>>> q = Book.objects.annotate(Count("authors"), Count("store"))
>>> q[0].authors__count
6
>>> q[0].store__count
6
```

For most aggregates, there is no way to avoid this problem, however, the *`Count`* aggregate has a `distinct` parameter that may help:

```
>>> q = Book.objects.annotate(
...     Count("authors", distinct=True), Count("store", distinct=True)
... )
>>> q[0].authors__count
2
>>> q[0].store__count
3
```

If in doubt, inspect the SQL query!

In order to understand what happens in your query, consider inspecting the `query` property of your `QuerySet`.

Joins and aggregates

So far, we have dealt with aggregates over fields that belong to the model being queried. However, sometimes the value you want to aggregate will belong to a model that is related to the model you are querying.

When specifying the field to be aggregated in an aggregate function, Django will allow you to use the same double underscore notation that is used when referring to related fields in filters. Django will then handle any table joins that are required to retrieve and aggregate the related value.

For example, to find the price range of books offered in each store, you could use the annotation:

```
>>> from django.db.models import Max, Min
>>> Store.objects.annotate(min_price=Min("books__price"), max_price=Max("books__price"))
```

This tells Django to retrieve the `Store` model, join (through the many-to-many relationship) with the `Book` model, and aggregate on the price field of the book model to produce a minimum and maximum value.

The same rules apply to the `aggregate()` clause. If you wanted to know the lowest and highest price of any book that is available for sale in any of the stores, you could use the aggregate:

```
>>> Store.objects.aggregate(min_price=Min("books__price"), max_price=Max("books__price"))
```

Join chains can be as deep as you require. For example, to extract the age of the youngest author of any book available for sale, you could issue the query:

```
>>> Store.objects.aggregate(youngest_age=Min("books__authors__age"))
```

Following relationships backwards

In a way similar to Lookups that span relationships, aggregations and annotations on fields of models or models that are related to the one you are querying can include traversing "reverse" relationships. The lowercase name of related models and double-underscores are used here too.

For example, we can ask for all publishers, annotated with their respective total book stock counters (note how we use `'book'` to specify the `Publisher` -> `Book` reverse foreign key hop):

```
>>> from django.db.models import Avg, Count, Min, Sum
>>> Publisher.objects.annotate(Count("book"))
```

(Every `Publisher` in the resulting `QuerySet` will have an extra attribute called `book__count`.)

We can also ask for the oldest book of any of those managed by every publisher:

```
>>> Publisher.objects.aggregate(oldest_pubdate=Min("book__pubdate"))
```

(The resulting dictionary will have a key called `'oldest_pubdate'`. If no such alias were specified, it would be the rather long `'book__pubdate__min'`.)

This doesn't apply just to foreign keys. It also works with many-to-many relations. For example, we can ask for every author, annotated with the total number of pages considering all the books the author has (co-)authored (note how we use `'book'` to specify the `Author` -> `Book` reverse many-to-many hop):

```
>>> Author.objects.annotate(total_pages=Sum("book__pages"))
```

(Every `Author` in the resulting `QuerySet` will have an extra attribute called `total_pages`. If no such alias were specified, it would be the rather long `book__pages__sum`.)

Or ask for the average rating of all the books written by author(s) we have on file:

```
>>> Author.objects.aggregate(average_rating=Avg("book__rating"))
```

(The resulting dictionary will have a key called `'average_rating'`. If no such alias were specified, it would be the rather long `'book__rating__avg'`.)

Aggregations and other `QuerySet` clauses

`filter()` and `exclude()`

Aggregates can also participate in filters. Any `filter()` (or `exclude()`) applied to normal model fields will have the effect of constraining the objects that are considered for aggregation.

When used with an `annotate()` clause, a filter has the effect of constraining the objects for which an annotation is calculated. For example, you can generate an annotated list of all books that have a title starting with "Django" using the query:

```
>>> from django.db.models import Avg, Count
>>> Book.objects.filter(name__startswith="Django").annotate(num_authors=Count("authors"))
```

When used with an `aggregate()` clause, a filter has the effect of constraining the objects over which the aggregate is calculated. For example, you can generate the average price of all books with a title that starts with "Django" using the query:

```
>>> Book.objects.filter(name__startswith="Django").aggregate(Avg("price"))
```

Filtering on annotations

Annotated values can also be filtered. The alias for the annotation can be used in `filter()` and `exclude()` clauses in the same way as any other model field.

For example, to generate a list of books that have more than one author, you can issue the query:

```
>>> Book.objects.annotate(num_authors=Count("authors")).filter(num_authors__gt=1)
```

This query generates an annotated result set, and then generates a filter based upon that annotation.

If you need two annotations with two separate filters you can use the `filter` argument with any aggregate. For example, to generate a list of authors with a count of highly rated books:

```
>>> highly_rated = Count("book", filter=Q(book__rating__gte=7))
>>> Author.objects.annotate(num_books=Count("book"), highly_rated_books=highly_rated)
```

Each `Author` in the result set will have the `num_books` and `highly_rated_books` attributes. See also Conditional aggregation.

> **Choosing between `filter` and `QuerySet.filter()`**
>
> Avoid using the `filter` argument with a single annotation or aggregation. It's more efficient to use `QuerySet.filter()` to exclude rows. The aggregation `filter` argument is only useful when using two or more aggregations over the same relations with different conditionals.

Order of `annotate()` and `filter()` clauses

When developing a complex query that involves both `annotate()` and `filter()` clauses, pay particular attention to the order in which the clauses are applied to the `QuerySet`.

When an `annotate()` clause is applied to a query, the annotation is computed over the state of the query up to the point where the annotation is requested. The practical implication of this is that `filter()` and `annotate()` are not commutative operations.

Given:

- Publisher A has two books with ratings 4 and 5.
- Publisher B has two books with ratings 1 and 4.
- Publisher C has one book with rating 1.

Here's an example with the `Count` aggregate:

```
>>> a, b = Publisher.objects.annotate(num_books=Count("book", distinct=True)).filter(
...     book__rating__gt=3.0
... )
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 2)

>>> a, b = Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count("book
↪"))
>>> a, a.num_books
```

(continues on next page)

(continued from previous page)

```
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 1)
```

Both queries return a list of publishers that have at least one book with a rating exceeding 3.0, hence publisher C is excluded.

In the first query, the annotation precedes the filter, so the filter has no effect on the annotation. `distinct=True` is required to avoid a query bug.

The second query counts the number of books that have a rating exceeding 3.0 for each publisher. The filter precedes the annotation, so the filter constrains the objects considered when calculating the annotation.

Here's another example with the `Avg` aggregate:

```
>>> a, b = Publisher.objects.annotate(avg_rating=Avg("book__rating")).filter(
...     book__rating__gt=3.0
... )
>>> a, a.avg_rating
(<Publisher: A>, 4.5)  # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 2.5)  # (1+4)/2

>>> a, b = Publisher.objects.filter(book__rating__gt=3.0).annotate(
...     avg_rating=Avg("book__rating")
... )
>>> a, a.avg_rating
(<Publisher: A>, 4.5)  # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 4.0)  # 4/1 (book with rating 1 excluded)
```

The first query asks for the average rating of all a publisher's books for publisher's that have at least one book with a rating exceeding 3.0. The second query asks for the average of a publisher's book's ratings for only those ratings exceeding 3.0.

It's difficult to intuit how the ORM will translate complex querysets into SQL queries so when in doubt, inspect the SQL with `str(queryset.query)` and write plenty of tests.

`order_by()`

Annotations can be used as a basis for ordering. When you define an `order_by()` clause, the aggregates you provide can reference any alias defined as part of an `annotate()` clause in the query.

For example, to order a `QuerySet` of books by the number of authors that have contributed to the book, you could use the following query:

```
>>> Book.objects.annotate(num_authors=Count("authors")).order_by("num_authors")
```

values()

Ordinarily, annotations are generated on a per-object basis - an annotated `QuerySet` will return one result for each object in the original `QuerySet`. However, when a `values()` clause is used to constrain the columns that are returned in the result set, the method for evaluating annotations is slightly different. Instead of returning an annotated result for each result in the original `QuerySet`, the original results are grouped according to the unique combinations of the fields specified in the `values()` clause. An annotation is then provided for each unique group; the annotation is computed over all members of the group.

For example, consider an author query that attempts to find out the average rating of books written by each author:

```
>>> Author.objects.annotate(average_rating=Avg("book__rating"))
```

This will return one result for each author in the database, annotated with their average book rating.

However, the result will be slightly different if you use a `values()` clause:

```
>>> Author.objects.values("name").annotate(average_rating=Avg("book__rating"))
```

In this example, the authors will be grouped by name, so you will only get an annotated result for each unique author name. This means if you have two authors with the same name, their results will be merged into a single result in the output of the query; the average will be computed as the average over the books written by both authors.

Order of annotate() and values() clauses

As with the `filter()` clause, the order in which `annotate()` and `values()` clauses are applied to a query is significant. If the `values()` clause precedes the `annotate()`, the annotation will be computed using the grouping described by the `values()` clause.

However, if the `annotate()` clause precedes the `values()` clause, the annotations will be generated over the entire query set. In this case, the `values()` clause only constrains the fields that are generated on output.

For example, if we reverse the order of the `values()` and `annotate()` clause from our previous example:

```
>>> Author.objects.annotate(average_rating=Avg("book__rating")).values(
...     "name", "average_rating"
... )
```

This will now yield one unique result for each author; however, only the author's name and the `average_rating` annotation will be returned in the output data.

You should also note that `average_rating` has been explicitly included in the list of values to be returned. This is required because of the ordering of the `values()` and `annotate()` clause.

If the `values()` clause precedes the `annotate()` clause, any annotations will be automatically added to the result set. However, if the `values()` clause is applied after the `annotate()` clause, you need to explicitly include the aggregate column.

Interaction with `order_by()`

Fields that are mentioned in the `order_by()` part of a queryset are used when selecting the output data, even if they are not otherwise specified in the `values()` call. These extra fields are used to group "like" results together and they can make otherwise identical result rows appear to be separate. This shows up, particularly, when counting things.

By way of example, suppose you have a model like this:

```
from django.db import models


class Item(models.Model):
    name = models.CharField(max_length=10)
    data = models.IntegerField()
```

If you want to count how many times each distinct `data` value appears in an ordered queryset, you might try this:

```
items = Item.objects.order_by("name")
# Warning: not quite correct!
items.values("data").annotate(Count("id"))
```

...which will group the `Item` objects by their common `data` values and then count the number of `id` values in each group. Except that it won't quite work. The ordering by `name` will also play a part in the grouping, so this query will group by distinct `(data, name)` pairs, which isn't what you want. Instead, you should construct this queryset:

```
items.values("data").annotate(Count("id")).order_by()
```

...clearing any ordering in the query. You could also order by, say, `data` without any harmful effects, since that is already playing a role in the query.

This behavior is the same as that noted in the queryset documentation for *`distinct()`* and the general rule is the same: normally you won't want extra columns playing a part in the result, so clear out the ordering, or at least make sure it's restricted only to those fields you also select in a `values()` call.

> **Note**
>
> You might reasonably ask why Django doesn't remove the extraneous columns for you. The main reason is consistency with `distinct()` and other places: Django never removes ordering constraints that you have specified (and we can't change those other methods' behavior, as that would violate our API stability policy).

Aggregating annotations

You can also generate an aggregate on the result of an annotation. When you define an `aggregate()` clause, the aggregates you provide can reference any alias defined as part of an `annotate()` clause in the query.

For example, if you wanted to calculate the average number of authors per book you first annotate the set of books with the author count, then aggregate that author count, referencing the annotation field:

```
>>> from django.db.models import Avg, Count
>>> Book.objects.annotate(num_authors=Count("authors")).aggregate(Avg("num_authors"))
{'num_authors__avg': 1.66}
```

Aggregating on empty querysets or groups

When an aggregation is applied to an empty queryset or grouping, the result defaults to its default parameter, typically `None`. This behavior occurs because aggregate functions return `NULL` when the executed query returns no rows.

You can specify a return value by providing the default argument for most aggregations. However, since *`Count`* does not support the default argument, it will always return `0` for empty querysets or groups.

For example, assuming that no book contains web in its name, calculating the total price for this book set would return `None` since there are no matching rows to compute the *`Sum`* aggregation on:

```
>>> from django.db.models import Sum
>>> Book.objects.filter(name__contains="web").aggregate(Sum("price"))
{"price__sum": None}
```

However, the default argument can be set when calling *`Sum`* to return a different default value if no books can be found:

```
>>> Book.objects.filter(name__contains="web").aggregate(Sum("price", default=0))
{"price__sum": Decimal("0")}
```

Under the hood, the default argument is implemented by wrapping the aggregate function with *`Coalesce`*.

3.2.4 Search

A common task for web applications is to search some data in the database with user input. In a simple case, this could be filtering a list of objects by a category. A more complex use case might require searching with weighting, categorization, highlighting, multiple languages, and so on. This document explains some of the possible use cases and the tools you can use.

We'll refer to the same models used in Making queries.

Use Cases

Standard textual queries

Text-based fields have a selection of matching operations. For example, you may wish to allow lookup up an author like so:

```
>>> Author.objects.filter(name__contains="Terry")
[<Author: Terry Gilliam>, <Author: Terry Jones>]
```

This is a very fragile solution as it requires the user to know an exact substring of the author's name. A better approach could be a case-insensitive match (*icontains*), but this is only marginally better.

A database's more advanced comparison functions

If you're using PostgreSQL, Django provides a selection of database specific tools to allow you to leverage more complex querying options. Other databases have different selections of tools, possibly via plugins or user-defined functions. Django doesn't include any support for them at this time. We'll use some examples from PostgreSQL to demonstrate the kind of functionality databases may have.

> **Searching in other databases**
>
> All of the searching tools provided by *django.contrib.postgres* are constructed entirely on public APIs such as custom lookups and database functions. Depending on your database, you should be able to construct queries to allow similar APIs. If there are specific things which cannot be achieved this way, please open a ticket.

In the above example, we determined that a case insensitive lookup would be more useful. When dealing with non-English names, a further improvement is to use *unaccented comparison*:

```
>>> Author.objects.filter(name__unaccent__icontains="Helen")
[<Author: Helen Mirren>, <Author: Helena Bonham Carter>, <Author: Hélène Joy>]
```

This shows another issue, where we are matching against a different spelling of the name. In this case we have an asymmetry though - a search for `Helen` will pick up `Helena` or `Hélène`, but not the reverse. Another option would be to use a *trigram_similar* comparison, which compares sequences of letters.

For example:

```
>>> Author.objects.filter(name__unaccent__lower__trigram_similar="Hélène")
[<Author: Helen Mirren>, <Author: Hélène Joy>]
```

Now we have a different problem - the longer name of "Helena Bonham Carter" doesn't show up as it is much longer. Trigram searches consider all combinations of three letters, and compares how many appear in both search and source strings. For the longer name, there are more combinations that don't appear in the source string, so it is no longer considered a close match.

The correct choice of comparison functions here depends on your particular data set, for example the language(s) used and the type of text being searched. All of the examples we've seen are on short strings where the user is likely to enter something close (by varying definitions) to the source data.

Document-based search

Standard database operations stop being a useful approach when you start considering large blocks of text. Whereas the examples above can be thought of as operations on a string of characters, full text search looks at the actual words. Depending on the system used, it's likely to use some of the following ideas:

- Ignoring "stop words" such as "a", "the", "and".
- Stemming words, so that "pony" and "ponies" are considered similar.
- Weighting words based on different criteria such as how frequently they appear in the text, or the importance of the fields, such as the title or keywords, that they appear in.

There are many alternatives for using searching software, some of the most prominent are Elastic and Solr. These are full document-based search solutions. To use them with data from Django models, you'll need a layer which translates your data into a textual document, including back-references to the database ids. When a search using the engine returns a certain document, you can then look it up in the database. There are a variety of third-party libraries which are designed to help with this process.

PostgreSQL support

PostgreSQL has its own full text search implementation built-in. While not as powerful as some other search engines, it has the advantage of being inside your database and so can easily be combined with other relational queries such as categorization.

The *`django.contrib.postgres`* module provides some helpers to make these queries. For example, a query might select all the blog entries which mention "cheese":

```
>>> Entry.objects.filter(body_text__search="cheese")
[<Entry: Cheese on Toast recipes>, <Entry: Pizza recipes>]
```

You can also filter on a combination of fields and on related models:

```
>>> Entry.objects.annotate(
...     search=SearchVector("blog__tagline", "body_text"),
... ).filter(search="cheese")
[
    <Entry: Cheese on Toast recipes>,
    <Entry: Pizza Recipes>,
    <Entry: Dairy farming in Argentina>,
]
```

See the `contrib.postgres` Full text search document for complete details.

3.2.5 Managers

class Manager

A `Manager` is the interface through which database query operations are provided to Django models. At least one `Manager` exists for every model in a Django application.

The way `Manager` classes work is documented in Making queries; this document specifically touches on model options that customize `Manager` behavior.

Manager names

By default, Django adds a `Manager` with the name `objects` to every Django model class. However, if you want to use `objects` as a field name, or if you want to use a name other than `objects` for the `Manager`, you can rename it on a per-model basis. To rename the `Manager` for a given class, define a class attribute of type `models.Manager()` on that model. For example:

```
from django.db import models


class Person(models.Model):
    # ...
    people = models.Manager()
```

Using this example model, `Person.objects` will generate an `AttributeError` exception, but `Person.people.all()` will provide a list of all `Person` objects.

Custom managers

You can use a custom `Manager` in a particular model by extending the base `Manager` class and instantiating your custom `Manager` in your model.

There are two reasons you might want to customize a `Manager`: to add extra `Manager` methods, and/or to modify the initial `QuerySet` the `Manager` returns.

Adding extra manager methods

Adding extra `Manager` methods is the preferred way to add "table-level" functionality to your models. (For "row-level" functionality – i.e., functions that act on a single instance of a model object – use Model methods, not custom `Manager` methods.)

For example, this custom `Manager` adds a method `with_counts()`:

```
from django.db import models
from django.db.models.functions import Coalesce


class PollManager(models.Manager):
    def with_counts(self):
        return self.annotate(num_responses=Coalesce(models.Count("response"), 0))


class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    objects = PollManager()


class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll, on_delete=models.CASCADE)
    # ...
```

With this example, you'd use `OpinionPoll.objects.with_counts()` to get a `QuerySet` of `OpinionPoll` objects with the extra `num_responses` attribute attached.

A custom `Manager` method can return anything you want. It doesn't have to return a `QuerySet`.

Another thing to note is that `Manager` methods can access `self.model` to get the model class to which they're attached.

Modifying a manager's initial `QuerySet`

A `Manager`'s base `QuerySet` returns all objects in the system. For example, using this model:

```
from django.db import models


class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

...the statement `Book.objects.all()` will return all books in the database.

You can override a `Manager`'s base `QuerySet` by overriding the `Manager.get_queryset()` method. `get_queryset()` should return a `QuerySet` with the properties you require.

For example, the following model has two `Managers` – one that returns all objects, and one that returns only the books by Roald Dahl:

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(author="Roald Dahl")


# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager()  # The default manager.
    dahl_objects = DahlBookManager()  # The Dahl-specific manager.
```

With this sample model, `Book.objects.all()` will return all books in the database, but `Book.dahl_objects.all()` will only return the ones written by Roald Dahl.

Because `get_queryset()` returns a `QuerySet` object, you can use `filter()`, `exclude()` and all the other `QuerySet` methods on it. So these statements are all legal:

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title="Matilda")
Book.dahl_objects.count()
```

This example also pointed out another interesting technique: using multiple managers on the same model. You can attach as many `Manager()` instances to a model as you'd like. This is a non-repetitive way to define common "filters" for your models.

For example:

```
class AuthorManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(role="A")


class EditorManager(models.Manager):
```

(continues on next page)

(continued from previous page)

```
    def get_queryset(self):
        return super().get_queryset().filter(role="E")


class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices={"A": _("Author"), "E": _("Editor")})
    people = models.Manager()
    authors = AuthorManager()
    editors = EditorManager()
```

This example allows you to request `Person.authors.all()`, `Person.editors.all()`, and `Person.people.all()`, yielding predictable results.

Default managers

`Model._default_manager`

If you use custom `Manager` objects, take note that the first `Manager` Django encounters (in the order in which they're defined in the model) has a special status. Django interprets the first `Manager` defined in a class as the "default" `Manager`, and several parts of Django (including *`dumpdata`*) will use that `Manager` exclusively for that model. As a result, it's a good idea to be careful in your choice of default manager in order to avoid a situation where overriding `get_queryset()` results in an inability to retrieve objects you'd like to work with.

You can specify a custom default manager using *`Meta.default_manager_name`*.

If you're writing some code that must handle an unknown model, for example, in a third-party app that implements a generic view, use this manager (or *`_base_manager`*) rather than assuming the model has an `objects` manager.

Base managers

`Model._base_manager`

Using managers for related object access

By default, Django uses an instance of the `Model._base_manager` manager class when accessing related objects (e.g. `choice.question`), not the `_default_manager` on the related object. This is because Django needs to be able to retrieve the related object, even if it would otherwise be filtered out (and hence be inaccessible) by the default manager.

If the normal base manager class (*`django.db.models.Manager`*) isn't appropriate for your circumstances, you can tell Django which class to use by setting *`Meta.base_manager_name`*.

Base managers aren't used when querying on related models, or when accessing a one-to-many or many-to-many relationship. For example, if the `Question` model from the tutorial had a `deleted` field and a base manager that filters out instances with `deleted=True`, a queryset like `Choice.objects.filter(question__name__startswith='What')` would include choices related to deleted questions.

Don't filter away any results in this type of manager subclass

This manager is used to access objects that are related to from some other model. In those situations, Django has to be able to see all the objects for the model it is fetching, so that anything which is referred to can be retrieved.

Therefore, you should not override `get_queryset()` to filter out any rows. If you do so, Django will return incomplete results.

Calling custom `QuerySet` methods from the manager

While most methods from the standard `QuerySet` are accessible directly from the `Manager`, this is only the case for the extra methods defined on a custom `QuerySet` if you also implement them on the `Manager`:

```
class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role="A")

    def editors(self):
        return self.filter(role="E")


class PersonManager(models.Manager):
    def get_queryset(self):
        return PersonQuerySet(self.model, using=self._db)

    def authors(self):
        return self.get_queryset().authors()

    def editors(self):
        return self.get_queryset().editors()


class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices={"A": _("Author"), "E": _("Editor")})
    people = PersonManager()
```

This example allows you to call both `authors()` and `editors()` directly from the manager `Person.people`.

Creating a manager with `QuerySet` methods

In lieu of the above approach which requires duplicating methods on both the `QuerySet` and the `Manager`, *`QuerySet.as_manager()`* can be used to create an instance of `Manager` with a copy of a custom `QuerySet`'s methods:

```
class Person(models.Model):
    ...
    people = PersonQuerySet.as_manager()
```

The `Manager` instance created by *`QuerySet.as_manager()`* will be virtually identical to the `PersonManager` from the previous example.

Not every `QuerySet` method makes sense at the `Manager` level; for instance we intentionally prevent the *`QuerySet.delete()`* method from being copied onto the `Manager` class.

Methods are copied according to the following rules:

- Public methods are copied by default.
- Private methods (starting with an underscore) are not copied by default.
- Methods with a `queryset_only` attribute set to `False` are always copied.
- Methods with a `queryset_only` attribute set to `True` are never copied.

For example:

```
class CustomQuerySet(models.QuerySet):
    # Available on both Manager and QuerySet.
    def public_method(self):
        return

    # Available only on QuerySet.
    def _private_method(self):
        return

    # Available only on QuerySet.
    def opted_out_public_method(self):
        return

    opted_out_public_method.queryset_only = True

    # Available on both Manager and QuerySet.
```

(continues on next page)

(continued from previous page)

```
    def _opted_in_private_method(self):
        return

    _opted_in_private_method.queryset_only = False
```

from_queryset()

classmethod from_queryset(queryset_class)

For advanced usage you might want both a custom `Manager` and a custom `QuerySet`. You can do that by calling `Manager.from_queryset()` which returns a subclass of your base `Manager` with a copy of the custom `QuerySet` methods:

```
class CustomManager(models.Manager):
    def manager_only_method(self):
        return


class CustomQuerySet(models.QuerySet):
    def manager_and_queryset_method(self):
        return


class MyModel(models.Model):
    objects = CustomManager.from_queryset(CustomQuerySet)()
```

You may also store the generated class into a variable:

```
MyManager = CustomManager.from_queryset(CustomQuerySet)


class MyModel(models.Model):
    objects = MyManager()
```

Custom managers and model inheritance

Here's how Django handles custom managers and model inheritance:

1. Managers from base classes are always inherited by the child class, using Python's normal name resolution order (names on the child class override all others; then come names on the first parent class, and so on).

2. If no managers are declared on a model and/or its parents, Django automatically creates the `objects` manager.
3. The default manager on a class is either the one chosen with *`Meta.default_manager_name`*, or the first manager declared on the model, or the default manager of the first parent model.

These rules provide the necessary flexibility if you want to install a collection of custom managers on a group of models, via an abstract base class, but still customize the default manager. For example, suppose you have this base class:

```
class AbstractBase(models.Model):
    # ...
    objects = CustomManager()

    class Meta:
        abstract = True
```

If you use this directly in a child class, `objects` will be the default manager if you declare no managers in the child class:

```
class ChildA(AbstractBase):
    # ...
    # This class has CustomManager as the default manager.
    pass
```

If you want to inherit from `AbstractBase`, but provide a different default manager, you can provide the default manager on the child class:

```
class ChildB(AbstractBase):
    # ...
    # An explicit default manager.
    default_manager = OtherManager()
```

Here, `default_manager` is the default. The `objects` manager is still available, since it's inherited, but isn't used as the default.

Finally for this example, suppose you want to add extra managers to the child class, but still use the default from `AbstractBase`. You can't add the new manager directly in the child class, as that would override the default and you would have to also explicitly include all the managers from the abstract base class. The solution is to put the extra managers in another base class and introduce it into the inheritance hierarchy after the defaults:

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        abstract = True


class ChildC(AbstractBase, ExtraManager):
    # ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
    pass
```

Note that while you can define a custom manager on the abstract model, you can't invoke any methods using the abstract model. That is:

```
ClassA.objects.do_something()
```

is legal, but:

```
AbstractBase.objects.do_something()
```

will raise an exception. This is because managers are intended to encapsulate logic for managing collections of objects. Since you can't have a collection of abstract objects, it doesn't make sense to be managing them. If you have functionality that applies to the abstract model, you should put that functionality in a `staticmethod` or `classmethod` on the abstract model.

Implementation concerns

Whatever features you add to your custom `Manager`, it must be possible to make a shallow copy of a `Manager` instance; i.e., the following code must work:

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django makes shallow copies of manager objects during certain queries; if your Manager cannot be copied, those queries will fail.

This won't be an issue for most custom managers. If you are just adding simple methods to your `Manager`, it is unlikely that you will inadvertently make instances of your `Manager` uncopyable. However, if you're overriding `__getattr__` or some other private method of your `Manager` object that controls object state, you should ensure that you don't affect the ability of your `Manager` to be copied.

3.2.6 Performing raw SQL queries

Django gives you two ways of performing raw SQL queries: you can use *Manager.raw()* to perform raw queries and return model instances, or you can avoid the model layer entirely and execute custom SQL directly.

> **Explore the ORM before using raw SQL!**
>
> The Django ORM provides many tools to express queries without writing raw SQL. For example:
>
> - The QuerySet API is extensive.
> - You can *annotate* and aggregate using many built-in database functions. Beyond those, you can create custom query expressions.
>
> Before using raw SQL, explore the ORM. Ask on one of the support channels to see if the ORM supports your use case.

> **Warning**
>
> You should be very careful whenever you write raw SQL. Every time you use it, you should properly escape any parameters that the user can control by using `params` in order to protect against SQL injection attacks. Please read more about SQL injection protection.

Performing raw queries

The `raw()` manager method can be used to perform raw SQL queries that return model instances:

`Manager.raw`(raw_query, params=(), translations=None)

This method takes a raw SQL query, executes it, and returns a `django.db.models.query.RawQuerySet` instance. This `RawQuerySet` instance can be iterated over like a normal *QuerySet* to provide object instances.

This is best illustrated with an example. Suppose you have the following model:

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

You could then execute custom SQL like so:

```
>>> for p in Person.objects.raw("SELECT * FROM myapp_person"):
...     print(p)
...
```

(continues on next page)

(continued from previous page)

```
John Smith
Jane Jones
```

This example isn't very exciting – it's exactly the same as running `Person.objects.all()`. However, `raw()` has a bunch of other options that make it very powerful.

Model table names

Where did the name of the `Person` table come from in that example?

By default, Django figures out a database table name by joining the model's "app label" – the name you used in `manage.py startapp` – to the model's class name, with an underscore between them. In the example we've assumed that the `Person` model lives in an app named `myapp`, so its table would be `myapp_person`.

For more details check out the documentation for the *db_table* option, which also lets you manually set the database table name.

Warning

No checking is done on the SQL statement that is passed in to `.raw()`. Django expects that the statement will return a set of rows from the database, but does nothing to enforce that. If the query does not return rows, a (possibly cryptic) error will result.

Warning

If you are performing queries on MySQL, note that MySQL's silent type coercion may cause unexpected results when mixing types. If you query on a string type column, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison. For example, if your table contains the values `'abc'`, `'def'` and you query for `WHERE mycolumn=0`, both rows will match. To prevent this, perform the correct typecasting before using the value in a query.

Mapping query fields to model fields

`raw()` automatically maps fields in the query to fields on the model.

The order of fields in your query doesn't matter. In other words, both of the following queries work identically:

```
>>> Person.objects.raw("SELECT id, first_name, last_name, birth_date FROM myapp_person")
>>> Person.objects.raw("SELECT last_name, birth_date, first_name, id FROM myapp_person")
```

Matching is done by name. This means that you can use SQL's `AS` clauses to map fields in the query to model fields. So if you had some other table that had `Person` data in it, you could easily map it into `Person` instances:

```
>>> Person.objects.raw(
...     """
...     SELECT first AS first_name,
...            last AS last_name,
...            bd AS birth_date,
...            pk AS id,
...     FROM some_other_table
...     """
... )
```

As long as the names match, the model instances will be created correctly.

Alternatively, you can map fields in the query to model fields using the `translations` argument to `raw()`. This is a dictionary mapping names of fields in the query to names of fields on the model. For example, the above query could also be written:

```
>>> name_map = {"first": "first_name", "last": "last_name", "bd": "birth_date", "pk": "id
↪"}
>>> Person.objects.raw("SELECT * FROM some_other_table", translations=name_map)
```

Index lookups

`raw()` supports indexing, so if you need only the first result you can write:

```
>>> first_person = Person.objects.raw("SELECT * FROM myapp_person")[0]
```

However, the indexing and slicing are not performed at the database level. If you have a large number of `Person` objects in your database, it is more efficient to limit the query at the SQL level:

```
>>> first_person = Person.objects.raw("SELECT * FROM myapp_person LIMIT 1")[0]
```

Deferring model fields

Fields may also be left out:

```
>>> people = Person.objects.raw("SELECT id, first_name FROM myapp_person")
```

The `Person` objects returned by this query will be deferred model instances (see *`defer()`*). This means that the fields that are omitted from the query will be loaded on demand. For example:

```
>>> for p in Person.objects.raw("SELECT id, first_name FROM myapp_person"):
...     print(
...         p.first_name,  # This will be retrieved by the original query
...         p.last_name,  # This will be retrieved on demand
...     )
...
John Smith
Jane Jones
```

From outward appearances, this looks like the query has retrieved both the first name and last name. However, this example actually issued 3 queries. Only the first names were retrieved by the `raw()` query – the last names were both retrieved on demand when they were printed.

There is only one field that you can't leave out - the primary key field. Django uses the primary key to identify model instances, so it must always be included in a raw query. A *`FieldDoesNotExist`* exception will be raised if you forget to include the primary key.

Adding annotations

You can also execute queries containing fields that aren't defined on the model. For example, we could use PostgreSQL's age() function to get a list of people with their ages calculated by the database:

```
>>> people = Person.objects.raw("SELECT *, age(birth_date) AS age FROM myapp_person")
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
...
John is 37.
Jane is 42.
...
```

You can often avoid using raw SQL to compute annotations by instead using a Func() expression.

Passing parameters into `raw()`

If you need to perform parameterized queries, you can use the `params` argument to `raw()`:

```
>>> lname = "Doe"
>>> Person.objects.raw("SELECT * FROM myapp_person WHERE last_name = %s", [lname])
```

`params` is a list or dictionary of parameters. You'll use `%s` placeholders in the query string for a list, or `%(key)s` placeholders for a dictionary (where `key` is replaced by a dictionary key), regardless of your database engine. Such placeholders will be replaced with parameters from the `params` argument.

> **Note**
>
> Dictionary params are not supported with the SQLite backend; with this backend, you must pass parameters as a list.

> **Warning**
>
> Do not use string formatting on raw queries or quote placeholders in your SQL strings!
>
> It's tempting to write the above query as:
>
> ```
> >>> query = "SELECT * FROM myapp_person WHERE last_name = %s" % lname
> >>> Person.objects.raw(query)
> ```
>
> You might also think you should write your query like this (with quotes around `%s`):
>
> ```
> >>> query = "SELECT * FROM myapp_person WHERE last_name = '%s'"
> ```
>
> Don't make either of these mistakes.
>
> As discussed in SQL injection protection, using the `params` argument and leaving the placeholders unquoted protects you from SQL injection attacks, a common exploit where attackers inject arbitrary SQL into your database. If you use string interpolation or quote the placeholder, you're at risk for SQL injection.

Executing custom SQL directly

Sometimes even *`Manager.raw()`* isn't quite enough: you might need to perform queries that don't map cleanly to models, or directly execute `UPDATE`, `INSERT`, or `DELETE` queries.

In these cases, you can always access the database directly, routing around the model layer entirely.

The object `django.db.connection` represents the default database connection. To use the database connection, call `connection.cursor()` to get a cursor object. Then, call `cursor.execute(sql, [params])` to execute the SQL and `cursor.fetchone()` or `cursor.fetchall()` to return the resulting rows.

For example:

```
from django.db import connection


def my_custom_sql(self):
    with connection.cursor() as cursor:
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
        row = cursor.fetchone()
```

(continues on next page)

(continued from previous page)

```
    return row
```

To protect against SQL injection, you must not include quotes around the `%s` placeholders in the SQL string.

Note that if you want to include literal percent signs in the query, you have to double them in the case you are passing parameters:

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id = %s", [self.id])
```

If you are using more than one database, you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections

with connections["my_db_alias"].cursor() as cursor:
    # Your code here
    ...
```

By default, the Python DB API will return results without their field names, which means you end up with a `list` of values, rather than a `dict`. At a small performance and memory cost, you can return results as a `dict` by using something like this:

```
def dictfetchall(cursor):
    """
    Return all rows from a cursor as a dict.
    Assume the column names are unique.
    """
    columns = [col[0] for col in cursor.description]
    return [dict(zip(columns, row)) for row in cursor.fetchall()]
```

Another option is to use `collections.namedtuple()` from the Python standard library. A `namedtuple` is a tuple-like object that has fields accessible by attribute lookup; it's also indexable and iterable. Results are immutable and accessible by field names or indices, which might be useful:

```
from collections import namedtuple


def namedtuplefetchall(cursor):
    """
```

(continues on next page)

(continued from previous page)

```
    Return all rows from a cursor as a namedtuple.
    Assume the column names are unique.
    """
    desc = cursor.description
    nt_result = namedtuple("Result", [col[0] for col in desc])
    return [nt_result(*row) for row in cursor.fetchall()]
```

The `dictfetchall()` and `namedtuplefetchall()` examples assume unique column names, since a cursor cannot distinguish columns from different tables.

Here is an example of the difference between the three:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2")
>>> cursor.fetchall()
((54360982, None), (54360880, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2")
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982}, {'parent_id': None, 'id': 54360880}]

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2")
>>> results = namedtuplefetchall(cursor)
>>> results
[Result(id=54360982, parent_id=None), Result(id=54360880, parent_id=None)]
>>> results[0].id
54360982
>>> results[0][0]
54360982
```

Connections and cursors

`connection` and `cursor` mostly implement the standard Python DB-API described in PEP 249 — except when it comes to transaction handling.

If you're not familiar with the Python DB-API, note that the SQL statement in `cursor.execute()` uses placeholders, `"%s"`, rather than adding parameters directly within the SQL. If you use this technique, the underlying database library will automatically escape your parameters as necessary.

Also note that Django expects the `"%s"` placeholder, not the `"?"` placeholder, which is used by the SQLite Python bindings. This is for the sake of consistency and sanity.

Using a cursor as a context manager:

```
with connection.cursor() as c:
    c.execute(...)
```

is equivalent to:

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

Calling stored procedures

`CursorWrapper.callproc`(procname, params=None, kparams=None)

Calls a database stored procedure with the given name. A sequence (`params`) or dictionary (`kparams`) of input parameters may be provided. Most databases don't support `kparams`. Of Django's built-in backends, only Oracle supports it.

For example, given this stored procedure in an Oracle database:

```
CREATE PROCEDURE "TEST_PROCEDURE"(v_i INTEGER, v_text NVARCHAR2(10)) AS
    p_i INTEGER;
    p_text NVARCHAR2(10);
BEGIN
    p_i := v_i;
    p_text := v_text;
    ...
END;
```

This will call it:

```
with connection.cursor() as cursor:
    cursor.callproc("test_procedure", [1, "test"])
```

3.2.7 Database transactions

Django gives you a few ways to control how database transactions are managed.

Managing database transactions

Django's default transaction behavior

Django's default behavior is to run in autocommit mode. Each query is immediately committed to the database, unless a transaction is active. See below for details.

Django uses transactions or savepoints automatically to guarantee the integrity of ORM operations that require multiple queries, especially delete() and update() queries.

Django's *TestCase* class also wraps each test in a transaction for performance reasons.

Tying transactions to HTTP requests

A common way to handle transactions on the web is to wrap each request in a transaction. Set *ATOMIC_REQUESTS* to `True` in the configuration of each database for which you want to enable this behavior.

It works like this. Before calling a view function, Django starts a transaction. If the response is produced without problems, Django commits the transaction. If the view produces an exception, Django rolls back the transaction.

You may perform subtransactions using savepoints in your view code, typically with the *atomic()* context manager. However, at the end of the view, either all or none of the changes will be committed.

> **Warning**
>
> While the simplicity of this transaction model is appealing, it also makes it inefficient when traffic increases. Opening a transaction for every view has some overhead. The impact on performance depends on the query patterns of your application and on how well your database handles locking.

> **Per-request transactions and streaming responses**
>
> When a view returns a *StreamingHttpResponse*, reading the contents of the response will often execute code to generate the content. Since the view has already returned, such code runs outside of the transaction.
>
> Generally speaking, it isn't advisable to write to the database while generating a streaming response, since there's no sensible way to handle errors after starting to send the response.

In practice, this feature wraps every view function in the *atomic()* decorator described below.

Note that only the execution of your view is enclosed in the transactions. Middleware runs outside of the transaction, and so does the rendering of template responses.

When *ATOMIC_REQUESTS* is enabled, it's still possible to prevent views from running in a transaction.

non_atomic_requests(using=None)

This decorator will negate the effect of *ATOMIC_REQUESTS* for a given view:

```
from django.db import transaction


@transaction.non_atomic_requests
def my_view(request):
    do_stuff()


@transaction.non_atomic_requests(using="other")
def my_other_view(request):
    do_stuff_on_the_other_database()
```

It only works if it's applied to the view itself.

Controlling transactions explicitly

Django provides a single API to control database transactions.

atomic(using=None, savepoint=True, durable=False)

Atomicity is the defining property of database transactions. `atomic` allows us to create a block of code within which the atomicity on the database is guaranteed. If the block of code is successfully completed, the changes are committed to the database. If there is an exception, the changes are rolled back.

`atomic` blocks can be nested. In this case, when an inner block completes successfully, its effects can still be rolled back if an exception is raised in the outer block at a later point.

It is sometimes useful to ensure an `atomic` block is always the outermost `atomic` block, ensuring that any database changes are committed when the block is exited without errors. This is known as durability and can be achieved by setting `durable=True`. If the `atomic` block is nested within another it raises a `RuntimeError`.

`atomic` is usable both as a decorator:

```
from django.db import transaction


@transaction.atomic
```

(continues on next page)

(continued from previous page)

```
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```

and as a context manager:

```
from django.db import transaction


def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

Wrapping `atomic` in a try/except block allows for natural handling of integrity errors:

```
from django.db import IntegrityError, transaction


@transaction.atomic
def viewfunc(request):
    create_parent()

    try:
        with transaction.atomic():
            generate_relationships()
    except IntegrityError:
        handle_exception()

    add_children()
```

In this example, even if `generate_relationships()` causes a database error by breaking an integrity constraint, you can execute queries in `add_children()`, and the changes from `create_parent()` are still there and bound to the same transaction. Note that any operations attempted in `generate_relationships()` will already have been rolled back safely when `handle_exception()` is called, so the exception handler can also operate on the database if necessary.

Avoid catching exceptions inside `atomic`!

When exiting an `atomic` block, Django looks at whether it's exited normally or with an exception to determine whether to commit or roll back. If you catch and handle exceptions inside an `atomic` block, you may hide from Django the fact that a problem has happened. This can result in unexpected behavior.

This is mostly a concern for *`DatabaseError`* and its subclasses such as *`IntegrityError`*. After such an error, the transaction is broken and Django will perform a rollback at the end of the `atomic` block. If you attempt to run database queries before the rollback happens, Django will raise a *`TransactionManagementError`*. You may also encounter this behavior when an ORM-related signal handler raises an exception.

The correct way to catch database errors is around an `atomic` block as shown above. If necessary, add an extra `atomic` block for this purpose. This pattern has another advantage: it delimits explicitly which operations will be rolled back if an exception occurs.

If you catch exceptions raised by raw SQL queries, Django's behavior is unspecified and database-dependent.

You may need to manually revert app state when rolling back a transaction.

The values of a model's fields won't be reverted when a transaction rollback happens. This could lead to an inconsistent model state unless you manually restore the original field values.

For example, given `MyModel` with an `active` field, this snippet ensures that the `if obj.active` check at the end uses the correct value if updating `active` to `True` fails in the transaction:

```
from django.db import DatabaseError, transaction

obj = MyModel(active=False)
obj.active = True
try:
    with transaction.atomic():
        obj.save()
except DatabaseError:
    obj.active = False

if obj.active:
    ...
```

This also applies to any other mechanism that may hold app state, such as caching or global variables. For example, if the code proactively updates data in the cache after saving an object, it's recommended to use transaction.on_commit() instead, to defer cache alterations until the transac-

tion is actually committed.

In order to guarantee atomicity, `atomic` disables some APIs. Attempting to commit, roll back, or change the autocommit state of the database connection within an `atomic` block will raise an exception.

`atomic` takes a `using` argument which should be the name of a database. If this argument isn't provided, Django uses the `"default"` database.

Under the hood, Django's transaction management code:

- opens a transaction when entering the outermost `atomic` block;
- creates a savepoint when entering an inner `atomic` block;
- releases or rolls back to the savepoint when exiting an inner block;
- commits or rolls back the transaction when exiting the outermost block.

You can disable the creation of savepoints for inner blocks by setting the `savepoint` argument to `False`. If an exception occurs, Django will perform the rollback when exiting the first parent block with a savepoint if there is one, and the outermost block otherwise. Atomicity is still guaranteed by the outer transaction. This option should only be used if the overhead of savepoints is noticeable. It has the drawback of breaking the error handling described above.

You may use `atomic` when autocommit is turned off. It will only use savepoints, even for the outermost block.

Performance considerations

Open transactions have a performance cost for your database server. To minimize this overhead, keep your transactions as short as possible. This is especially important if you're using *`atomic()`* in long-running processes, outside of Django's request / response cycle.

Autocommit

Why Django uses autocommit

In the SQL standards, each SQL query starts a transaction, unless one is already active. Such transactions must then be explicitly committed or rolled back.

This isn't always convenient for application developers. To alleviate this problem, most databases provide an autocommit mode. When autocommit is turned on and no transaction is active, each SQL query gets wrapped in its own transaction. In other words, not only does each such query start a transaction, but the transaction also gets automatically committed or rolled back, depending on whether the query succeeded.

PEP 249, the Python Database API Specification v2.0, requires autocommit to be initially turned off. Django overrides this default and turns autocommit on.

To avoid this, you can deactivate the transaction management, but it isn't recommended.

Deactivating transaction management

You can totally disable Django's transaction management for a given database by setting *AUTOCOMMIT* to `False` in its configuration. If you do this, Django won't enable autocommit, and won't perform any commits. You'll get the regular behavior of the underlying database library.

This requires you to commit explicitly every transaction, even those started by Django or by third-party libraries. Thus, this is best used in situations where you want to run your own transaction-controlling middleware or do something really strange.

Performing actions after commit

Sometimes you need to perform an action related to the current database transaction, but only if the transaction successfully commits. Examples might include a background task, an email notification, or a cache invalidation.

on_commit() allows you to register callbacks that will be executed after the open transaction is successfully committed:

on_commit(func, using=None, robust=False)

Pass a function, or any callable, to *on_commit()*:

```
from django.db import transaction


def send_welcome_email(): ...


transaction.on_commit(send_welcome_email)
```

Callbacks will not be passed any arguments, but you can bind them with `functools.partial()`:

```
from functools import partial

for user in users:
    transaction.on_commit(partial(send_invite_email, user=user))
```

Callbacks are called after the open transaction is successfully committed. If the transaction is instead rolled back (typically when an unhandled exception is raised in an *atomic()* block), the callback will be discarded, and never called.

If you call `on_commit()` while there isn't an open transaction, the callback will be executed immediately.

It's sometimes useful to register callbacks that can fail. Passing `robust=True` allows the next callbacks to be executed even if the current one throws an exception. All errors derived from Python's `Exception` class are caught and logged to the `django.db.backends.base` logger.

You can use *`TestCase.captureOnCommitCallbacks()`* to test callbacks registered with *`on_commit()`*.

Savepoints

Savepoints (i.e. nested *`atomic()`* blocks) are handled correctly. That is, an *`on_commit()`* callable registered after a savepoint (in a nested *`atomic()`* block) will be called after the outer transaction is committed, but not if a rollback to that savepoint or any previous savepoint occurred during the transaction:

```
with transaction.atomic():  # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    with transaction.atomic():  # Inner atomic block, create a savepoint
        transaction.on_commit(bar)

# foo() and then bar() will be called when leaving the outermost block
```

On the other hand, when a savepoint is rolled back (due to an exception being raised), the inner callable will not be called:

```
with transaction.atomic():  # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    try:
        with transaction.atomic():  # Inner atomic block, create a savepoint
            transaction.on_commit(bar)
            raise SomeError()  # Raising an exception - abort the savepoint
    except SomeError:
        pass

# foo() will be called, but not bar()
```

Order of execution

On-commit functions for a given transaction are executed in the order they were registered.

Exception handling

If one on-commit function registered with `robust=False` within a given transaction raises an uncaught exception, no later registered functions in that same transaction will run. This is the same behavior as if you'd executed the functions sequentially yourself without *on_commit()*.

Timing of execution

Your callbacks are executed after a successful commit, so a failure in a callback will not cause the transaction to roll back. They are executed conditionally upon the success of the transaction, but they are not part of the transaction. For the intended use cases (mail notifications, background tasks, etc.), this should be fine. If it's not (if your follow-up action is so critical that its failure should mean the failure of the transaction itself), then you don't want to use the *on_commit()* hook. Instead, you may want two-phase commit such as the psycopg Two-Phase Commit protocol support and the optional Two-Phase Commit Extensions in the Python DB-API specification.

Callbacks are not run until autocommit is restored on the connection following the commit (because otherwise any queries done in a callback would open an implicit transaction, preventing the connection from going back into autocommit mode).

When in autocommit mode and outside of an *atomic()* block, the function will run immediately, not on commit.

On-commit functions only work with autocommit mode and the *atomic()* (or *ATOMIC_REQUESTS*) transaction API. Calling *on_commit()* when autocommit is disabled and you are not within an atomic block will result in an error.

Use in tests

Django's *TestCase* class wraps each test in a transaction and rolls back that transaction after each test, in order to provide test isolation. This means that no transaction is ever actually committed, thus your *on_commit()* callbacks will never be run.

You can overcome this limitation by using *TestCase.captureOnCommitCallbacks()*. This captures your *on_commit()* callbacks in a list, allowing you to make assertions on them, or emulate the transaction committing by calling them.

Another way to overcome the limitation is to use *TransactionTestCase* instead of *TestCase*. This will mean your transactions are committed, and the callbacks will run. However *TransactionTestCase* flushes the database between tests, which is significantly slower than *TestCase*'s isolation.

Why no rollback hook?

A rollback hook is harder to implement robustly than a commit hook, since a variety of things can cause an implicit rollback.

For instance, if your database connection is dropped because your process was killed without a chance to shut down gracefully, your rollback hook will never run.

But there is a solution: instead of doing something during the atomic block (transaction) and then undoing it if the transaction fails, use *on_commit()* to delay doing it in the first place until after the transaction succeeds. It's a lot easier to undo something you never did in the first place!

Low-level APIs

> **Warning**
>
> Always prefer *atomic()* if possible at all. It accounts for the idiosyncrasies of each database and prevents invalid operations.
>
> The low level APIs are only useful if you're implementing your own transaction management.

Autocommit

Django provides an API in the *django.db.transaction* module to manage the autocommit state of each database connection.

get_autocommit(using=None)

set_autocommit(autocommit, using=None)

These functions take a `using` argument which should be the name of a database. If it isn't provided, Django uses the `"default"` database.

Autocommit is initially turned on. If you turn it off, it's your responsibility to restore it.

Once you turn autocommit off, you get the default behavior of your database adapter, and Django won't help you. Although that behavior is specified in PEP 249, implementations of adapters aren't always consistent with one another. Review the documentation of the adapter you're using carefully.

You must ensure that no transaction is active, usually by issuing a *commit()* or a *rollback()*, before turning autocommit back on.

Django will refuse to turn autocommit off when an *atomic()* block is active, because that would break atomicity.

Transactions

A transaction is an atomic set of database queries. Even if your program crashes, the database guarantees that either all the changes will be applied, or none of them.

Django doesn't provide an API to start a transaction. The expected way to start a transaction is to disable autocommit with *set_autocommit()*.

Once you're in a transaction, you can choose either to apply the changes you've performed until this point with *commit()*, or to cancel them with *rollback()*. These functions are defined in *django.db.transaction*.

commit(using=None)

rollback(using=None)

These functions take a `using` argument which should be the name of a database. If it isn't provided, Django uses the `"default"` database.

Django will refuse to commit or to rollback when an *atomic()* block is active, because that would break atomicity.

Savepoints

A savepoint is a marker within a transaction that enables you to roll back part of a transaction, rather than the full transaction. Savepoints are available with the SQLite, PostgreSQL, Oracle, and MySQL (when using the InnoDB storage engine) backends. Other backends provide the savepoint functions, but they're empty operations – they don't actually do anything.

Savepoints aren't especially useful if you are using autocommit, the default behavior of Django. However, once you open a transaction with *atomic()*, you build up a series of database operations awaiting a commit or rollback. If you issue a rollback, the entire transaction is rolled back. Savepoints provide the ability to perform a fine-grained rollback, rather than the full rollback that would be performed by `transaction.rollback()`.

When the *atomic()* decorator is nested, it creates a savepoint to allow partial commit or rollback. You're strongly encouraged to use *atomic()* rather than the functions described below, but they're still part of the public API, and there's no plan to deprecate them.

Each of these functions takes a `using` argument which should be the name of a database for which the behavior applies. If no `using` argument is provided then the `"default"` database is used.

Savepoints are controlled by three functions in *django.db.transaction*:

savepoint(using=None)

Creates a new savepoint. This marks a point in the transaction that is known to be in a "good" state. Returns the savepoint ID (`sid`).

savepoint_commit(sid, using=None)

Releases savepoint `sid`. The changes performed since the savepoint was created become part of the transaction.

savepoint_rollback(sid, using=None)

Rolls back the transaction to savepoint `sid`.

These functions do nothing if savepoints aren't supported or if the database is in autocommit mode.

In addition, there's a utility function:

clean_savepoints(using=None)

Resets the counter used to generate unique savepoint IDs.

The following example demonstrates the use of savepoints:

```
from django.db import transaction


# open a transaction
@transaction.atomic
def viewfunc(request):
    a.save()
    # transaction now contains a.save()

    sid = transaction.savepoint()

    b.save()
    # transaction now contains a.save() and b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # open transaction still contains a.save() and b.save()
    else:
        transaction.savepoint_rollback(sid)
        # open transaction now contains only a.save()
```

Savepoints may be used to recover from a database error by performing a partial rollback. If you're doing this inside an *atomic()* block, the entire block will still be rolled back, because it doesn't know you've handled the situation at a lower level! To prevent this, you can control the rollback behavior with the following functions.

get_rollback(using=None)

set_rollback(rollback, using=None)

Setting the rollback flag to `True` forces a rollback when exiting the innermost atomic block. This may be useful to trigger a rollback without raising an exception.

Setting it to `False` prevents such a rollback. Before doing that, make sure you've rolled back the transaction to a known-good savepoint within the current atomic block! Otherwise you're breaking atomicity and data corruption may occur.

Database-specific notes

Savepoints in SQLite

While SQLite supports savepoints, a flaw in the design of the `sqlite3` module makes them hardly usable.

When autocommit is enabled, savepoints don't make sense. When it's disabled, `sqlite3` commits implicitly before savepoint statements. (In fact, it commits before any statement other than `SELECT`, `INSERT`, `UPDATE`, `DELETE` and `REPLACE`.) This bug has two consequences:

- The low level APIs for savepoints are only usable inside a transaction i.e. inside an *`atomic()`* block.
- It's impossible to use *`atomic()`* when autocommit is turned off.

Transactions in MySQL

If you're using MySQL, your tables may or may not support transactions; it depends on your MySQL version and the table types you're using. (By "table types," we mean something like "InnoDB" or "MyISAM".) MySQL transaction peculiarities are outside the scope of this article, but the MySQL site has information on MySQL transactions.

If your MySQL setup does not support transactions, then Django will always function in autocommit mode: statements will be executed and committed as soon as they're called. If your MySQL setup does support transactions, Django will handle transactions as explained in this document.

Handling exceptions within PostgreSQL transactions

> **Note**
>
> This section is relevant only if you're implementing your own transaction management. This problem cannot occur in Django's default mode and *`atomic()`* handles it automatically.

Inside a transaction, when a call to a PostgreSQL cursor raises an exception (typically `IntegrityError`), all subsequent SQL in the same transaction will fail with the error "current transaction is aborted, queries ignored until end of transaction block". While the basic use of `save()` is unlikely to raise an exception in PostgreSQL, there are more advanced usage patterns which might, such as saving objects with unique fields, saving using the `force_insert`/`force_update` flag, or invoking custom SQL.

There are several ways to recover from this sort of error.

Transaction rollback

The first option is to roll back the entire transaction. For example:

```
a.save()  # Succeeds, but may be undone by transaction rollback
try:
    b.save()  # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save()  # Succeeds, but a.save() may have been undone
```

Calling `transaction.rollback()` rolls back the entire transaction. Any uncommitted database operations will be lost. In this example, the changes made by `a.save()` would be lost, even though that operation raised no error itself.

Savepoint rollback

You can use savepoints to control the extent of a rollback. Before performing a database operation that could fail, you can set or update the savepoint; that way, if the operation fails, you can roll back the single offending operation, rather than the entire transaction. For example:

```
a.save()  # Succeeds, and never undone by savepoint rollback
sid = transaction.savepoint()
try:
    b.save()  # Could throw exception
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save()  # Succeeds, and a.save() is never undone
```

In this example, `a.save()` will not be undone in the case where `b.save()` raises an exception.

3.2.8 Multiple databases

This topic guide describes Django's support for interacting with multiple databases. Most of the rest of Django's documentation assumes you are interacting with a single database. If you want to interact with multiple databases, you'll need to take some additional steps.

> See also
>
> See Multi-database support for information about testing with multiple databases.

Defining your databases

The first step to using more than one database with Django is to tell Django about the database servers you'll be using. This is done using the *DATABASES* setting. This setting maps database aliases, which are a way to refer to a specific database throughout Django, to a dictionary of settings for that specific connection. The settings in the inner dictionaries are described fully in the *DATABASES* documentation.

Databases can have any alias you choose. However, the alias `default` has special significance. Django uses the database with the alias of `default` when no other database has been selected.

The following is an example `settings.py` snippet defining two databases – a default PostgreSQL database and a MySQL database called `users`:

```
DATABASES = {
    "default": {
        "NAME": "app_data",
        "ENGINE": "django.db.backends.postgresql",
        "USER": "postgres_user",
        "PASSWORD": "s3krit",
    },
    "users": {
        "NAME": "user_data",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_user",
        "PASSWORD": "priv4te",
    },
}
```

If the concept of a `default` database doesn't make sense in the context of your project, you need to be careful to always specify the database that you want to use. Django requires that a `default` database entry be defined, but the parameters dictionary can be left blank if it will not be used. To do this, you must set up *DATABASE_ROUTERS* for all of your apps' models, including those in any contrib and third-party apps you're using, so that no queries are routed to the default database. The following is an example `settings.py` snippet defining two non-default databases, with the `default` entry intentionally left empty:

```
DATABASES = {
    "default": {},
    "users": {
        "NAME": "user_data",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_user",
        "PASSWORD": "superS3cret",
    },
```

(continues on next page)

(continued from previous page)

```
    "customers": {
        "NAME": "customer_data",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_cust",
        "PASSWORD": "veryPriv@ate",
    },
}
```

If you attempt to access a database that you haven't defined in your *DATABASES* setting, Django will raise a `django.utils.connection.ConnectionDoesNotExist` exception.

Synchronizing your databases

The *migrate* management command operates on one database at a time. By default, it operates on the `default` database, but by providing the *--database* option, you can tell it to synchronize a different database. So, to synchronize all models onto all databases in the first example above, you would need to call:

```
$ ./manage.py migrate
$ ./manage.py migrate --database=users
```

If you don't want every application to be synchronized onto a particular database, you can define a database router that implements a policy constraining the availability of particular models.

If, as in the second example above, you've left the `default` database empty, you must provide a database name each time you run *migrate*. Omitting the database name would raise an error. For the second example:

```
$ ./manage.py migrate --database=users
$ ./manage.py migrate --database=customers
```

Using other management commands

Most other `django-admin` commands that interact with the database operate in the same way as *migrate* – they only ever operate on one database at a time, using `--database` to control the database used.

An exception to this rule is the *makemigrations* command. It validates the migration history in the databases to catch problems with the existing migration files (which could be caused by editing them) before creating new migrations. By default, it checks only the `default` database, but it consults the *allow_migrate()* method of routers if any are installed.

Automatic database routing

The easiest way to use multiple databases is to set up a database routing scheme. The default routing scheme ensures that objects remain 'sticky' to their original database (i.e., an object retrieved from the `foo` database will be saved on the same database). The default routing scheme ensures that if a database isn't specified, all queries fall back to the `default` database.

You don't have to do anything to activate the default routing scheme – it is provided 'out of the box' on every Django project. However, if you want to implement more interesting database allocation behaviors, you can define and install your own database routers.

Database routers

A database Router is a class that provides up to four methods:

`db_for_read`(model, **hints)

Suggest the database that should be used for read operations for objects of type `model`.

If a database operation is able to provide any additional information that might assist in selecting a database, it will be provided in the `hints` dictionary. Details on valid hints are provided below.

Returns `None` if there is no suggestion.

`db_for_write`(model, **hints)

Suggest the database that should be used for writes of objects of type Model.

If a database operation is able to provide any additional information that might assist in selecting a database, it will be provided in the `hints` dictionary. Details on valid hints are provided below.

Returns `None` if there is no suggestion.

`allow_relation`(obj1, obj2, **hints)

Return `True` if a relation between `obj1` and `obj2` should be allowed, `False` if the relation should be prevented, or `None` if the router has no opinion. This is purely a validation operation, used by foreign key and many to many operations to determine if a relation should be allowed between two objects.

If no router has an opinion (i.e. all routers return `None`), only relations within the same database are allowed.

`allow_migrate`(db, app_label, model_name=None, **hints)

Determine if the migration operation is allowed to run on the database with alias `db`. Return `True` if the operation should run, `False` if it shouldn't run, or `None` if the router has no opinion.

The `app_label` positional argument is the label of the application being migrated.

`model_name` is set by most migration operations to the value of `model._meta.model_name` (the lowercased version of the model `__name__`) of the model being migrated. Its value is `None` for the *`RunPython`* and *`RunSQL`* operations unless they provide it using hints.

`hints` are used by certain operations to communicate additional information to the router.

When `model_name` is set, `hints` normally contains the model class under the key `'model'`. Note that it may be a historical model, and thus not have any custom attributes, methods, or managers. You should only rely on `_meta`.

This method can also be used to determine the availability of a model on a given database.

makemigrations always creates migrations for model changes, but if `allow_migrate()` returns `False`, any migration operations for the `model_name` will be silently skipped when running *migrate* on the `db`. Changing the behavior of `allow_migrate()` for models that already have migrations may result in broken foreign keys, extra tables, or missing tables. When *makemigrations* verifies the migration history, it skips databases where no app is allowed to migrate.

A router doesn't have to provide all these methods – it may omit one or more of them. If one of the methods is omitted, Django will skip that router when performing the relevant check.

Hints

The hints received by the database router can be used to decide which database should receive a given request.

At present, the only hint that will be provided is `instance`, an object instance that is related to the read or write operation that is underway. This might be the instance that is being saved, or it might be an instance that is being added in a many-to-many relation. In some cases, no instance hint will be provided at all. The router checks for the existence of an instance hint, and determine if that hint should be used to alter routing behavior.

Using routers

Database routers are installed using the *DATABASE_ROUTERS* setting. This setting defines a list of class names, each specifying a router that should be used by the base router (`django.db.router`).

The base router is used by Django's database operations to allocate database usage. Whenever a query needs to know which database to use, it calls the base router, providing a model and a hint (if available). The base router tries each router class in turn until one returns a database suggestion. If no routers return a suggestion, the base router tries the current *instance._state.db* of the hint instance. If no hint instance was provided, or *instance._state.db* is `None`, the base router will allocate the `default` database.

An example

Example purposes only!

This example is intended as a demonstration of how the router infrastructure can be used to alter database usage. It intentionally ignores some complex issues in order to demonstrate how routers are used.

This example won't work if any of the models in `myapp` contain relationships to models outside of the `other` database. Cross-database relationships introduce referential integrity problems that Django can't currently handle.

The primary/replica (referred to as master/slave by some databases) configuration described is also flawed – it doesn't provide any solution for handling replication lag (i.e., query inconsistencies introduced because of the time taken for a write to propagate to the replicas). It also doesn't consider the interaction of transactions with the database utilization strategy.

So - what does this mean in practice? Let's consider another sample configuration. This one will have several databases: one for the `auth` application, and all other apps using a primary/replica setup with two read replicas. Here are the settings specifying these databases:

```
DATABASES = {
    "default": {},
    "auth_db": {
        "NAME": "auth_db_name",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_user",
        "PASSWORD": "swordfish",
    },
    "primary": {
        "NAME": "primary_name",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_user",
        "PASSWORD": "spam",
    },
    "replica1": {
        "NAME": "replica1_name",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_user",
        "PASSWORD": "eggs",
    },
    "replica2": {
        "NAME": "replica2_name",
        "ENGINE": "django.db.backends.mysql",
        "USER": "mysql_user",
        "PASSWORD": "bacon",
    },
}
```

Now we'll need to handle routing. First we want a router that knows to send queries for the `auth` and `contenttypes` apps to `auth_db` (`auth` models are linked to `ContentType`, so they must be stored in the same database):

```
class AuthRouter:
    """
    A router to control all database operations on models in the
    auth and contenttypes applications.
    """

    route_app_labels = {"auth", "contenttypes"}

    def db_for_read(self, model, **hints):
        """
        Attempts to read auth and contenttypes models go to auth_db.
        """
        if model._meta.app_label in self.route_app_labels:
            return "auth_db"
        return None

    def db_for_write(self, model, **hints):
        """
        Attempts to write auth and contenttypes models go to auth_db.
        """
        if model._meta.app_label in self.route_app_labels:
            return "auth_db"
        return None

    def allow_relation(self, obj1, obj2, **hints):
        """
        Allow relations if a model in the auth or contenttypes apps is
        involved.
        """
        if (
            obj1._meta.app_label in self.route_app_labels
            or obj2._meta.app_label in self.route_app_labels
        ):
            return True
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Make sure the auth and contenttypes apps only appear in the
        'auth_db' database.
```

(continues on next page)

(continued from previous page)

```
        """
        if app_label in self.route_app_labels:
            return db == "auth_db"
        return None
```

And we also want a router that sends all other apps to the primary/replica configuration, and randomly chooses a replica to read from:

```
import random


class PrimaryReplicaRouter:
    def db_for_read(self, model, **hints):
        """
        Reads go to a randomly-chosen replica.
        """
        return random.choice(["replica1", "replica2"])

    def db_for_write(self, model, **hints):
        """
        Writes always go to primary.
        """
        return "primary"

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects are
        in the primary/replica pool.
        """
        db_set = {"primary", "replica1", "replica2"}
        if obj1._state.db in db_set and obj2._state.db in db_set:
            return True
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        All non-auth models end up in this pool.
        """
        return True
```

Finally, in the settings file, we add the following (substituting `path.to.` with the actual Python path to the

module(s) where the routers are defined):

```
DATABASE_ROUTERS = ["path.to.AuthRouter", "path.to.PrimaryReplicaRouter"]
```

The order in which routers are processed is significant. Routers will be queried in the order they are listed in the *DATABASE_ROUTERS* setting. In this example, the `AuthRouter` is processed before the `PrimaryReplicaRouter`, and as a result, decisions concerning the models in `auth` are processed before any other decision is made. If the *DATABASE_ROUTERS* setting listed the two routers in the other order, `PrimaryReplicaRouter.allow_migrate()` would be processed first. The catch-all nature of the PrimaryReplicaRouter implementation would mean that all models would be available on all databases.

With this setup installed, and all databases migrated as per Synchronizing your databases, lets run some Django code:

```
>>> # This retrieval will be performed on the 'auth_db' database
>>> fred = User.objects.get(username="fred")
>>> fred.first_name = "Frederick"

>>> # This save will also be directed to 'auth_db'
>>> fred.save()

>>> # These retrieval will be randomly allocated to a replica database
>>> dna = Person.objects.get(name="Douglas Adams")

>>> # A new object has no database allocation when created
>>> mh = Book(title="Mostly Harmless")

>>> # This assignment will consult the router, and set mh onto
>>> # the same database as the author object
>>> mh.author = dna

>>> # This save will force the 'mh' instance onto the primary database...
>>> mh.save()

>>> # ... but if we re-retrieve the object, it will come back on a replica
>>> mh = Book.objects.get(title="Mostly Harmless")
```

This example defined a router to handle interaction with models from the `auth` app, and other routers to handle interaction with all other apps. If you left your `default` database empty and don't want to define a catch-all database router to handle all apps not otherwise specified, your routers must handle the names of all apps in *INSTALLED_APPS* before you migrate. See Behavior of contrib apps for information about contrib apps that must be together in one database.

Manually selecting a database

Django also provides an API that allows you to maintain complete control over database usage in your code. A manually specified database allocation will take priority over a database allocated by a router.

Manually selecting a database for a `QuerySet`

You can select the database for a `QuerySet` at any point in the `QuerySet` "chain." Call `using()` on the `QuerySet` to get another `QuerySet` that uses the specified database.

`using()` takes a single argument: the alias of the database on which you want to run the query. For example:

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()

>>> # So will this.
>>> Author.objects.using("default")

>>> # This will run on the 'other' database.
>>> Author.objects.using("other")
```

Selecting a database for `save()`

Use the `using` keyword to `Model.save()` to specify to which database the data should be saved.

For example, to save an object to the `legacy_users` database, you'd use this:

```
>>> my_object.save(using="legacy_users")
```

If you don't specify `using`, the `save()` method will save into the default database allocated by the routers.

Moving an object from one database to another

If you've saved an instance to one database, it might be tempting to use `save(using=...)` as a way to migrate the instance to a new database. However, if you don't take appropriate steps, this could have some unexpected consequences.

Consider the following example:

```
>>> p = Person(name="Fred")
>>> p.save(using="first")  # (statement 1)
>>> p.save(using="second")  # (statement 2)
```

In statement 1, a new `Person` object is saved to the `first` database. At this time, `p` doesn't have a primary key, so Django issues an SQL `INSERT` statement. This creates a primary key, and Django assigns that primary key to `p`.

When the save occurs in statement 2, `p` already has a primary key value, and Django will attempt to use that primary key on the new database. If the primary key value isn't in use in the `second` database, then you won't have any problems – the object will be copied to the new database.

However, if the primary key of `p` is already in use on the `second` database, the existing object in the `second` database will be overridden when `p` is saved.

You can avoid this in two ways. First, you can clear the primary key of the instance. If an object has no primary key, Django will treat it as a new object, avoiding any loss of data on the `second` database:

```
>>> p = Person(name="Fred")
>>> p.save(using="first")
>>> p.pk = None  # Clear the primary key.
>>> p.save(using="second")  # Write a completely new object.
```

The second option is to use the `force_insert` option to `save()` to ensure that Django does an SQL `INSERT`:

```
>>> p = Person(name="Fred")
>>> p.save(using="first")
>>> p.save(using="second", force_insert=True)
```

This will ensure that the person named `Fred` will have the same primary key on both databases. If that primary key is already in use when you try to save onto the `second` database, an error will be raised.

Selecting a database to delete from

By default, a call to delete an existing object will be executed on the same database that was used to retrieve the object in the first place:

```
>>> u = User.objects.using("legacy_users").get(username="fred")
>>> u.delete()  # will delete from the `legacy_users` database
```

To specify the database from which a model will be deleted, pass a `using` keyword argument to the `Model.delete()` method. This argument works just like the `using` keyword argument to `save()`.

For example, if you're migrating a user from the `legacy_users` database to the `new_users` database, you might use these commands:

```
>>> user_obj.save(using="new_users")
>>> user_obj.delete(using="legacy_users")
```

Using managers with multiple databases

Use the `db_manager()` method on managers to give managers access to a non-default database.

For example, say you have a custom manager method that touches the database – `User.objects.create_user()`. Because `create_user()` is a manager method, not a `QuerySet` method, you can't do `User.objects.using('new_users').create_user()`. (The `create_user()` method is only available on `User.objects`, the manager, not on `QuerySet` objects derived from the manager.) The solution is to use `db_manager()`, like this:

```
User.objects.db_manager("new_users").create_user(...)
```

`db_manager()` returns a copy of the manager bound to the database you specify.

Using `get_queryset()` with multiple databases

If you're overriding `get_queryset()` on your manager, be sure to either call the method on the parent (using `super()`) or do the appropriate handling of the `_db` attribute on the manager (a string containing the name of the database to use).

For example, if you want to return a custom `QuerySet` class from the `get_queryset` method, you could do this:

```
class MyManager(models.Manager):
    def get_queryset(self):
        qs = CustomQuerySet(self.model)
        if self._db is not None:
            qs = qs.using(self._db)
        return qs
```

Exposing multiple databases in Django's admin interface

Django's admin doesn't have any explicit support for multiple databases. If you want to provide an admin interface for a model on a database other than that specified by your router chain, you'll need to write custom *`ModelAdmin`* classes that will direct the admin to use a specific database for content.

`ModelAdmin` objects have the following methods that require customization for multiple-database support:

```
class MultiDBModelAdmin(admin.ModelAdmin):
    # A handy constant for the name of the alternate database.
    using = "other"

    def save_model(self, request, obj, form, change):
        # Tell Django to save objects to the 'other' database.
        obj.save(using=self.using)
```

(continues on next page)

(continued from previous page)

```
    def delete_model(self, request, obj):
        # Tell Django to delete objects from the 'other' database
        obj.delete(using=self.using)

    def get_queryset(self, request):
        # Tell Django to look for objects on the 'other' database.
        return super().get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super().formfield_for_foreignkey(
            db_field, request, using=self.using, **kwargs
        )

    def formfield_for_manytomany(self, db_field, request, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super().formfield_for_manytomany(
            db_field, request, using=self.using, **kwargs
        )
```

The implementation provided here implements a multi-database strategy where all objects of a given type are stored on a specific database (e.g., all `User` objects are in the `other` database). If your usage of multiple databases is more complex, your `ModelAdmin` will need to reflect that strategy.

`InlineModelAdmin` objects can be handled in a similar fashion. They require three customized methods:

```
class MultiDBTabularInline(admin.TabularInline):
    using = "other"

    def get_queryset(self, request):
        # Tell Django to look for inline objects on the 'other' database.
        return super().get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super().formfield_for_foreignkey(
            db_field, request, using=self.using, **kwargs
```

(continues on next page)

(continued from previous page)

```
        )

    def formfield_for_manytomany(self, db_field, request, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super().formfield_for_manytomany(
            db_field, request, using=self.using, **kwargs
        )
```

Once you've written your model admin definitions, they can be registered with any `Admin` instance:

```
from django.contrib import admin
from myapp.models import Author, Book, Publisher

# Import our custom ModelAdmin and TabularInline from where they're defined.
from myproject.admin import MultiDBModelAdmin, MultiDBTabularInline


# Specialize the multi-db admin objects for use with specific models.
class BookInline(MultiDBTabularInline):
    model = Book


class PublisherAdmin(MultiDBModelAdmin):
    inlines = [BookInline]


admin.site.register(Author, MultiDBModelAdmin)
admin.site.register(Publisher, PublisherAdmin)

othersite = admin.AdminSite("othersite")
othersite.register(Publisher, MultiDBModelAdmin)
```

This example sets up two admin sites. On the first site, the `Author` and `Publisher` objects are exposed; `Publisher` objects have a tabular inline showing books published by that publisher. The second site exposes just publishers, without the inlines.

Using raw cursors with multiple databases

If you are using more than one database you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections

with connections["my_db_alias"].cursor() as cursor:
    ...
```

Limitations of multiple databases

Cross-database relations

Django doesn't currently provide any support for foreign key or many-to-many relationships spanning multiple databases. If you have used a router to partition models to different databases, any foreign key and many-to-many relationships defined by those models must be internal to a single database.

This is because of referential integrity. In order to maintain a relationship between two objects, Django needs to know that the primary key of the related object is valid. If the primary key is stored on a separate database, it's not possible to easily evaluate the validity of a primary key.

If you're using Postgres, SQLite, Oracle, or MySQL with InnoDB, this is enforced at the database integrity level – database level key constraints prevent the creation of relations that can't be validated.

However, if you're using MySQL with MyISAM tables, there is no enforced referential integrity; as a result, you may be able to 'fake' cross database foreign keys. However, this configuration is not officially supported by Django.

Behavior of contrib apps

Several contrib apps include models, and some apps depend on others. Since cross-database relationships are impossible, this creates some restrictions on how you can split these models across databases:

- each one of `contenttypes.ContentType`, `sessions.Session` and `sites.Site` can be stored in any database, given a suitable router.
- `auth` models — `User`, `Group` and `Permission` — are linked together and linked to `ContentType`, so they must be stored in the same database as `ContentType`.
- `admin` depends on `auth`, so its models must be in the same database as `auth`.
- `flatpages` and `redirects` depend on `sites`, so their models must be in the same database as `sites`.

In addition, some objects are automatically created just after *`migrate`* creates a table to hold them in a database:

- a default `Site`,

- a `ContentType` for each model (including those not stored in that database),
- the `Permission`s for each model (including those not stored in that database).

For common setups with multiple databases, it isn't useful to have these objects in more than one database. Common setups include primary/replica and connecting to external databases. Therefore, it's recommended to write a database router that allows synchronizing these three models to only one database. Use the same approach for contrib and third-party apps that don't need their tables in multiple databases.

> **Warning**
>
> If you're synchronizing content types to more than one database, be aware that their primary keys may not match across databases. This may result in data corruption or data loss.

3.2.9 Tablespaces

A common paradigm for optimizing performance in database systems is the use of tablespaces to organize disk layout.

> **Warning**
>
> Django does not create the tablespaces for you. Please refer to your database engine's documentation for details on creating and managing tablespaces.

Declaring tablespaces for tables

A tablespace can be specified for the table generated by a model by supplying the *db_tablespace* option inside the model's `class Meta`. This option also affects tables automatically created for *ManyToManyField*s in the model.

You can use the *DEFAULT_TABLESPACE* setting to specify a default value for *db_tablespace*. This is useful for setting a tablespace for the built-in Django apps and other applications whose code you cannot control.

Declaring tablespaces for indexes

You can pass the *db_tablespace* option to an `Index` constructor to specify the name of a tablespace to use for the index. For single field indexes, you can pass the *db_tablespace* option to a `Field` constructor to specify an alternate tablespace for the field's column index. If the column doesn't have an index, the option is ignored.

You can use the *DEFAULT_INDEX_TABLESPACE* setting to specify a default value for *db_tablespace*.

If *db_tablespace* isn't specified and you didn't set *DEFAULT_INDEX_TABLESPACE*, the index is created in the same tablespace as the tables.

An example

```
class TablespaceExample(models.Model):
    name = models.CharField(max_length=30, db_index=True, db_tablespace="indexes")
    data = models.CharField(max_length=255, db_index=True)
    shortcut = models.CharField(max_length=7)
    edges = models.ManyToManyField(to="self", db_tablespace="indexes")

    class Meta:
        db_tablespace = "tables"
        indexes = [models.Index(fields=["shortcut"], db_tablespace="other_indexes")]
```

In this example, the tables generated by the `TablespaceExample` model (i.e. the model table and the many-to-many table) would be stored in the `tables` tablespace. The index for the name field and the indexes on the many-to-many table would be stored in the `indexes` tablespace. The `data` field would also generate an index, but no tablespace for it is specified, so it would be stored in the model tablespace `tables` by default. The index for the `shortcut` field would be stored in the `other_indexes` tablespace.

Database support

PostgreSQL and Oracle support tablespaces. SQLite, MariaDB and MySQL don't.

When you use a backend that lacks support for tablespaces, Django ignores all tablespace-related options.

3.2.10 Database access optimization

Django's database layer provides various ways to help developers get the most out of their databases. This document gathers together links to the relevant documentation, and adds various tips, organized under a number of headings that outline the steps to take when attempting to optimize your database usage.

Profile first

As general programming practice, this goes without saying. Find out what queries you are doing and what they are costing you. Use *`QuerySet.explain()`* to understand how specific `QuerySet`s are executed by your database. You may also want to use an external project like django-debug-toolbar, or a tool that monitors your database directly.

Remember that you may be optimizing for speed or memory or both, depending on your requirements. Sometimes optimizing for one will be detrimental to the other, but sometimes they will help each other. Also, work that is done by the database process might not have the same cost (to you) as the same amount of work done in your Python process. It is up to you to decide what your priorities are, where the balance must lie, and profile all of these as required since this will depend on your application and server.

With everything that follows, remember to profile after every change to ensure that the change is a benefit, and a big enough benefit given the decrease in readability of your code. All of the suggestions below come with the caveat that in your circumstances the general principle might not apply, or might even be reversed.

Use standard DB optimization techniques

. . .including:

- Indexes. This is a number one priority, after you have determined from profiling what indexes should be added. Use *`Meta.indexes`* or *`Field.db_index`* to add these from Django. Consider adding indexes to fields that you frequently query using *`filter()`*, *`exclude()`*, *`order_by()`*, etc. as indexes may help to speed up lookups. Note that determining the best indexes is a complex database-dependent topic that will depend on your particular application. The overhead of maintaining an index may outweigh any gains in query speed.
- Appropriate use of field types.

We will assume you have done the things listed above. The rest of this document focuses on how to use Django in such a way that you are not doing unnecessary work. This document also does not address other optimization techniques that apply to all expensive operations, such as general purpose caching.

Understand `QuerySets`

Understanding QuerySets is vital to getting good performance with simple code. In particular:

Understand `QuerySet` evaluation

To avoid performance problems, it is important to understand:

- that QuerySets are lazy.
- when they are evaluated.
- how the data is held in memory.

Understand cached attributes

As well as caching of the whole `QuerySet`, there is caching of the result of attributes on ORM objects. In general, attributes that are not callable will be cached. For example, assuming the example blog models:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog  # Blog object is retrieved at this point
>>> entry.blog  # cached version, no DB access
```

But in general, callable attributes cause DB lookups every time:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()  # query performed
>>> entry.authors.all()  # query performed again
```

Be careful when reading template code - the template system does not allow use of parentheses, but will call callables automatically, hiding the above distinction.

Be careful with your own custom properties - it is up to you to implement caching when required, for example using the *cached_property* decorator.

Use the `with` template tag

To make use of the caching behavior of `QuerySet`, you may need to use the *with* template tag.

Use `iterator()`

When you have a lot of objects, the caching behavior of the `QuerySet` can cause a large amount of memory to be used. In this case, *iterator()* may help.

Use `explain()`

QuerySet.explain() gives you detailed information about how the database executes a query, including indexes and joins that are used. These details may help you find queries that could be rewritten more efficiently, or identify indexes that could be added to improve performance.

Do database work in the database rather than in Python

For instance:

- At the most basic level, use filter and exclude to do filtering in the database.
- Use *F expressions* to filter based on other fields within the same model.
- Use annotate to do aggregation in the database.

If these aren't enough to generate the SQL you need:

Use `RawSQL`

A less portable but more powerful method is the *RawSQL* expression, which allows some SQL to be explicitly added to the query. If that still isn't powerful enough:

Use raw SQL

Write your own custom SQL to retrieve data or populate models. Use `django.db.connection.queries` to find out what Django is writing for you and start from there.

Retrieve individual objects using a unique, indexed column

There are two reasons to use a column with *unique* or *db_index* when using *get()* to retrieve individual objects. First, the query will be quicker because of the underlying database index. Also, the query could run much slower if multiple objects match the lookup; having a unique constraint on the column guarantees this will never happen.

So using the example blog models:

```
>>> entry = Entry.objects.get(id=10)
```

will be quicker than:

```
>>> entry = Entry.objects.get(headline="News Item Title")
```

because `id` is indexed by the database and is guaranteed to be unique.

Doing the following is potentially quite slow:

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

First of all, `headline` is not indexed, which will make the underlying database fetch slower.

Second, the lookup doesn't guarantee that only one object will be returned. If the query matches more than one object, it will retrieve and transfer all of them from the database. This penalty could be substantial if hundreds or thousands of records are returned. The penalty will be compounded if the database lives on a separate server, where network overhead and latency also play a factor.

Retrieve everything at once if you know you will need it

Hitting the database multiple times for different parts of a single 'set' of data that you will need all parts of is, in general, less efficient than retrieving it all in one query. This is particularly important if you have a query that is executed in a loop, and could therefore end up doing many database queries, when only one was needed. So:

Use `QuerySet.select_related()` and `prefetch_related()`

Understand *`select_related()`* and *`prefetch_related()`* thoroughly, and use them:

- in managers and default managers where appropriate. Be aware when your manager is and is not used; sometimes this is tricky so don't make assumptions.
- in view code or other layers, possibly making use of *`prefetch_related_objects()`* where needed.

Don't retrieve things you don't need

Use `QuerySet.values()` and `values_list()`

When you only want a `dict` or `list` of values, and don't need ORM model objects, make appropriate usage of *`values()`*. These can be useful for replacing model objects in template code - as long as the dicts you supply have the same attributes as those used in the template, you are fine.

Use `QuerySet.defer()` and `only()`

Use *`defer()`* and *`only()`* if there are database columns you know that you won't need (or won't need in most cases) to avoid loading them. Note that if you do use them, the ORM will have to go and get them in a separate query, making this a pessimization if you use it inappropriately.

Don't be too aggressive in deferring fields without profiling as the database has to read most of the non-text, non-`VARCHAR` data from the disk for a single row in the results, even if it ends up only using a few columns. The `defer()` and `only()` methods are most useful when you can avoid loading a lot of text data or for fields that might take a lot of processing to convert back to Python. As always, profile first, then optimize.

Use `QuerySet.contains(obj)`

. . .if you only want to find out if `obj` is in the queryset, rather than `if obj in queryset`.

Use `QuerySet.count()`

. . .if you only want the count, rather than doing `len(queryset)`.

Use `QuerySet.exists()`

. . .if you only want to find out if at least one result exists, rather than `if queryset`.

But:

Don't overuse `contains()`, `count()`, and `exists()`

If you are going to need other data from the QuerySet, evaluate it immediately.

For example, assuming a `Group` model that has a many-to-many relation to `User`, the following code is optimal:

```
members = group.members.all()

if display_group_members:
    if members:
        if current_user in members:
            print("You and", len(members) - 1, "other users are members of this group.")
        else:
            print("There are", len(members), "members in this group.")

        for member in members:
            print(member.username)
    else:
        print("There are no members in this group.")
```

It is optimal because:

1. Since QuerySets are lazy, this does no database queries if `display_group_members` is `False`.
2. Storing `group.members.all()` in the `members` variable allows its result cache to be reused.
3. The line `if members:` causes `QuerySet.__bool__()` to be called, which causes the `group.members.all()` query to be run on the database. If there aren't any results, it will return `False`, otherwise `True`.
4. The line `if current_user in members:` checks if the user is in the result cache, so no additional database queries are issued.
5. The use of `len(members)` calls `QuerySet.__len__()`, reusing the result cache, so again, no database queries are issued.
6. The `for member` loop iterates over the result cache.

In total, this code does either one or zero database queries. The only deliberate optimization performed is using the `members` variable. Using `QuerySet.exists()` for the `if`, `QuerySet.contains()` for the `in`, or `QuerySet.count()` for the count would each cause additional queries.

Use `QuerySet.update()` and `delete()`

Rather than retrieve a load of objects, set some values, and save them individual, use a bulk SQL UPDATE statement, via QuerySet.update(). Similarly, do bulk deletes where possible.

Note, however, that these bulk update methods cannot call the `save()` or `delete()` methods of individual instances, which means that any custom behavior you have added for these methods will not be executed, including anything driven from the normal database object signals.

Use foreign key values directly

If you only need a foreign key value, use the foreign key value that is already on the object you've got, rather than getting the whole related object and taking its primary key. i.e. do:

```
entry.blog_id
```

instead of:

```
entry.blog.id
```

Don't order results if you don't care

Ordering is not free; each field to order by is an operation the database must perform. If a model has a default ordering (*`Meta.ordering`*) and you don't need it, remove it on a `QuerySet` by calling *`order_by()`* with no parameters.

Adding an index to your database may help to improve ordering performance.

Use bulk methods

Use bulk methods to reduce the number of SQL statements.

Create in bulk

When creating objects, where possible, use the *bulk_create()* method to reduce the number of SQL queries. For example:

```
Entry.objects.bulk_create(
    [
        Entry(headline="This is a test"),
        Entry(headline="This is only a test"),
    ]
)
```

. . .is preferable to:

```
Entry.objects.create(headline="This is a test")
Entry.objects.create(headline="This is only a test")
```

Note that there are a number of *caveats to this method*, so make sure it's appropriate for your use case.

Update in bulk

When updating objects, where possible, use the *bulk_update()* method to reduce the number of SQL queries. Given a list or queryset of objects:

```
entries = Entry.objects.bulk_create(
    [
        Entry(headline="This is a test"),
        Entry(headline="This is only a test"),
    ]
)
```

The following example:

```
entries[0].headline = "This is not a test"
entries[1].headline = "This is no longer a test"
Entry.objects.bulk_update(entries, ["headline"])
```

. . .is preferable to:

```
entries[0].headline = "This is not a test"
entries[0].save()
entries[1].headline = "This is no longer a test"
entries[1].save()
```

Note that there are a number of *caveats to this method*, so make sure it's appropriate for your use case.

Insert in bulk

When inserting objects into *ManyToManyFields*, use *add()* with multiple objects to reduce the number of SQL queries. For example:

```
my_band.members.add(me, my_friend)
```

...is preferable to:

```
my_band.members.add(me)
my_band.members.add(my_friend)
```

...where `Band` and `Artist` are models with a many-to-many relationship.

When inserting different pairs of objects into *ManyToManyField* or when the custom *through* table is defined, use *bulk_create()* method to reduce the number of SQL queries. For example:

```
PizzaToppingRelationship = Pizza.toppings.through
PizzaToppingRelationship.objects.bulk_create(
    [
        PizzaToppingRelationship(pizza=my_pizza, topping=pepperoni),
        PizzaToppingRelationship(pizza=your_pizza, topping=pepperoni),
        PizzaToppingRelationship(pizza=your_pizza, topping=mushroom),
    ],
    ignore_conflicts=True,
)
```

...is preferable to:

```
my_pizza.toppings.add(pepperoni)
your_pizza.toppings.add(pepperoni, mushroom)
```

...where `Pizza` and `Topping` have a many-to-many relationship. Note that there are a number of *caveats to this method*, so make sure it's appropriate for your use case.

Remove in bulk

When removing objects from *ManyToManyFields*, use *remove()* with multiple objects to reduce the number of SQL queries. For example:

```
my_band.members.remove(me, my_friend)
```

. . .is preferable to:

```
my_band.members.remove(me)
my_band.members.remove(my_friend)
```

. . .where `Band` and `Artist` are models with a many-to-many relationship.

When removing different pairs of objects from *ManyToManyFields*, use *delete()* on a *Q* expression with multiple *through* model instances to reduce the number of SQL queries. For example:

```
from django.db.models import Q

PizzaToppingRelationship = Pizza.toppings.through
PizzaToppingRelationship.objects.filter(
    Q(pizza=my_pizza, topping=pepperoni)
    | Q(pizza=your_pizza, topping=pepperoni)
    | Q(pizza=your_pizza, topping=mushroom)
).delete()
```

. . .is preferable to:

```
my_pizza.toppings.remove(pepperoni)
your_pizza.toppings.remove(pepperoni, mushroom)
```

. . .where `Pizza` and `Topping` have a many-to-many relationship.

3.2.11 Database instrumentation

To help you understand and control the queries issued by your code, Django provides a hook for installing wrapper functions around the execution of database queries. For example, wrappers can count queries, measure query duration, log queries, or even prevent query execution (e.g. to make sure that no queries are issued while rendering a template with prefetched data).

The wrappers are modeled after middleware – they are callables which take another callable as one of their arguments. They call that callable to invoke the (possibly wrapped) database query, and they can do what they want around that call. They are, however, created and installed by user code, and so don't need a separate factory like middleware do.

Installing a wrapper is done in a context manager – so the wrappers are temporary and specific to some flow in your code.

As mentioned above, an example of a wrapper is a query execution blocker. It could look like this:

```
def blocker(*args):
    raise Exception("No database access allowed here.")
```

And it would be used in a view to block queries from the template like so:

```
from django.db import connection
from django.shortcuts import render


def my_view(request):
    context = {...}  # Code to generate context with all data.
    template_name = ...
    with connection.execute_wrapper(blocker):
        return render(request, template_name, context)
```

The parameters sent to the wrappers are:

- `execute` – a callable, which should be invoked with the rest of the parameters in order to execute the query.
- `sql` – a `str`, the SQL query to be sent to the database.
- `params` – a list/tuple of parameter values for the SQL command, or a list/tuple of lists/tuples if the wrapped call is `executemany()`.
- `many` – a `bool` indicating whether the ultimately invoked call is `execute()` or `executemany()` (and whether `params` is expected to be a sequence of values, or a sequence of sequences of values).
- `context` – a dictionary with further data about the context of invocation. This includes the connection and cursor.

Using the parameters, a slightly more complex version of the blocker could include the connection name in the error message:

```
def blocker(execute, sql, params, many, context):
    alias = context["connection"].alias
    raise Exception("Access to database '{}' blocked here".format(alias))
```

For a more complete example, a query logger could look like this:

```
import time
```

(continues on next page)

(continued from previous page)

```
class QueryLogger:
    def __init__(self):
        self.queries = []

    def __call__(self, execute, sql, params, many, context):
        current_query = {"sql": sql, "params": params, "many": many}
        start = time.monotonic()
        try:
            result = execute(sql, params, many, context)
        except Exception as e:
            current_query["status"] = "error"
            current_query["exception"] = e
            raise
        else:
            current_query["status"] = "ok"
            return result
        finally:
            duration = time.monotonic() - start
            current_query["duration"] = duration
            self.queries.append(current_query)
```

To use this, you would create a logger object and install it as a wrapper:

```
from django.db import connection

ql = QueryLogger()
with connection.execute_wrapper(ql):
    do_queries()
# Now we can print the log.
print(ql.queries)
```

`connection.execute_wrapper()`

execute_wrapper(wrapper)

Returns a context manager which, when entered, installs a wrapper around database query executions, and when exited, removes the wrapper. The wrapper is installed on the thread-local connection object.

`wrapper` is a callable taking five arguments. It is called for every query execution in the scope of the context manager, with arguments `execute`, `sql`, `params`, `many`, and `context` as described above. It's expected to call `execute(sql, params, many, context)` and return the return value of that call.

3.2.12 Fixtures

A fixture is a collection of files that contain the serialized contents of the database. Each fixture has a unique name, and the files that comprise the fixture can be distributed over multiple directories, in multiple applications.

See also

- How to provide initial data for models

How to produce a fixture

Fixtures can be generated by *manage.py dumpdata*. It's also possible to generate custom fixtures by directly using serialization tools or even by handwriting them.

How to use a fixture

Fixtures can be used to pre-populate the database with data for tests:

```
class MyTestCase(TestCase):
    fixtures = ["fixture-label"]
```

or to provide some initial data using the *loaddata* command:

```
django-admin loaddata <fixture label>
```

How fixtures are discovered

Django will search in these locations for fixtures:

1. In the `fixtures` directory of every installed application
2. In any directory listed in the *FIXTURE_DIRS* setting
3. In the literal path named by the fixture

Django will load any and all fixtures it finds in these locations that match the provided fixture names. If the named fixture has a file extension, only fixtures of that type will be loaded. For example:

```
django-admin loaddata mydata.json
```

would only load JSON fixtures called `mydata`. The fixture extension must correspond to the registered name of a serializer (e.g., `json` or `xml`).

If you omit the extensions, Django will search all available fixture types for a matching fixture. For example:

```
django-admin loaddata mydata
```

would look for any fixture of any fixture type called `mydata`. If a fixture directory contained `mydata.json`, that fixture would be loaded as a JSON fixture.

The fixtures that are named can include directory components. These directories will be included in the search path. For example:

```
django-admin loaddata foo/bar/mydata.json
```

would search `<app_label>/fixtures/foo/bar/mydata.json` for each installed application, `<dirname>/foo/bar/mydata.json` for each directory in *FIXTURE_DIRS*, and the literal path `foo/bar/mydata.json`.

Fixtures loading order

Multiple fixtures can be specified in the same invocation. For example:

```
django-admin loaddata mammals birds insects
```

or in a test case class:

```
class AnimalTestCase(TestCase):
    fixtures = ["mammals", "birds", "insects"]
```

The order in which fixtures are loaded follows the order in which they are listed, whether it's when using the management command or when listing them in the test case class as shown above.

In these examples, all the fixtures named `mammals` from all applications (in the order in which applications are defined in *INSTALLED_APPS*) will be loaded first. Subsequently, all the `birds` fixtures will be loaded, followed by all the `insects` fixtures.

Be aware that if the database backend supports row-level constraints, these constraints will be checked at the end of the transaction. Any relationships across fixtures may result in a load error if the database configuration does not support deferred constraint checking (refer to the MySQL docs for an example).

How fixtures are saved to the database

When fixture files are processed, the data is saved to the database as is. Model defined *save()* methods are not called, and any *pre_save* or *post_save* signals will be called with `raw=True` since the instance only contains attributes that are local to the model. You may, for example, want to disable handlers that access related fields that aren't present during fixture loading and would otherwise raise an exception:

```
from django.db.models.signals import post_save
from .models import MyModel
```

(continues on next page)

(continued from previous page)

```
def my_handler(**kwargs):
    # disable the handler during fixture loading
    if kwargs["raw"]:
        return
    ...


post_save.connect(my_handler, sender=MyModel)
```

You could also write a decorator to encapsulate this logic:

```
from functools import wraps


def disable_for_loaddata(signal_handler):
    """
    Decorator that turns off signal handlers when loading fixture data.
    """

    @wraps(signal_handler)
    def wrapper(*args, **kwargs):
        if kwargs["raw"]:
            return
        signal_handler(*args, **kwargs)

    return wrapper


@disable_for_loaddata
def my_handler(**kwargs): ...
```

Just be aware that this logic will disable the signals whenever fixtures are deserialized, not just during *loaddata*.

Compressed fixtures

Fixtures may be compressed in `zip`, `gz`, `bz2`, `lzma`, or `xz` format. For example:

```
django-admin loaddata mydata.json
```

would look for any of `mydata.json`, `mydata.json.zip`, `mydata.json.gz`, `mydata.json.bz2`, `mydata.json.`

`lzma`, or `mydata.json.xz`. The first file contained within a compressed archive is used.

Note that if two fixtures with the same name but different fixture type are discovered (for example, if `mydata.json` and `mydata.xml.gz` were found in the same fixture directory), fixture installation will be aborted, and any data installed in the call to *`loaddata`* will be removed from the database.

> **MySQL with MyISAM and fixtures**
>
> The MyISAM storage engine of MySQL doesn't support transactions or constraints, so if you use MyISAM, you won't get validation of fixture data, or a rollback if multiple transaction files are found.

Database-specific fixtures

If you're in a multi-database setup, you might have fixture data that you want to load onto one database, but not onto another. In this situation, you can add a database identifier into the names of your fixtures.

For example, if your *`DATABASES`* setting has a `users` database defined, name the fixture `mydata.users.json` or `mydata.users.json.gz` and the fixture will only be loaded when you specify you want to load data into the `users` database.

3.2.13 Examples of model relationship API usage

Many-to-many relationships

To define a many-to-many relationship, use *`ManyToManyField`*.

In this example, an `Article` can be published in multiple `Publication` objects, and a `Publication` has multiple `Article` objects:

```
from django.db import models


class Publication(models.Model):
    title = models.CharField(max_length=30)

    class Meta:
        ordering = ["title"]

    def __str__(self):
        return self.title


class Article(models.Model):
    headline = models.CharField(max_length=100)
```

(continues on next page)

(continued from previous page)

```
    publications = models.ManyToManyField(Publication)

    class Meta:
        ordering = ["headline"]

    def __str__(self):
        return self.headline
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a few `Publication` instances:

```
>>> p1 = Publication(title="The Python Journal")
>>> p1.save()
>>> p2 = Publication(title="Science News")
>>> p2.save()
>>> p3 = Publication(title="Science Weekly")
>>> p3.save()
```

Create an `Article`:

```
>>> a1 = Article(headline="Django lets you build web apps easily")
```

You can't associate it with a `Publication` until it's been saved:

```
>>> a1.publications.add(p1)
Traceback (most recent call last):
...
ValueError: "<Article: Django lets you build web apps easily>" needs to have a value for␣
↪field "id" before this many-to-many relationship can be used.
```

Save it!

```
>>> a1.save()
```

Associate the `Article` with a `Publication`:

```
>>> a1.publications.add(p1)
```

Create another `Article`, and set it to appear in its publications:

```
>>> a2 = Article(headline="NASA uses Python")
>>> a2.save()
```

(continues on next page)

(continued from previous page)

```
>>> a2.publications.add(p1, p2)
>>> a2.publications.add(p3)
```

Adding a second time is OK, it will not duplicate the relation:

```
>>> a2.publications.add(p3)
```

Adding an object of the wrong type raises `TypeError`:

```
>>> a2.publications.add(a1)
Traceback (most recent call last):
...
TypeError: 'Publication' instance expected
```

Create and add a `Publication` to an `Article` in one step using *`create()`*:

```
>>> new_publication = a2.publications.create(title="Highlights for Children")
```

`Article` objects have access to their related `Publication` objects:

```
>>> a1.publications.all()
<QuerySet [<Publication: The Python Journal>]>
>>> a2.publications.all()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>,
↪<Publication: Science Weekly>, <Publication: The Python Journal>]>
```

`Publication` objects have access to their related `Article` objects:

```
>>> p2.article_set.all()
<QuerySet [<Article: NASA uses Python>]>
>>> p1.article_set.all()
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>
>>> Publication.objects.get(id=4).article_set.all()
<QuerySet [<Article: NASA uses Python>]>
```

Many-to-many relationships can be queried using lookups across relationships:

```
>>> Article.objects.filter(publications__id=1)
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>
>>> Article.objects.filter(publications__pk=1)
```

(continues on next page)

(continued from previous page)

```
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>
>>> Article.objects.filter(publications=1)
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>
>>> Article.objects.filter(publications=p1)
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>

>>> Article.objects.filter(publications__title__startswith="Science")
<QuerySet [<Article: NASA uses Python>, <Article: NASA uses Python>]>

>>> Article.objects.filter(publications__title__startswith="Science").distinct()
<QuerySet [<Article: NASA uses Python>]>
```

The *count()* function respects *distinct()* as well:

```
>>> Article.objects.filter(publications__title__startswith="Science").count()
2

>>> Article.objects.filter(publications__title__startswith="Science").distinct().count()
1

>>> Article.objects.filter(publications__in=[1, 2]).distinct()
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>
>>> Article.objects.filter(publications__in=[p1, p2]).distinct()
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA uses Python>
↪]>
```

Reverse m2m queries are supported (i.e., starting at the table that doesn't have a *ManyToManyField*):

```
>>> Publication.objects.filter(id=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(pk=1)
<QuerySet [<Publication: The Python Journal>]>

>>> Publication.objects.filter(article__headline__startswith="NASA")
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>,
↪<Publication: Science Weekly>, <Publication: The Python Journal>]>
```

(continues on next page)

(continued from previous page)

```
>>> Publication.objects.filter(article__id=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(article__pk=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(article=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(article=a1)
<QuerySet [<Publication: The Python Journal>]>

>>> Publication.objects.filter(article__in=[1, 2]).distinct()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>,
↪<Publication: Science Weekly>, <Publication: The Python Journal>]>
>>> Publication.objects.filter(article__in=[a1, a2]).distinct()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>,
↪<Publication: Science Weekly>, <Publication: The Python Journal>]>
```

Excluding a related item works as you would expect, too (although the SQL involved is a little complex):

```
>>> Article.objects.exclude(publications=p2)
<QuerySet [<Article: Django lets you build web apps easily>]>
```

If we delete a `Publication`, its related `Article` instances won't be able to access it:

```
>>> p1.delete()
>>> Publication.objects.all()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>,
↪<Publication: Science Weekly>]>
>>> a1 = Article.objects.get(pk=1)
>>> a1.publications.all()
<QuerySet []>
```

If we delete an `Article`, its related `Publication` instances won't be able to access it:

```
>>> a2.delete()
>>> Article.objects.all()
<QuerySet [<Article: Django lets you build web apps easily>]>
>>> p2.article_set.all()
<QuerySet []>
```

Adding via the 'other' end of an m2m:

```
>>> a4 = Article(headline="NASA finds intelligent life on Earth")
>>> a4.save()
>>> p2.article_set.add(a4)
>>> p2.article_set.all()
<QuerySet [<Article: NASA finds intelligent life on Earth>]>
>>> a4.publications.all()
<QuerySet [<Publication: Science News>]>
```

Adding via the other end using keywords:

```
>>> new_article = p2.article_set.create(headline="Oxygen-free diet works wonders")
>>> p2.article_set.all()
<QuerySet [<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet␣
↪works wonders>]>
>>> a5 = p2.article_set.all()[1]
>>> a5.publications.all()
<QuerySet [<Publication: Science News>]>
```

Removing `Publication` from an `Article`:

```
>>> a4.publications.remove(p2)
>>> p2.article_set.all()
<QuerySet [<Article: Oxygen-free diet works wonders>]>
>>> a4.publications.all()
<QuerySet []>
```

And from the other end:

```
>>> p2.article_set.remove(a5)
>>> p2.article_set.all()
<QuerySet []>
>>> a5.publications.all()
<QuerySet []>
```

Relation sets can be set:

```
>>> a4.publications.all()
<QuerySet [<Publication: Science News>]>
>>> a4.publications.set([p3])
>>> a4.publications.all()
<QuerySet [<Publication: Science Weekly>]>
```

Relation sets can be cleared:

```
>>> p2.article_set.clear()
>>> p2.article_set.all()
<QuerySet []>
```

And you can clear from the other end:

```
>>> p2.article_set.add(a4, a5)
>>> p2.article_set.all()
<QuerySet [<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet␣
↪works wonders>]>
>>> a4.publications.all()
<QuerySet [<Publication: Science News>, <Publication: Science Weekly>]>
>>> a4.publications.clear()
>>> a4.publications.all()
<QuerySet []>
>>> p2.article_set.all()
<QuerySet [<Article: Oxygen-free diet works wonders>]>
```

Recreate the `Article` and `Publication` we have deleted:

```
>>> p1 = Publication(title="The Python Journal")
>>> p1.save()
>>> a2 = Article(headline="NASA uses Python")
>>> a2.save()
>>> a2.publications.add(p1, p2, p3)
```

Bulk delete some `Publication` instances, and the references to deleted publications will no longer be included in the related entries:

```
>>> Publication.objects.filter(title__startswith="Science").delete()
>>> Publication.objects.all()
<QuerySet [<Publication: Highlights for Children>, <Publication: The Python Journal>]>
>>> Article.objects.all()
<QuerySet [<Article: Django lets you build web apps easily>, <Article: NASA finds␣
↪intelligent life on Earth>, <Article: NASA uses Python>, <Article: Oxygen-free diet␣
↪works wonders>]>
>>> a2.publications.all()
<QuerySet [<Publication: The Python Journal>]>
```

Bulk delete some articles - references to deleted objects should go:

```
>>> q = Article.objects.filter(headline__startswith="Django")
>>> print(q)
<QuerySet [<Article: Django lets you build web apps easily>]>
>>> q.delete()
```

After the *delete()*, the *QuerySet* cache needs to be cleared, and the referenced objects should be gone:

```
>>> print(q)
<QuerySet []>
>>> p1.article_set.all()
<QuerySet [<Article: NASA uses Python>]>
```

Many-to-one relationships

To define a many-to-one relationship, use *ForeignKey*.

In this example, a `Reporter` can be associated with many `Article` objects, but an `Article` can only have one `Reporter` object:

```
from django.db import models


class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()

    def __str__(self):
        return f"{self.first_name} {self.last_name}"


class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

    def __str__(self):
        return self.headline

    class Meta:
        ordering = ["headline"]
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a few Reporters:

```
>>> r = Reporter(first_name="John", last_name="Smith", email="john@example.com")
>>> r.save()

>>> r2 = Reporter(first_name="Paul", last_name="Jones", email="paul@example.com")
>>> r2.save()
```

Create an Article:

```
>>> from datetime import date
>>> a = Article(id=None, headline="This is a test", pub_date=date(2005, 7, 27),␣
↪reporter=r)
>>> a.save()

>>> a.reporter.id
1

>>> a.reporter
<Reporter: John Smith>
```

Note that you must save an object before it can be assigned to a foreign key relationship. For example, creating an `Article` with unsaved `Reporter` raises `ValueError`:

```
>>> r3 = Reporter(first_name="John", last_name="Smith", email="john@example.com")
>>> Article.objects.create(
...     headline="This is a test", pub_date=date(2005, 7, 27), reporter=r3
... )
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object
↪'reporter'.
```

Article objects have access to their related Reporter objects:

```
>>> r = a.reporter
```

Create an Article via the Reporter object:

```
>>> new_article = r.article_set.create(
...     headline="John's second story", pub_date=date(2005, 7, 29)
... )
>>> new_article
```

(continues on next page)

(continued from previous page)

```
<Article: John's second story>
>>> new_article.reporter
<Reporter: John Smith>
>>> new_article.reporter.id
1
```

Create a new article:

```
>>> new_article2 = Article.objects.create(
...     headline="Paul's story", pub_date=date(2006, 1, 17), reporter=r
... )
>>> new_article2.reporter
<Reporter: John Smith>
>>> new_article2.reporter.id
1
>>> r.article_set.all()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a␣
↪test>]>
```

Add the same article to a different article set - check that it moves:

```
>>> r2.article_set.add(new_article2)
>>> new_article2.reporter.id
2
>>> new_article2.reporter
<Reporter: Paul Jones>
```

Adding an object of the wrong type raises TypeError:

```
>>> r.article_set.add(r2)
Traceback (most recent call last):
...
TypeError: 'Article' instance expected, got <Reporter: Paul Jones>

>>> r.article_set.all()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> r2.article_set.all()
<QuerySet [<Article: Paul's story>]>

>>> r.article_set.count()
2
```

(continues on next page)

(continued from previous page)

```
>>> r2.article_set.count()
1
```

Note that in the last example the article has moved from John to Paul.

Related managers support field lookups as well. The API automatically follows relationships as far as you need. Use double underscores to separate relationships. This works as many levels deep as you want. There's no limit. For example:

```
>>> r.article_set.filter(headline__startswith="This")
<QuerySet [<Article: This is a test>]>

# Find all Articles for any Reporter whose first name is "John".
>>> Article.objects.filter(reporter__first_name="John")
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Exact match is implied here:

```
>>> Article.objects.filter(reporter__first_name="John")
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Query twice over the related field. This translates to an AND condition in the WHERE clause:

```
>>> Article.objects.filter(reporter__first_name="John", reporter__last_name="Smith")
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

For the related lookup you can supply a primary key value or pass the related object explicitly:

```
>>> Article.objects.filter(reporter__pk=1)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter=1)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter=r)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>

>>> Article.objects.filter(reporter__in=[1, 2]).distinct()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a␣
↪test>]>
>>> Article.objects.filter(reporter__in=[r, r2]).distinct()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a␣
↪test>]>
```

You can also use a queryset instead of a literal list of instances:

```
>>> Article.objects.filter(
...     reporter__in=Reporter.objects.filter(first_name="John")
... ).distinct()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Querying in the opposite direction:

```
>>> Reporter.objects.filter(article__pk=1)
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article=1)
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article=a)
<QuerySet [<Reporter: John Smith>]>

>>> Reporter.objects.filter(article__headline__startswith="This")
<QuerySet [<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]>
>>> Reporter.objects.filter(article__headline__startswith="This").distinct()
<QuerySet [<Reporter: John Smith>]>
```

Counting in the opposite direction works in conjunction with `distinct()`:

```
>>> Reporter.objects.filter(article__headline__startswith="This").count()
3
>>> Reporter.objects.filter(article__headline__startswith="This").distinct().count()
1
```

Queries can go round in circles:

```
>>> Reporter.objects.filter(article__reporter__first_name__startswith="John")
<QuerySet [<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>,
↪<Reporter: John Smith>]>
>>> Reporter.objects.filter(article__reporter__first_name__startswith="John").distinct()
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article__reporter=r).distinct()
<QuerySet [<Reporter: John Smith>]>
```

If you delete a reporter, their articles will be deleted (assuming that the ForeignKey was defined with *django.db.models.ForeignKey.on_delete* set to `CASCADE`, which is the default):

```
>>> Article.objects.all()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a␣
```

(continues on next page)

(continued from previous page)

```
↪test>]>
>>> Reporter.objects.order_by("first_name")
<QuerySet [<Reporter: John Smith>, <Reporter: Paul Jones>]>
>>> r2.delete()
>>> Article.objects.all()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Reporter.objects.order_by("first_name")
<QuerySet [<Reporter: John Smith>]>
```

You can delete using a JOIN in the query:

```
>>> Reporter.objects.filter(article__headline__startswith="This").delete()
>>> Reporter.objects.all()
<QuerySet []>
>>> Article.objects.all()
<QuerySet []>
```

One-to-one relationships

To define a one-to-one relationship, use *OneToOneField*.

In this example, a `Place` optionally can be a `Restaurant`:

```
from django.db import models


class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

    def __str__(self):
        return f"{self.name} the place"


class Restaurant(models.Model):
    place = models.OneToOneField(
        Place,
        on_delete=models.CASCADE,
        primary_key=True,
    )
    serves_hot_dogs = models.BooleanField(default=False)
```

(continues on next page)

(continued from previous page)

```
    serves_pizza = models.BooleanField(default=False)

    def __str__(self):
        return "%s the restaurant" % self.place.name


class Waiter(models.Model):
    restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE)
    name = models.CharField(max_length=50)

    def __str__(self):
        return "%s the waiter at %s" % (self.name, self.restaurant)
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a couple of Places:

```
>>> p1 = Place(name="Demon Dogs", address="944 W. Fullerton")
>>> p1.save()
>>> p2 = Place(name="Ace Hardware", address="1013 N. Ashland")
>>> p2.save()
```

Create a Restaurant. Pass the "parent" object as this object's primary key:

```
>>> r = Restaurant(place=p1, serves_hot_dogs=True, serves_pizza=False)
>>> r.save()
```

A Restaurant can access its place:

```
>>> r.place
<Place: Demon Dogs the place>
```

A Place can access its restaurant, if available:

```
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

p2 doesn't have an associated restaurant:

```
>>> from django.core.exceptions import ObjectDoesNotExist
>>> try:
...     p2.restaurant
```

(continues on next page)

(continued from previous page)

```
... except ObjectDoesNotExist:
...     print("There is no restaurant here.")
...
There is no restaurant here.
```

You can also use `hasattr` to avoid the need for exception catching:

```
>>> hasattr(p2, "restaurant")
False
```

Set the place using assignment notation. Because place is the primary key on Restaurant, the save will create a new restaurant:

```
>>> r.place = p2
>>> r.save()
>>> p2.restaurant
<Restaurant: Ace Hardware the restaurant>
>>> r.place
<Place: Ace Hardware the place>
```

Set the place back again, using assignment in the reverse direction:

```
>>> p1.restaurant = r
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

Note that you must save an object before it can be assigned to a one-to-one relationship. For example, creating a `Restaurant` with unsaved `Place` raises `ValueError`:

```
>>> p3 = Place(name="Demon Dogs", address="944 W. Fullerton")
>>> Restaurant.objects.create(place=p3, serves_hot_dogs=True, serves_pizza=False)
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object 'place'.
```

Restaurant.objects.all() returns the Restaurants, not the Places. Note that there are two restaurants - Ace Hardware the Restaurant was created in the call to r.place = p2:

```
>>> Restaurant.objects.all()
<QuerySet [<Restaurant: Demon Dogs the restaurant>, <Restaurant: Ace Hardware the␣
↪restaurant>]>
```

Place.objects.all() returns all Places, regardless of whether they have Restaurants:

```
>>> Place.objects.order_by("name")
<QuerySet [<Place: Ace Hardware the place>, <Place: Demon Dogs the place>]>
```

You can query the models using lookups across relationships:

```
>>> Restaurant.objects.get(place=p1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.get(place__pk=1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.filter(place__name__startswith="Demon")
<QuerySet [<Restaurant: Demon Dogs the restaurant>]>
>>> Restaurant.objects.exclude(place__address__contains="Ashland")
<QuerySet [<Restaurant: Demon Dogs the restaurant>]>
```

This also works in reverse:

```
>>> Place.objects.get(pk=1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place=p1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant=r)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place__name__startswith="Demon")
<Place: Demon Dogs the place>
```

If you delete a place, its restaurant will be deleted (assuming that the `OneToOneField` was defined with *`on_delete`* set to `CASCADE`, which is the default):

```
>>> p2.delete()
(2, {'one_to_one.Restaurant': 1, 'one_to_one.Place': 1})
>>> Restaurant.objects.all()
<QuerySet [<Restaurant: Demon Dogs the restaurant>]>
```

Add a Waiter to the Restaurant:

```
>>> w = r.waiter_set.create(name="Joe")
>>> w
<Waiter: Joe the waiter at Demon Dogs the restaurant>
```

Query the waiters:

```
>>> Waiter.objects.filter(restaurant__place=p1)
```

(continues on next page)

(continued from previous page)

```
<QuerySet [<Waiter: Joe the waiter at Demon Dogs the restaurant>]>
>>> Waiter.objects.filter(restaurant__place__name__startswith="Demon")
<QuerySet [<Waiter: Joe the waiter at Demon Dogs the restaurant>]>
```

3.3 Handling HTTP requests

Information on handling HTTP requests in Django:

3.3.1 URL dispatcher

A clean, elegant URL scheme is an important detail in a high-quality web application. Django lets you design URLs however you want, with no framework limitations.

See Cool URIs don't change, by World Wide Web creator Tim Berners-Lee, for excellent arguments on why URLs should be clean and usable.

Overview

To design URLs for an app, you create a Python module informally called a URLconf (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it's pure Python code, it can be constructed dynamically.

Django also provides a way to translate URLs according to the active language. See the internationalization documentation for more information.

How Django processes a request

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django determines the root URLconf module to use. Ordinarily, this is the value of the *ROOT_URLCONF* setting, but if the incoming `HttpRequest` object has a *urlconf* attribute (set by middleware), its value will be used in place of the *ROOT_URLCONF* setting.
2. Django loads that Python module and looks for the variable `urlpatterns`. This should be a sequence of *django.urls.path()* and/or *django.urls.re_path()* instances.
3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL, matching against *path_info*.
4. Once one of the URL patterns matches, Django imports and calls the given view, which is a Python function (or a class-based view). The view gets passed the following arguments:
    - An instance of *HttpRequest*.

- If the matched URL pattern contained no named groups, then the matches from the regular expression are provided as positional arguments.
- The keyword arguments are made up of any named parts matched by the path expression that are provided, overridden by any arguments specified in the optional `kwargs` argument to *`django.urls.path()`* or *`django.urls.re_path()`*.

5. If no URL pattern matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view. See Error handling below.

Example

Here's a sample URLconf:

```
from django.urls import path

from . import views

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    path("articles/<int:year>/", views.year_archive),
    path("articles/<int:year>/<int:month>/", views.month_archive),
    path("articles/<int:year>/<int:month>/<slug:slug>/", views.article_detail),
]
```

Notes:

- To capture a value from the URL, use angle brackets.
- Captured values can optionally include a converter type. For example, use `<int:name>` to capture an integer parameter. If a converter isn't included, any string, excluding a `/` character, is matched.
- There's no need to add a leading slash, because every URL has that. For example, it's `articles`, not `/articles`.

Example requests:

- A request to `/articles/2005/03/` would match the third entry in the list. Django would call the function `views.month_archive(request, year=2005, month=3)`.
- `/articles/2003/` would match the first pattern in the list, not the second one, because the patterns are tested in order, and the first one is the first test to pass. Feel free to exploit the ordering to insert special cases like this. Here, Django would call the function `views.special_case_2003(request)`
- `/articles/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/articles/2003/03/building-a-django-site/` would match the final pattern. Django would call the function `views.article_detail(request, year=2003, month=3,`

slug="building-a-django-site").

Path converters

The following path converters are available by default:

- `str` - Matches any non-empty string, excluding the path separator, `'/'`. This is the default if a converter isn't included in the expression.
- `int` - Matches zero or any positive integer. Returns an `int`.
- `slug` - Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters. For example, `building-your-1st-django-site`.
- `uuid` - Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase. For example, `075194d3-6885-417e-a8a8-6c931e272f00`. Returns a `UUID` instance.
- `path` - Matches any non-empty string, including the path separator, `'/'`. This allows you to match against a complete URL path rather than a segment of a URL path as with `str`.

Registering custom path converters

For more complex matching requirements, you can define your own path converters.

A converter is a class that includes the following:

- A `regex` class attribute, as a string.
- A `to_python(self, value)` method, which handles converting the matched string into the type that should be passed to the view function. It should raise `ValueError` if it can't convert the given value. A `ValueError` is interpreted as no match and as a consequence a 404 response is sent to the user unless another URL pattern matches.
- A `to_url(self, value)` method, which handles converting the Python type into a string to be used in the URL. It should raise `ValueError` if it can't convert the given value. A `ValueError` is interpreted as no match and as a consequence *`reverse()`* will raise *`NoReverseMatch`* unless another URL pattern matches.

For example:

```
class FourDigitYearConverter:
    regex = "[0-9]{4}"

    def to_python(self, value):
        return int(value)

    def to_url(self, value):
        return "%04d" % value
```

Register custom converter classes in your URLconf using *register_converter()*:

```
from django.urls import path, register_converter

from . import converters, views

register_converter(converters.FourDigitYearConverter, "yyyy")

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    path("articles/<yyyy:year>/", views.year_archive),
    ...,
]
```

Deprecated since version 5.1: Overriding existing converters with `django.urls.register_converter()` is deprecated.

Using regular expressions

If the paths and converters syntax isn't sufficient for defining your URL patterns, you can also use regular expressions. To do so, use *re_path()* instead of *path()*.

In Python regular expressions, the syntax for named regular expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match.

Here's the example URLconf from earlier, rewritten using regular expressions:

```
from django.urls import path, re_path

from . import views

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    re_path(r"^articles/(?P<year>[0-9]{4})/$", views.year_archive),
    re_path(r"^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$", views.month_archive),
    re_path(
        r"^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$",
        views.article_detail,
    ),
]
```

This accomplishes roughly the same thing as the previous example, except:

- The exact URLs that will match are slightly more constrained. For example, the year 10000 will no longer match since the year integers are constrained to be exactly four digits long.

- Each captured argument is sent to the view as a string, regardless of what sort of match the regular expression makes.

When switching from using *path()* to *re_path()* or vice versa, it's particularly important to be aware that the type of the view arguments may change, and so you may need to adapt your views.

Using unnamed regular expression groups

As well as the named group syntax, e.g. `(?P<year>[0-9]{4})`, you can also use the shorter unnamed group, e.g. `([0-9]{4})`.

This usage isn't particularly recommended as it makes it easier to accidentally introduce errors between the intended meaning of a match and the arguments of the view.

In either case, using only one style within a given regex is recommended. When both styles are mixed, any unnamed groups are ignored and only named groups are passed to the view function.

Nested arguments

Regular expressions allow nested arguments, and Django will resolve them and pass them to the view. When reversing, Django will try to fill in all outer captured arguments, ignoring any nested captured arguments. Consider the following URL patterns which optionally take a page argument:

```
from django.urls import re_path

urlpatterns = [
    re_path(r"^blog/(page-([0-9]+)/)?$", blog_articles),  # bad
    re_path(r"^comments/(?:page-(?P<page_number>[0-9]+)/)?$", comments),  # good
]
```

Both patterns use nested arguments and will resolve: for example, `blog/page-2/` will result in a match to `blog_articles` with two positional arguments: `page-2/` and `2`. The second pattern for `comments` will match `comments/page-2/` with keyword argument `page_number` set to 2. The outer argument in this case is a non-capturing argument `(?:...)`.

The `blog_articles` view needs the outermost captured argument to be reversed, `page-2/` or no arguments in this case, while `comments` can be reversed with either no arguments or a value for `page_number`.

Nested captured arguments create a strong coupling between the view arguments and the URL as illustrated by `blog_articles`: the view receives part of the URL (`page-2/`) instead of only the value the view is interested in. This coupling is even more pronounced when reversing, since to reverse the view we need to pass the piece of URL instead of the page number.

As a rule of thumb, only capture the values the view needs to work with and use non-capturing arguments when the regular expression needs an argument but the view ignores it.

What the URLconf searches against

The URLconf searches against the requested URL, as a normal Python string. This does not include GET or POST parameters, or the domain name.

For example, in a request to `https://www.example.com/myapp/`, the URLconf will look for `myapp/`.

In a request to `https://www.example.com/myapp/?page=3`, the URLconf will look for `myapp/`.

The URLconf doesn't look at the request method. In other words, all request methods – `POST`, `GET`, `HEAD`, etc. – will be routed to the same function for the same URL.

Specifying defaults for view arguments

A convenient trick is to specify default parameters for your views' arguments. Here's an example URLconf and view:

```
# URLconf
from django.urls import path

from . import views

urlpatterns = [
    path("blog/", views.page),
    path("blog/page<int:num>/", views.page),
]


# View (in blog/views.py)
def page(request, num=1):
    # Output the appropriate page of blog entries, according to num.
    ...
```

In the above example, both URL patterns point to the same view – `views.page` – but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `1`. If the second pattern matches, `page()` will use whatever `num` value was captured.

Performance

Django processes regular expressions in the `urlpatterns` list which is compiled the first time it's accessed. Subsequent requests use the cached configuration via the URL resolver.

Syntax of the `urlpatterns` variable

`urlpatterns` should be a sequence of *`path()`* and/or *`re_path()`* instances.

Error handling

When Django can't find a match for the requested URL, or when an exception is raised, Django invokes an error-handling view.

The views to use for these cases are specified by four variables. Their default values should suffice for most projects, but further customization is possible by overriding their default values.

See the documentation on customizing error views for the full details.

Such values can be set in your root URLconf. Setting these variables in any other URLconf will have no effect.

Values must be callables, or strings representing the full Python import path to the view that should be called to handle the error condition at hand.

The variables are:

- `handler400` – See *`django.conf.urls.handler400`*.
- `handler403` – See *`django.conf.urls.handler403`*.
- `handler404` – See *`django.conf.urls.handler404`*.
- `handler500` – See *`django.conf.urls.handler500`*.

Including other URLconfs

At any point, your `urlpatterns` can "include" other URLconf modules. This essentially "roots" a set of URLs below other ones.

For example, here's an excerpt of the URLconf for the Django website itself. It includes a number of other URLconfs:

```
from django.urls import include, path

urlpatterns = [
    # ... snip ...
    path("community/", include("aggregator.urls")),
    path("contact/", include("contact.urls")),
    # ... snip ...
]
```

Whenever Django encounters *`include()`*, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Another possibility is to include additional URL patterns by using a list of *path()* instances. For example, consider this URLconf:

```
from django.urls import include, path

from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    path("reports/", credit_views.report),
    path("reports/<int:id>/", credit_views.report),
    path("charge/", credit_views.charge),
]

urlpatterns = [
    path("", main_views.homepage),
    path("help/", include("apps.help.urls")),
    path("credit/", include(extra_patterns)),
]
```

In this example, the `/credit/reports/` URL will be handled by the `credit_views.report()` Django view.

This can be used to remove redundancy from URLconfs where a single pattern prefix is used repeatedly. For example, consider this URLconf:

```
from django.urls import path
from . import views

urlpatterns = [
    path("<page_slug>-<page_id>/history/", views.history),
    path("<page_slug>-<page_id>/edit/", views.edit),
    path("<page_slug>-<page_id>/discuss/", views.discuss),
    path("<page_slug>-<page_id>/permissions/", views.permissions),
]
```

We can improve this by stating the common path prefix only once and grouping the suffixes that differ:

```
from django.urls import include, path
from . import views

urlpatterns = [
    path(
        "<page_slug>-<page_id>/",
```

(continues on next page)

(continued from previous page)

```
        include(
            [
                path("history/", views.history),
                path("edit/", views.edit),
                path("discuss/", views.discuss),
                path("permissions/", views.permissions),
            ]
        ),
    ),
]
```

Captured parameters

An included URLconf receives any captured parameters from parent URLconfs, so the following example is valid:

```
# In settings/urls/main.py
from django.urls import include, path

urlpatterns = [
    path("<username>/blog/", include("foo.urls.blog")),
]

# In foo/urls/blog.py
from django.urls import path
from . import views

urlpatterns = [
    path("", views.blog.index),
    path("archive/", views.blog.archive),
]
```

In the above example, the captured `"username"` variable is passed to the included URLconf, as expected.

Passing extra options to view functions

URLconfs have a hook that lets you pass extra arguments to your view functions, as a Python dictionary.

The *`path()`* function can take an optional third argument which should be a dictionary of extra keyword arguments to pass to the view function.

For example:

```
from django.urls import path
from . import views

urlpatterns = [
    path("blog/<int:year>/", views.year_archive, {"foo": "bar"}),
]
```

In this example, for a request to `/blog/2005/`, Django will call `views.year_archive(request, year=2005, foo='bar')`.

This technique is used in the syndication framework to pass metadata and options to views.

> **Dealing with conflicts**
>
> It's possible to have a URL pattern which captures named keyword arguments, and also passes arguments with the same names in its dictionary of extra arguments. When this happens, the arguments in the dictionary will be used instead of the arguments captured in the URL.

Passing extra options to `include()`

Similarly, you can pass extra options to *`include()`* and each line in the included URLconf will be passed the extra options.

For example, these two URLconf sets are functionally identical:

Set one:

```
# main.py
from django.urls import include, path

urlpatterns = [
    path("blog/", include("inner"), {"blog_id": 3}),
]

# inner.py
from django.urls import path
from mysite import views

urlpatterns = [
    path("archive/", views.archive),
    path("about/", views.about),
]
```

Set two:

```
# main.py
from django.urls import include, path
from mysite import views

urlpatterns = [
    path("blog/", include("inner")),
]

# inner.py
from django.urls import path

urlpatterns = [
    path("archive/", views.archive, {"blog_id": 3}),
    path("about/", views.about, {"blog_id": 3}),
]
```

Note that extra options will always be passed to every line in the included URLconf, regardless of whether the line's view actually accepts those options as valid. For this reason, this technique is only useful if you're certain that every view in the included URLconf accepts the extra options you're passing.

Reverse resolution of URLs

A common need when working on a Django project is the possibility to obtain URLs in their final forms either for embedding in generated content (views and assets URLs, URLs shown to the user, etc.) or for handling of the navigation flow on the server side (redirections, etc.)

It is strongly desirable to avoid hard-coding these URLs (a laborious, non-scalable and error-prone strategy). Equally dangerous is devising ad-hoc mechanisms to generate URLs that are parallel to the design described by the URLconf, which can result in the production of URLs that become stale over time.

In other words, what's needed is a DRY mechanism. Among other advantages it would allow evolution of the URL design without having to go over all the project source code to search and replace outdated URLs.

The primary piece of information we have available to get a URL is an identification (e.g. the name) of the view in charge of handling it. Other pieces of information that necessarily must participate in the lookup of the right URL are the types (positional, keyword) and values of the view arguments.

Django provides a solution such that the URL mapper is the only repository of the URL design. You feed it with your URLconf and then it can be used in both directions:

- Starting with a URL requested by the user/browser, it calls the right Django view providing any arguments it might need with their values as extracted from the URL.
- Starting with the identification of the corresponding Django view plus the values of arguments that would be passed to it, obtain the associated URL.

The first one is the usage we've been discussing in the previous sections. The second one is what is known as reverse resolution of URLs, reverse URL matching, reverse URL lookup, or simply URL reversing.

Django provides tools for performing URL reversing that match the different layers where URLs are needed:

- In templates: Using the *url* template tag.
- In Python code: Using the *reverse()* function.
- In higher level code related to handling of URLs of Django model instances: The *get_absolute_url()* method.

Examples

Consider again this URLconf entry:

```
from django.urls import path

from . import views

urlpatterns = [
    # ...
    path("articles/<int:year>/", views.year_archive, name="news-year-archive"),
    # ...
]
```

According to this design, the URL for the archive corresponding to year nnnn is `/articles/<nnnn>/`.

You can obtain these in template code by using:

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>
{# Or with the year in a template context variable: #}
<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }} Archive</a></li>
{% endfor %}
</ul>
```

Or in Python code:

```
from django.http import HttpResponseRedirect
from django.urls import reverse


def redirect_to_year(request):
    # ...
```

(continues on next page)

(continued from previous page)

```
    year = 2006
    # ...
    return HttpResponseRedirect(reverse("news-year-archive", args=(year,)))
```

If, for some reason, it was decided that the URLs where content for yearly article archives are published at should be changed then you would only need to change the entry in the URLconf.

In some scenarios where views are of a generic nature, a many-to-one relationship might exist between URLs and views. For these cases the view name isn't a good enough identifier for it when comes the time of reversing URLs. Read the next section to know about the solution Django provides for this.

Naming URL patterns

In order to perform URL reversing, you'll need to use named URL patterns as done in the examples above. The string used for the URL name can contain any characters you like. You are not restricted to valid Python names.

When naming URL patterns, choose names that are unlikely to clash with other applications' choice of names. If you call your URL pattern `comment` and another application does the same thing, the URL that *`reverse()`* finds depends on whichever pattern is last in your project's `urlpatterns` list.

Putting a prefix on your URL names, perhaps derived from the application name (such as `myapp-comment` instead of `comment`), decreases the chance of collision.

You can deliberately choose the same URL name as another application if you want to override a view. For example, a common use case is to override the *`LoginView`*. Parts of Django and most third-party apps assume that this view has a URL pattern with the name `login`. If you have a custom login view and give its URL the name `login`, *`reverse()`* will find your custom view as long as it's in `urlpatterns` after `django.contrib.auth.urls` is included (if that's included at all).

You may also use the same name for multiple URL patterns if they differ in their arguments. In addition to the URL name, *`reverse()`* matches the number of arguments and the names of the keyword arguments. Path converters can also raise `ValueError` to indicate no match, see Registering custom path converters for details.

URL namespaces

Introduction

URL namespaces allow you to uniquely reverse named URL patterns even if different applications use the same URL names. It's a good practice for third-party apps to always use namespaced URLs (as we did in the tutorial). Similarly, it also allows you to reverse URLs if multiple instances of an application are deployed. In other words, since multiple instances of a single application will share named URLs, namespaces provide a way to tell these named URLs apart.

Django applications that make proper use of URL namespacing can be deployed more than once for a particular site. For example *django.contrib.admin* has an *AdminSite* class which allows you to deploy more than one instance of the admin. In a later example, we'll discuss the idea of deploying the polls application from the tutorial in two different locations so we can serve the same functionality to two different audiences (authors and publishers).

A URL namespace comes in two parts, both of which are strings:

application namespace
: This describes the name of the application that is being deployed. Every instance of a single application will have the same application namespace. For example, Django's admin application has the somewhat predictable application namespace of `'admin'`.

instance namespace
: This identifies a specific instance of an application. Instance namespaces should be unique across your entire project. However, an instance namespace can be the same as the application namespace. This is used to specify a default instance of an application. For example, the default Django admin instance has an instance namespace of `'admin'`.

Namespaced URLs are specified using the `':'` operator. For example, the main index page of the admin application is referenced using `'admin:index'`. This indicates a namespace of `'admin'`, and a named URL of `'index'`.

Namespaces can also be nested. The named URL `'sports:polls:index'` would look for a pattern named `'index'` in the namespace `'polls'` that is itself defined within the top-level namespace `'sports'`.

Reversing namespaced URLs

When given a namespaced URL (e.g. `'polls:index'`) to resolve, Django splits the fully qualified name into parts and then tries the following lookup:

1. First, Django looks for a matching application namespace (in this example, `'polls'`). This will yield a list of instances of that application.

2. If there is a current application defined, Django finds and returns the URL resolver for that instance. The current application can be specified with the `current_app` argument to the *reverse()* function.

   The *url* template tag uses the namespace of the currently resolved view as the current application in a *RequestContext*. You can override this default by setting the current application on the *request.current_app* attribute.

3. If there is no current application, Django looks for a default application instance. The default application instance is the instance that has an instance namespace matching the application namespace (in this example, an instance of `polls` called `'polls'`).

4. If there is no default application instance, Django will pick the last deployed instance of the application, whatever its instance name may be.

5. If the provided namespace doesn't match an application namespace in step 1, Django will attempt a direct lookup of the namespace as an instance namespace.

If there are nested namespaces, these steps are repeated for each part of the namespace until only the view name is unresolved. The view name will then be resolved into a URL in the namespace that has been found.

Example

To show this resolution strategy in action, consider an example of two instances of the `polls` application from the tutorial: one called `'author-polls'` and one called `'publisher-polls'`. Assume we have enhanced that application so that it takes the instance namespace into consideration when creating and displaying polls.

Listing 2: `urls.py`

```
from django.urls import include, path

urlpatterns = [
    path("author-polls/", include("polls.urls", namespace="author-polls")),
    path("publisher-polls/", include("polls.urls", namespace="publisher-polls")),
]
```

Listing 3: `polls/urls.py`

```
from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.IndexView.as_view(), name="index"),
    path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    ...,
]
```

Using this setup, the following lookups are possible:

- If one of the instances is current - say, if we were rendering the detail page in the instance `'author-polls'` - `'polls:index'` will resolve to the index page of the `'author-polls'` instance; i.e. both of the following will result in `"/author-polls/"`.

  In the method of a class-based view:

  ```
  reverse("polls:index", current_app=self.request.resolver_match.namespace)
  ```

  and in the template:

```
{% url 'polls:index' %}
```

- If there is no current instance - say, if we were rendering a page somewhere else on the site - `'polls:index'` will resolve to the last registered instance of `polls`. Since there is no default instance (instance namespace of `'polls'`), the last instance of `polls` that is registered will be used. This would be `'publisher-polls'` since it's declared last in the `urlpatterns`.
- `'author-polls:index'` will always resolve to the index page of the instance `'author-polls'` (and likewise for `'publisher-polls'`).

If there were also a default instance - i.e., an instance named `'polls'` - the only change from above would be in the case where there is no current instance (the second item in the list above). In this case `'polls:index'` would resolve to the index page of the default instance instead of the instance declared last in `urlpatterns`.

URL namespaces and included URLconfs

Application namespaces of included URLconfs can be specified in two ways.

Firstly, you can set an `app_name` attribute in the included URLconf module, at the same level as the `urlpatterns` attribute. You have to pass the actual module, or a string reference to the module, to *include()*, not the list of `urlpatterns` itself.

Listing 4: `polls/urls.py`

```
from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.IndexView.as_view(), name="index"),
    path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    ...,
]
```

Listing 5: `urls.py`

```
from django.urls import include, path

urlpatterns = [
    path("polls/", include("polls.urls")),
]
```

The URLs defined in `polls.urls` will have an application namespace `polls`.

Secondly, you can include an object that contains embedded namespace data. If you `include()` a list of *`path()`* or *`re_path()`* instances, the URLs contained in that object will be added to the global namespace. However, you can also `include()` a 2-tuple containing:

```
(<list of path()/re_path() instances>, <application namespace>)
```

For example:

```
from django.urls import include, path

from . import views

polls_patterns = (
    [
        path("", views.IndexView.as_view(), name="index"),
        path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    ],
    "polls",
)

urlpatterns = [
    path("polls/", include(polls_patterns)),
]
```

This will include the nominated URL patterns into the given application namespace.

The instance namespace can be specified using the `namespace` argument to *`include()`*. If the instance namespace is not specified, it will default to the included URLconf's application namespace. This means it will also be the default instance for that namespace.

3.3.2 Writing views

A view function, or view for short, is a Python function that takes a web request and returns a web response. This response can be the HTML contents of a web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path. There's no other requirement–no "magic," so to speak. For the sake of putting the code somewhere, the convention is to put views in a file called `views.py`, placed in your project or application directory.

A simple view

Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime


def current_datetime(request):
    now = datetime.datetime.now()
    html = '<html lang="en"><body>It is now %s.</body></html>' % now
    return HttpResponse(html)
```

Let's step through this code one line at a time:

- First, we import the class *HttpResponse* from the *django.http* module, along with Python's `datetime` library.
- Next, we define a function called `current_datetime`. This is the view function. Each view function takes an *HttpRequest* object as its first parameter, which is typically named `request`.

  Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it `current_datetime` here, because that name clearly indicates what it does.
- The view returns an *HttpResponse* object that contains the generated response. Each view function is responsible for returning an *HttpResponse* object. (There are exceptions, but we'll get to those later.)

> **Django's Time Zone**
>
> Django includes a *TIME_ZONE* setting that defaults to `America/Chicago`. This probably isn't where you live, so you might want to change it in your settings file.

Mapping URLs to views

So, to recap, this view function returns an HTML page that includes the current date and time. To display this view at a particular URL, you'll need to create a URLconf; see URL dispatcher for instructions.

Returning errors

Django provides help for returning HTTP error codes. There are subclasses of *HttpResponse* for a number of common HTTP status codes other than 200 (which means "OK"). You can find the full list of available subclasses in the request/response documentation. Return an instance of one of those subclasses instead of a normal *HttpResponse* in order to signify an error. For example:

```
from django.http import HttpResponse, HttpResponseNotFound


def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound("<h1>Page not found</h1>")
    else:
        return HttpResponse("<h1>Page was found</h1>")
```

There isn't a specialized subclass for every possible HTTP response code, since many of them aren't going to be that common. However, as documented in the *HttpResponse* documentation, you can also pass the HTTP status code into the constructor for *HttpResponse* to create a return class for any status code you like. For example:

```
from django.http import HttpResponse


def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

Because 404 errors are by far the most common HTTP error, there's an easier way to handle those errors.

The `Http404` exception

class django.http.**Http404**

When you return an error such as *HttpResponseNotFound*, you're responsible for defining the HTML of the resulting error page:

```
return HttpResponseNotFound("<h1>Page not found</h1>")
```

For convenience, and because it's a good idea to have a consistent 404 error page across your site, Django provides an `Http404` exception. If you raise `Http404` at any point in a view function, Django will catch it and return the standard error page for your application, along with an HTTP error code 404.

Example usage:

```
from django.http import Http404
from django.shortcuts import render
```

(continues on next page)

(continued from previous page)

```
from polls.models import Poll


def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, "polls/detail.html", {"poll": p})
```

In order to show customized HTML when Django returns a 404, you can create an HTML template named `404.html` and place it in the top level of your template tree. This template will then be served when *DEBUG* is set to `False`.

When *DEBUG* is `True`, you can provide a message to `Http404` and it will appear in the standard 404 debug template. Use these messages for debugging purposes; they generally aren't suitable for use in a production 404 template.

Customizing error views

The default error views in Django should suffice for most web applications, but can easily be overridden if you need any custom behavior. Specify the handlers as seen below in your URLconf (setting them anywhere else will have no effect).

The *page_not_found()* view is overridden by *handler404*:

```
handler404 = "mysite.views.my_custom_page_not_found_view"
```

The *server_error()* view is overridden by *handler500*:

```
handler500 = "mysite.views.my_custom_error_view"
```

The *permission_denied()* view is overridden by *handler403*:

```
handler403 = "mysite.views.my_custom_permission_denied_view"
```

The *bad_request()* view is overridden by *handler400*:

```
handler400 = "mysite.views.my_custom_bad_request_view"
```

See also

Use the *CSRF_FAILURE_VIEW* setting to override the CSRF error view.

Testing custom error views

To test the response of a custom error handler, raise the appropriate exception in a test view. For example:

```
from django.core.exceptions import PermissionDenied
from django.http import HttpResponse
from django.test import SimpleTestCase, override_settings
from django.urls import path


def response_error_handler(request, exception=None):
    return HttpResponse("Error handler content", status=403)


def permission_denied_view(request):
    raise PermissionDenied


urlpatterns = [
    path("403/", permission_denied_view),
]

handler403 = response_error_handler


# ROOT_URLCONF must specify the module that contains handler403 = ...
@override_settings(ROOT_URLCONF=__name__)
class CustomErrorHandlerTests(SimpleTestCase):
    def test_handler_renders_template_response(self):
        response = self.client.get("/403/")
        # Make assertions on the response here. For example:
        self.assertContains(response, "Error handler content", status_code=403)
```

Async views

As well as being synchronous functions, views can also be asynchronous ("async") functions, normally defined using Python's `async def` syntax. Django will automatically detect these and run them in an async context. However, you will need to use an async server based on ASGI to get their performance benefits.

Here's an example of an async view:

```
import datetime
from django.http import HttpResponse
```

(continues on next page)

(continued from previous page)

```
async def current_datetime(request):
    now = datetime.datetime.now()
    html = '<html lang="en"><body>It is now %s.</body></html>' % now
    return HttpResponse(html)
```

You can read more about Django's async support, and how to best use async views, in Asynchronous support.

3.3.3 View decorators

Django provides several decorators that can be applied to views to support various HTTP features.

See Decorating the class for how to use these decorators with class-based views.

Allowed HTTP methods

The decorators in *django.views.decorators.http* can be used to restrict access to views based on the request method. These decorators will return a *django.http.HttpResponseNotAllowed* if the conditions are not met.

require_http_methods(request_method_list)

Decorator to require that a view only accepts particular request methods. Usage:

```
from django.views.decorators.http import require_http_methods


@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it this far
    # ...
    pass
```

Note that request methods should be in uppercase.

require_GET()

Decorator to require that a view only accepts the GET method.

require_POST()

Decorator to require that a view only accepts the POST method.

require_safe()

Decorator to require that a view only accepts the GET and HEAD methods. These methods are commonly considered "safe" because they should not have the significance of taking an action other than retrieving the requested resource.

> **Note**
>
> Web servers should automatically strip the content of responses to HEAD requests while leaving the headers unchanged, so you may handle HEAD requests exactly like GET requests in your views. Since some software, such as link checkers, rely on HEAD requests, you might prefer using `require_safe` instead of `require_GET`.

Conditional view processing

The following decorators in *django.views.decorators.http* can be used to control caching behavior on particular views.

condition(etag_func=None, last_modified_func=None)

conditional_page()

This decorator provides the conditional GET operation handling of *ConditionalGetMiddleware* to a view.

etag(etag_func)

last_modified(last_modified_func)

These decorators can be used to generate `ETag` and `Last-Modified` headers; see conditional view processing.

GZip compression

The decorators in *django.views.decorators.gzip* control content compression on a per-view basis.

gzip_page()

This decorator compresses content if the browser allows gzip compression. It sets the `Vary` header accordingly, so that caches will base their storage on the `Accept-Encoding` header.

Vary headers

The decorators in *django.views.decorators.vary* can be used to control caching based on specific request headers.

vary_on_cookie(func)

vary_on_headers(*headers)

The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key.

See using vary headers.

Caching

The decorators in *django.views.decorators.cache* control server and client-side caching.

`cache_control`(**kwargs)

This decorator patches the response's `Cache-Control` header by adding all of the keyword arguments to it. See *patch_cache_control()* for the details of the transformation.

`never_cache`(view_func)

This decorator adds an `Expires` header to the current date/time.

This decorator adds a `Cache-Control: max-age=0, no-cache, no-store, must-revalidate, private` header to a response to indicate that a page should never be cached.

Each header is only added if it isn't already set.

Common

The decorators in *django.views.decorators.common* allow per-view customization of *CommonMiddleware* behavior.

`no_append_slash`()

This decorator allows individual views to be excluded from *APPEND_SLASH* URL normalization.

3.3.4 File Uploads

When Django handles a file upload, the file data ends up placed in *request.FILES* (for more on the `request` object see the documentation for request and response objects). This document explains how files are stored on disk and in memory, and how to customize the default behavior.

> **Warning**
>
> There are security risks if you are accepting uploaded content from untrusted users! See the security guide's topic on User-uploaded content for mitigation details.

Basic file uploads

Consider a form containing a *FileField*:

Listing 6: `forms.py`

```
from django import forms


class UploadFileForm(forms.Form):
```

(continues on next page)

(continued from previous page)

```
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

A view handling this form will receive the file data in *request.FILES*, which is a dictionary containing a key for each *FileField* (or *ImageField*, or other *FileField* subclass) in the form. So the data from the above form would be accessible as `request.FILES['file']`.

Note that *request.FILES* will only contain data if the request method was `POST`, at least one file field was actually posted, and the `<form>` that posted the request has the attribute `enctype="multipart/form-data"`. Otherwise, `request.FILES` will be empty.

Most of the time, you'll pass the file data from `request` into the form as described in Binding uploaded files to a form. This would look something like:

Listing 7: `views.py`

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file


def upload_file(request):
    if request.method == "POST":
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES["file"])
            return HttpResponseRedirect("/success/url/")
    else:
        form = UploadFileForm()
    return render(request, "upload.html", {"form": form})
```

Notice that we have to pass *request.FILES* into the form's constructor; this is how file data gets bound into a form.

Here's a common way you might handle an uploaded file:

```
def handle_uploaded_file(f):
    with open("some/file/name.txt", "wb+") as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

Looping over `UploadedFile.chunks()` instead of using `read()` ensures that large files don't overwhelm your system's memory.

There are a few other methods and attributes available on `UploadedFile` objects; see *UploadedFile* for a complete reference.

Handling uploaded files with a model

If you're saving a file on a *Model* with a *FileField*, using a *ModelForm* makes this process much easier. The file object will be saved to the location specified by the *upload_to* argument of the corresponding *FileField* when calling `form.save()`:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField


def upload_file(request):
    if request.method == "POST":
        form = ModelFormWithFileField(request.POST, request.FILES)
        if form.is_valid():
            # file is saved
            form.save()
            return HttpResponseRedirect("/success/url/")
    else:
        form = ModelFormWithFileField()
    return render(request, "upload.html", {"form": form})
```

If you are constructing an object manually, you can assign the file object from *request.FILES* to the file field in the model:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm
from .models import ModelWithFileField


def upload_file(request):
    if request.method == "POST":
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FILES["file"])
            instance.save()
```

(continues on next page)

(continued from previous page)

```
            return HttpResponseRedirect("/success/url/")
    else:
        form = UploadFileForm()
    return render(request, "upload.html", {"form": form})
```

If you are constructing an object manually outside of a request, you can assign a *File* like object to the *FileField*:

```
from django.core.management.base import BaseCommand
from django.core.files.base import ContentFile


class MyCommand(BaseCommand):
    def handle(self, *args, **options):
        content_file = ContentFile(b"Hello world!", name="hello-world.txt")
        instance = ModelWithFileField(file_field=content_file)
        instance.save()
```

Uploading multiple files

If you want to upload multiple files using one form field, create a subclass of the field's widget and set its `allow_multiple_selected` class attribute to `True`.

In order for such files to be all validated by your form (and have the value of the field include them all), you will also have to subclass `FileField`. See below for an example.

> **Multiple file field**
>
> Django is likely to have a proper multiple file field support at some point in the future.

Listing 8: `forms.py`

```
from django import forms


class MultipleFileInput(forms.ClearableFileInput):
    allow_multiple_selected = True


class MultipleFileField(forms.FileField):
    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        kwargs.setdefault("widget", MultipleFileInput())
        super().__init__(*args, **kwargs)

    def clean(self, data, initial=None):
        single_file_clean = super().clean
        if isinstance(data, (list, tuple)):
            result = [single_file_clean(d, initial) for d in data]
        else:
            result = [single_file_clean(data, initial)]
        return result


class FileFieldForm(forms.Form):
    file_field = MultipleFileField()
```

Then override the `form_valid()` method of your *FormView* subclass to handle multiple file uploads:

Listing 9: `views.py`

```
from django.views.generic.edit import FormView
from .forms import FileFieldForm


class FileFieldFormView(FormView):
    form_class = FileFieldForm
    template_name = "upload.html"  # Replace with your template.
    success_url = "..."  # Replace with your URL or reverse().

    def form_valid(self, form):
        files = form.cleaned_data["file_field"]
        for f in files:
            ...  # Do something with each file.
        return super().form_valid(form)
```

Warning

This will allow you to handle multiple files at the form level only. Be aware that you cannot use it to put multiple files on a single model instance (in a single field), for example, even if the custom widget is used with a form field related to a model `FileField`.

Upload Handlers

When a user uploads a file, Django passes off the file data to an upload handler – a small class that handles file data as it gets uploaded. Upload handlers are initially defined in the *FILE_UPLOAD_HANDLERS* setting, which defaults to:

```
[
    "django.core.files.uploadhandler.MemoryFileUploadHandler",
    "django.core.files.uploadhandler.TemporaryFileUploadHandler",
]
```

Together *MemoryFileUploadHandler* and *TemporaryFileUploadHandler* provide Django's default file upload behavior of reading small files into memory and large ones onto disk.

You can write custom handlers that customize how Django handles files. You could, for example, use custom handlers to enforce user-level quotas, compress data on the fly, render progress bars, and even send data to another storage location directly without storing it locally. See Writing custom upload handlers for details on how you can customize or completely replace upload behavior.

Where uploaded data is stored

Before you save uploaded files, the data needs to be stored somewhere.

By default, if an uploaded file is smaller than 2.5 megabytes, Django will hold the entire contents of the upload in memory. This means that saving the file involves only a read from memory and a write to disk and thus is very fast.

However, if an uploaded file is too large, Django will write the uploaded file to a temporary file stored in your system's temporary directory. On a Unix-like platform this means you can expect Django to generate a file called something like `/tmp/tmpzfp6I6.upload`. If an upload is large enough, you can watch this file grow in size as Django streams the data onto disk.

These specifics – 2.5 megabytes; `/tmp`; etc. – are "reasonable defaults" which can be customized as described in the next section.

Changing upload handler behavior

There are a few settings which control Django's file upload behavior. See File Upload Settings for details.

Modifying upload handlers on the fly

Sometimes particular views require different upload behavior. In these cases, you can override upload handlers on a per-request basis by modifying `request.upload_handlers`. By default, this list will contain the upload handlers given by *FILE_UPLOAD_HANDLERS*, but you can modify the list as you would any other list.

For instance, suppose you've written a `ProgressBarUploadHandler` that provides feedback on upload progress to some sort of AJAX widget. You'd add this handler to your upload handlers like this:

```
request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
```

You'd probably want to use `list.insert()` in this case (instead of `append()`) because a progress bar handler would need to run before any other handlers. Remember, the upload handlers are processed in order.

If you want to replace the upload handlers completely, you can assign a new list:

```
request.upload_handlers = [ProgressBarUploadHandler(request)]
```

Note

You can only modify upload handlers before accessing `request.POST` or `request.FILES` – it doesn't make sense to change upload handlers after upload handling has already started. If you try to modify `request.upload_handlers` after reading from `request.POST` or `request.FILES` Django will throw an error.

Thus, you should always modify uploading handlers as early in your view as possible.

Also, `request.POST` is accessed by *CsrfViewMiddleware* which is enabled by default. This means you will need to use *csrf_exempt()* on your view to allow you to change the upload handlers. You will then need to use *csrf_protect()* on the function that actually processes the request. Note that this means that the handlers may start receiving the file upload before the CSRF checks have been done. Example code:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect


@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
    return _upload_file_view(request)


@csrf_protect
def _upload_file_view(request):
    # Process request
    ...
```

If you are using a class-based view, you will need to use *csrf_exempt()* on its *dispatch()* method and *csrf_protect()* on the method that actually processes the request. Example code:

```
from django.utils.decorators import method_decorator
from django.views import View
from django.views.decorators.csrf import csrf_exempt, csrf_protect


@method_decorator(csrf_exempt, name="dispatch")
class UploadFileView(View):
    def setup(self, request, *args, **kwargs):
        request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
        super().setup(request, *args, **kwargs)

    @method_decorator(csrf_protect)
    def post(self, request, *args, **kwargs):
        # Process request
        ...
```

3.3.5 Django shortcut functions

The package `django.shortcuts` collects helper functions and classes that "span" multiple levels of MVC. In other words, these functions/classes introduce controlled coupling for convenience's sake.

`render()`

render(request, template_name, context=None, content_type=None, status=None, using=None)

Combines a given template with a given context dictionary and returns an *HttpResponse* object with that rendered text.

Django does not provide a shortcut function which returns a *TemplateResponse* because the constructor of *TemplateResponse* offers the same level of convenience as *render()*.

Required arguments

request

The request object used to generate this response.

template_name

The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used. See the template loading documentation for more information on how templates are found.

Optional arguments

`context`
: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

`content_type`
: The MIME type to use for the resulting document. Defaults to `'text/html'`.

`status`
: The status code for the response. Defaults to 200.

`using`
: The *NAME* of a template engine to use for loading the template.

Example

The following example renders the template `myapp/index.html` with the MIME type *application/xhtml+xml*:

```
from django.shortcuts import render


def my_view(request):
    # View code here...
    return render(
        request,
        "myapp/index.html",
        {
            "foo": "bar",
        },
        content_type="application/xhtml+xml",
    )
```

This example is equivalent to:

```
from django.http import HttpResponse
from django.template import loader


def my_view(request):
    # View code here...
    t = loader.get_template("myapp/index.html")
    c = {"foo": "bar"}
```

(continues on next page)

(continued from previous page)

```
    return HttpResponse(t.render(c, request), content_type="application/xhtml+xml")
```

redirect()

redirect(to, *args, permanent=False, preserve_request=False, **kwargs)

Returns an *HttpResponseRedirect* to the appropriate URL for the arguments passed.

The arguments could be:

- A model: the model's *get_absolute_url()* function will be called.
- A view name, possibly with arguments: *reverse()* will be used to reverse-resolve the name.
- An absolute or relative URL, which will be used as-is for the redirect location.

By default, a temporary redirect is issued with a 302 status code. If `permanent=True`, a permanent redirect is issued with a 301 status code.

If `preserve_request=True`, the response instructs the user agent to preserve the method and body of the original request when issuing the redirect. In this case, temporary redirects use a 307 status code, and permanent redirects use a 308 status code. This is better illustrated in the following table:

permanent	preserve_request	HTTP status code
True	False	301
False	False	302
False	True	307
True	True	308

The argument `preserve_request` was added.

Examples

You can use the *redirect()* function in a number of ways.

1. By passing some object; that object's *get_absolute_url()* method will be called to figure out the redirect URL:

```
from django.shortcuts import redirect


def my_view(request):
    ...
    obj = MyModel.objects.get(...)
    return redirect(obj)
```

2. By passing the name of a view and optionally some positional or keyword arguments; the URL will be reverse resolved using the *reverse()* method:

```
def my_view(request):
    ...
    return redirect("some-view-name", foo="bar")
```

3. By passing a hardcoded URL to redirect to:

```
def my_view(request):
    ...
    return redirect("/some/url/")
```

   This also works with full URLs:

```
def my_view(request):
    ...
    return redirect("https://example.com/")
```

By default, *redirect()* returns a temporary redirect. All of the above forms accept a `permanent` argument; if set to `True` a permanent redirect will be returned:

```
def my_view(request):
    ...
    obj = MyModel.objects.get(...)
    return redirect(obj, permanent=True)
```

Additionally, the `preserve_request` argument can be used to preserve the original HTTP method:

```
def my_view(request):
    # ...
    obj = MyModel.objects.get(...)
    if request.method in ("POST", "PUT"):
        # Redirection preserves the original request method.
        return redirect(obj, preserve_request=True)
    # ...
```

get_object_or_404()

get_object_or_404(klass, *args, **kwargs)

aget_object_or_404(klass, *args, **kwargs)

Asynchronous version: `aget_object_or_404()`

Calls *get()* on a given model manager, but it raises *Http404* instead of the model's *DoesNotExist* exception.

Arguments

klass

A *Model* class, a *Manager*, or a *QuerySet* instance from which to get the object.

***args**

Q objects.

****kwargs**

Lookup parameters, which should be in the format accepted by `get()` and `filter()`.

Example

The following example gets the object with the primary key of 1 from `MyModel`:

```
from django.shortcuts import get_object_or_404


def my_view(request):
    obj = get_object_or_404(MyModel, pk=1)
```

This example is equivalent to:

```
from django.http import Http404


def my_view(request):
    try:
        obj = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No MyModel matches the given query.")
```

The most common use case is to pass a *Model*, as shown above. However, you can also pass a *QuerySet* instance:

```
queryset = Book.objects.filter(title__startswith="M")
get_object_or_404(queryset, pk=1)
```

The above example is a bit contrived since it's equivalent to doing:

```
get_object_or_404(Book, title__startswith="M", pk=1)
```

but it can be useful if you are passed the `queryset` variable from somewhere else.

Finally, you can also use a *Manager*. This is useful for example if you have a custom manager:

```
get_object_or_404(Book.dahl_objects, title="Matilda")
```

You can also use *related managers*:

```
author = Author.objects.get(name="Roald Dahl")
get_object_or_404(author.book_set, title="Matilda")
```

Note: As with `get()`, a *MultipleObjectsReturned* exception will be raised if more than one object is found.

get_list_or_404()

get_list_or_404(klass, *args, **kwargs)

aget_list_or_404(klass, *args, **kwargs)

Asynchronous version: `aget_list_or_404()`

Returns the result of *filter()* on a given model manager cast to a list, raising *Http404* if the resulting list is empty.

Arguments

klass

A *Model*, *Manager* or *QuerySet* instance from which to get the list.

***args**

Q objects.

****kwargs**

Lookup parameters, which should be in the format accepted by `get()` and `filter()`.

Example

The following example gets all published objects from `MyModel`:

```
from django.shortcuts import get_list_or_404


def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

This example is equivalent to:

```
from django.http import Http404


def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404("No MyModel matches the given query.")
```

3.3.6 Generic views

See Built-in class-based views API.

3.3.7 Middleware

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input or output.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, *AuthenticationMiddleware*, that associates users with requests using sessions.

This document explains how middleware works, how you activate middleware, and how to write your own middleware. Django ships with some built-in middleware you can use right out of the box. They're documented in the built-in middleware reference.

Writing your own middleware

A middleware factory is a callable that takes a `get_response` callable and returns a middleware. A middleware is a callable that takes a request and returns a response, just like a view.

A middleware can be written as a function that looks like this:

```
def simple_middleware(get_response):
    # One-time configuration and initialization.

    def middleware(request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
```

(continues on next page)

(continued from previous page)

```
    return middleware
```

Or it can be written as a class whose instances are callable, like this:

```
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
```

The `get_response` callable provided by Django might be the actual view (if this is the last listed middleware) or it might be the next middleware in the chain. The current middleware doesn't need to know or care what exactly it is, just that it represents whatever comes next.

The above is a slight simplification – the `get_response` callable for the last middleware in the chain won't be the actual view but rather a wrapper method from the handler which takes care of applying view middleware, calling the view with appropriate URL arguments, and applying template-response and exception middleware.

Middleware can either support only synchronous Python (the default), only asynchronous Python, or both. See Asynchronous support for details of how to advertise what you support, and know what kind of request you are getting.

Middleware can live anywhere on your Python path.

`__init__(get_response)`

Middleware factories must accept a `get_response` argument. You can also initialize some global state for the middleware. Keep in mind a couple of caveats:

- Django initializes your middleware with only the `get_response` argument, so you can't define `__init__()` as requiring any other arguments.

- Unlike the `__call__()` method which is called once per request, `__init__()` is called only once, when the web server starts.

Marking middleware as unused

It's sometimes useful to determine at startup time whether a piece of middleware should be used. In these cases, your middleware's `__init__()` method may raise *MiddlewareNotUsed*. Django will then remove that middleware from the middleware process and log a debug message to the django.request logger when *DEBUG* is `True`.

Activating middleware

To activate a middleware component, add it to the *MIDDLEWARE* list in your Django settings.

In *MIDDLEWARE*, each middleware component is represented by a string: the full Python path to the middleware factory's class or function name. For example, here's the default value created by *django-admin startproject*:

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]
```

A Django installation doesn't require any middleware — *MIDDLEWARE* can be empty, if you'd like — but it's strongly suggested that you at least use *CommonMiddleware*.

The order in *MIDDLEWARE* matters because a middleware can depend on other middleware. For instance, *AuthenticationMiddleware* stores the authenticated user in the session; therefore, it must run after *SessionMiddleware*. See Middleware ordering for some common hints about ordering of Django middleware classes.

Middleware order and layering

During the request phase, before calling the view, Django applies middleware in the order it's defined in *MIDDLEWARE*, top-down.

You can think of it like an onion: each middleware class is a "layer" that wraps the view, which is in the core of the onion. If the request passes through all the layers of the onion (each one calls `get_response` to pass the request in to the next layer), all the way to the view at the core, the response will then pass through every layer (in reverse order) on the way back out.

If one of the layers decides to short-circuit and return a response without ever calling its `get_response`, none of the layers of the onion inside that layer (including the view) will see the request or the response. The response will only return through the same layers that the request passed in through.

Other middleware hooks

Besides the basic request/response middleware pattern described earlier, you can add three other special methods to class-based middleware:

`process_view()`

`process_view`(request, view_func, view_args, view_kwargs)

`request` is an *`HttpRequest`* object. `view_func` is the Python function that Django is about to use. (It's the actual function object, not the name of the function as a string.) `view_args` is a list of positional arguments that will be passed to the view, and `view_kwargs` is a dictionary of keyword arguments that will be passed to the view. Neither `view_args` nor `view_kwargs` include the first view argument (`request`).

`process_view()` is called just before Django calls the view.

It should return either `None` or an *`HttpResponse`* object. If it returns `None`, Django will continue processing this request, executing any other `process_view()` middleware and, then, the appropriate view. If it returns an *`HttpResponse`* object, Django won't bother calling the appropriate view; it'll apply response middleware to that *`HttpResponse`* and return the result.

> **Note**
>
> Accessing *`request.POST`* inside middleware before the view runs or in `process_view()` will prevent any view running after the middleware from being able to modify the upload handlers for the request, and should normally be avoided.
>
> The *`CsrfViewMiddleware`* class can be considered an exception, as it provides the *`csrf_exempt()`* and *`csrf_protect()`* decorators which allow views to explicitly control at what point the CSRF validation should occur.

`process_exception()`

`process_exception`(request, exception)

`request` is an *`HttpRequest`* object. `exception` is an `Exception` object raised by the view function.

Django calls `process_exception()` when a view raises an exception. `process_exception()` should return either `None` or an *`HttpResponse`* object. If it returns an *`HttpResponse`* object, the template response and response middleware will be applied and the resulting response returned to the browser. Otherwise, default exception handling kicks in.

Again, middleware are run in reverse order during the response phase, which includes `process_exception`. If an exception middleware returns a response, the `process_exception` methods of the middleware classes above that middleware won't be called at all.

process_template_response()

process_template_response(request, response)

`request` is an *HttpRequest* object. `response` is the *TemplateResponse* object (or equivalent) returned by a Django view or by a middleware.

`process_template_response()` is called just after the view has finished executing, if the response instance has a `render()` method, indicating that it is a *TemplateResponse* or equivalent.

It must return a response object that implements a `render` method. It could alter the given `response` by changing `response.template_name` and `response.context_data`, or it could create and return a brand-new *TemplateResponse* or equivalent.

You don't need to explicitly render responses – responses will be automatically rendered once all template response middleware has been called.

Middleware are run in reverse order during the response phase, which includes `process_template_response()`.

Dealing with streaming responses

Unlike *HttpResponse*, *StreamingHttpResponse* does not have a `content` attribute. As a result, middleware can no longer assume that all responses will have a `content` attribute. If they need access to the content, they must test for streaming responses and adjust their behavior accordingly:

```
if response.streaming:
    response.streaming_content = wrap_streaming_content(response.streaming_content)
else:
    response.content = alter_content(response.content)
```

> **Note**
>
> `streaming_content` should be assumed to be too large to hold in memory. Response middleware may wrap it in a new generator, but must not consume it. Wrapping is typically implemented as follows:
>
> ```
> def wrap_streaming_content(content):
>     for chunk in content:
>         yield alter_content(chunk)
> ```

StreamingHttpResponse allows both synchronous and asynchronous iterators. The wrapping function must

match. Check *StreamingHttpResponse.is_async* if your middleware needs to support both types of iterator.

Exception handling

Django automatically converts exceptions raised by the view or by middleware into an appropriate HTTP response with an error status code. Certain exceptions are converted to 4xx status codes, while an unknown exception is converted to a 500 status code.

This conversion takes place before and after each middleware (you can think of it as the thin film in between each layer of the onion), so that every middleware can always rely on getting some kind of HTTP response back from calling its `get_response` callable. Middleware don't need to worry about wrapping their call to `get_response` in a `try/except` and handling an exception that might have been raised by a later middleware or the view. Even if the very next middleware in the chain raises an *Http404* exception, for example, your middleware won't see that exception; instead it will get an *HttpResponse* object with a *status_code* of 404.

You can set *DEBUG_PROPAGATE_EXCEPTIONS* to `True` to skip this conversion and propagate exceptions upward.

Asynchronous support

Middleware can support any combination of synchronous and asynchronous requests. Django will adapt requests to fit the middleware's requirements if it cannot support both, but at a performance penalty.

By default, Django assumes that your middleware is capable of handling only synchronous requests. To change these assumptions, set the following attributes on your middleware factory function or class:

- `sync_capable` is a boolean indicating if the middleware can handle synchronous requests. Defaults to `True`.
- `async_capable` is a boolean indicating if the middleware can handle asynchronous requests. Defaults to `False`.

If your middleware has both `sync_capable = True` and `async_capable = True`, then Django will pass it the request without converting it. In this case, you can work out if your middleware will receive async requests by checking if the `get_response` object you are passed is a coroutine function, using `asgiref.sync.iscoroutinefunction`.

The `django.utils.decorators` module contains *sync_only_middleware()*, *async_only_middleware()*, and *sync_and_async_middleware()* decorators that allow you to apply these flags to middleware factory functions.

The returned callable must match the sync or async nature of the `get_response` method. If you have an asynchronous `get_response`, you must return a coroutine function (`async def`).

`process_view`, `process_template_response` and `process_exception` methods, if they are provided, should also be adapted to match the sync/async mode. However, Django will individually adapt them as required if you do not, at an additional performance penalty.

Here's an example of how to create a middleware function that supports both:

```
from asgiref.sync import iscoroutinefunction
from django.utils.decorators import sync_and_async_middleware


@sync_and_async_middleware
def simple_middleware(get_response):
    # One-time configuration and initialization goes here.
    if iscoroutinefunction(get_response):

        async def middleware(request):
            # Do something here!
            response = await get_response(request)
            return response

    else:

        def middleware(request):
            # Do something here!
            response = get_response(request)
            return response

    return middleware
```

> **Note**
>
> If you declare a hybrid middleware that supports both synchronous and asynchronous calls, the kind of call you get may not match the underlying view. Django will optimize the middleware call stack to have as few sync/async transitions as possible.
>
> Thus, even if you are wrapping an async view, you may be called in sync mode if there is other, synchronous middleware between you and the view.

When using an asynchronous class-based middleware, you must ensure that instances are correctly marked as coroutine functions:

```
from asgiref.sync import iscoroutinefunction, markcoroutinefunction


class AsyncMiddleware:
    async_capable = True
    sync_capable = False
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, get_response):
        self.get_response = get_response
        if iscoroutinefunction(self.get_response):
            markcoroutinefunction(self)

    async def __call__(self, request):
        response = await self.get_response(request)
        # Some logic ...
        return response
```

Upgrading pre-Django 1.10-style middleware

class django.utils.deprecation.**MiddlewareMixin**

Django provides `django.utils.deprecation.MiddlewareMixin` to ease creating middleware classes that are compatible with both *MIDDLEWARE* and the old `MIDDLEWARE_CLASSES`, and support synchronous and asynchronous requests. All middleware classes included with Django are compatible with both settings.

The mixin provides an `__init__()` method that requires a `get_response` argument and stores it in `self.get_response`.

The `__call__()` method:

1. Calls `self.process_request(request)` (if defined).
2. Calls `self.get_response(request)` to get the response from later middleware and the view.
3. Calls `self.process_response(request, response)` (if defined).
4. Returns the response.

If used with `MIDDLEWARE_CLASSES`, the `__call__()` method will never be used; Django calls `process_request()` and `process_response()` directly.

In most cases, inheriting from this mixin will be sufficient to make an old-style middleware compatible with the new system with sufficient backwards-compatibility. The new short-circuiting semantics will be harmless or even beneficial to the existing middleware. In a few cases, a middleware class may need some changes to adjust to the new semantics.

These are the behavioral differences between using *MIDDLEWARE* and `MIDDLEWARE_CLASSES`:

1. Under `MIDDLEWARE_CLASSES`, every middleware will always have its `process_response` method called, even if an earlier middleware short-circuited by returning a response from its `process_request` method. Under *MIDDLEWARE*, middleware behaves more like an onion: the layers that a response goes through on the way out are the same layers that saw the request on the way in. If a middleware short-circuits, only that middleware and the ones before it in *MIDDLEWARE* will see the response.

2. Under `MIDDLEWARE_CLASSES`, `process_exception` is applied to exceptions raised from a middleware `process_request` method. Under *MIDDLEWARE*, `process_exception` applies only to exceptions raised from the view (or from the `render` method of a *TemplateResponse*). Exceptions raised from a middleware are converted to the appropriate HTTP response and then passed to the next middleware.

3. Under `MIDDLEWARE_CLASSES`, if a `process_response` method raises an exception, the `process_response` methods of all earlier middleware are skipped and a `500 Internal Server Error` HTTP response is always returned (even if the exception raised was e.g. an *Http404*). Under *MIDDLEWARE*, an exception raised from a middleware will immediately be converted to the appropriate HTTP response, and then the next middleware in line will see that response. Middleware are never skipped due to a middleware raising an exception.

3.3.8 How to use sessions

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID – not the data itself (unless you're using the cookie based backend).

Enabling sessions

Sessions are implemented via a piece of middleware.

To enable session functionality, do the following:

- Edit the *MIDDLEWARE* setting and make sure it contains `'django.contrib.sessions.middleware.SessionMiddleware'`. The default `settings.py` created by `django-admin startproject` has `SessionMiddleware` activated.

If you don't want to use sessions, you might as well remove the `SessionMiddleware` line from *MIDDLEWARE* and `'django.contrib.sessions'` from your *INSTALLED_APPS*. It'll save you a small bit of overhead.

Configuring the session engine

By default, Django stores sessions in your database (using the model `django.contrib.sessions.models.Session`). Though this is convenient, in some setups it's faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

Using database-backed sessions

If you want to use a database-backed session, you need to add `'django.contrib.sessions'` to your *INSTALLED_APPS* setting.

Once you have configured your installation, run `manage.py migrate` to install the single database table that stores session data.

Using cached sessions

For better performance, you may want to use a cache-based session backend.

To store session data using Django's cache system, you'll first need to make sure you've configured your cache; see the cache documentation for details.

> **Warning**
>
> You should only use cache-based sessions if you're using the Memcached or Redis cache backend. The local-memory cache backend doesn't retain data long enough to be a good choice, and it'll be faster to use file or database sessions directly instead of sending everything through the file or database cache backends. Additionally, the local-memory cache backend is NOT multi-process safe, therefore probably not a good choice for production environments.

If you have multiple caches defined in *CACHES*, Django will use the default cache. To use another cache, set *SESSION_CACHE_ALIAS* to the name of that cache.

Once your cache is configured, you have to choose between a database-backed cache or a non-persistent cache.

The cached database backend (`cached_db`) uses a write-through cache – session writes are applied to both the database and cache, in that order. If writing to the cache fails, the exception is handled and logged via the sessions logger, to avoid failing an otherwise successful write operation.

Handling and logging of exceptions when writing to the cache was added.

Session reads use the cache, or the database if the data has been evicted from the cache. To use this backend, set *SESSION_ENGINE* to `"django.contrib.sessions.backends.cached_db"`, and follow the configuration instructions for the using database-backed sessions.

The cache backend (`cache`) stores session data only in your cache. This is faster because it avoids database persistence, but you will have to consider what happens when cache data is evicted. Eviction can occur if the cache fills up or the cache server is restarted, and it will mean session data is lost, including logging out users. To use this backend, set *SESSION_ENGINE* to `"django.contrib.sessions.backends.cache"`.

The cache backend can be made persistent by using a persistent cache, such as Redis with appropriate configuration. But unless your cache is definitely configured for sufficient persistence, opt for the cached database backend. This avoids edge cases caused by unreliable data storage in production.

Using file-based sessions

To use file-based sessions, set the *SESSION_ENGINE* setting to `"django.contrib.sessions.backends.file"`.

You might also want to set the *SESSION_FILE_PATH* setting (which defaults to output from `tempfile.gettempdir()`, most likely `/tmp`) to control where Django stores session files. Be sure to check that your web server has permissions to read and write to this location.

Using cookie-based sessions

To use cookies-based sessions, set the *SESSION_ENGINE* setting to `"django.contrib.sessions.backends.signed_cookies"`. The session data will be stored using Django's tools for cryptographic signing and the *SECRET_KEY* setting.

> **Note**
>
> It's recommended to leave the *SESSION_COOKIE_HTTPONLY* setting on `True` to prevent access to the stored data from JavaScript.

> **Warning**
>
> The session data is signed but not encrypted
>
> When using the cookies backend the session data can be read by the client.
>
> A MAC (Message Authentication Code) is used to protect the data against changes by the client, so that the session data will be invalidated when being tampered with. The same invalidation happens if the client storing the cookie (e.g. your user's browser) can't store all of the session cookie and drops data. Even though Django compresses the data, it's still entirely possible to exceed the common limit of 4096 bytes per cookie.
>
> No freshness guarantee
>
> Note also that while the MAC can guarantee the authenticity of the data (that it was generated by your site, and not someone else), and the integrity of the data (that it is all there and correct), it cannot guarantee freshness i.e. that you are being sent back the last thing you sent to the client. This means that for some uses of session data, the cookie backend might open you up to replay attacks. Unlike other session backends which keep a server-side record of each session and invalidate it when a user logs out, cookie-based sessions are not invalidated when a user logs out. Thus if an attacker steals a user's cookie, they can use that cookie to login as that user even if the user logs out. Cookies will only be detected as 'stale' if they are older than your *SESSION_COOKIE_AGE*.
>
> Performance
>
> Finally, the size of a cookie can have an impact on the speed of your site.

Using sessions in views

When `SessionMiddleware` is activated, each *HttpRequest* object – the first argument to any Django view function – will have a `session` attribute, which is a dictionary-like object.

You can read it and write to `request.session` at any point in your view. You can edit it multiple times.

`class backends.base.SessionBase`

This is the base class for all session objects. It has the following standard dictionary methods:

__getitem__(key)

Example: `fav_color = request.session['fav_color']`

__setitem__(key, value)

Example: `request.session['fav_color'] = 'blue'`

__delitem__(key)

Example: `del request.session['fav_color']`. This raises `KeyError` if the given `key` isn't already in the session.

__contains__(key)

Example: `'fav_color' in request.session`

get(key, default=None)

aget(key, default=None)

Asynchronous version: `aget()`

Example: `fav_color = request.session.get('fav_color', 'red')`

`aget()` function was added.

aset(key, value)

Example: `await request.session.aset('fav_color', 'red')`

update(dict)

aupdate(dict)

Asynchronous version: `aupdate()`

Example: `request.session.update({'fav_color': 'red'})`

`aupdate()` function was added.

pop(key, default=__not_given)

apop(key, default=__not_given)

Asynchronous version: `apop()`

Example: `fav_color = request.session.pop('fav_color', 'blue')`

`apop()` function was added.

keys()

akeys()

Asynchronous version: `akeys()`

`akeys()` function was added.

`values()`

`avalues()`

Asynchronous version: `avalues()`

`avalues()` function was added.

`has_key`(key)

`ahas_key`(key)

Asynchronous version: `ahas_key()`

`ahas_key()` function was added.

`items()`

`aitems()`

Asynchronous version: `aitems()`

`aitems()` function was added.

`setdefault()`

`asetdefault()`

Asynchronous version: `asetdefault()`

`asetdefault()` function was added.

`clear()`

It also has these methods:

`flush()`

`aflush()`

Asynchronous version: `aflush()`

Deletes the current session data from the session and deletes the session cookie. This is used if you want to ensure that the previous session data can't be accessed again from the user's browser (for example, the *`django.contrib.auth.logout()`* function calls it).

`aflush()` function was added.

`set_test_cookie()`

`aset_test_cookie()`

Asynchronous version: `aset_test_cookie()`

Sets a test cookie to determine whether the user's browser supports cookies. Due to the way cookies work, you won't be able to test this until the user's next page request. See Setting test cookies below for more information.

`aset_test_cookie()` function was added.

`test_cookie_worked()`

`atest_cookie_worked()`

Asynchronous version: `atest_cookie_worked()`

Returns either `True` or `False`, depending on whether the user's browser accepted the test cookie. Due to the way cookies work, you'll have to call `set_test_cookie()` or `aset_test_cookie()` on a previous, separate page request. See Setting test cookies below for more information.

`atest_cookie_worked()` function was added.

`delete_test_cookie()`

`adelete_test_cookie()`

Asynchronous version: `adelete_test_cookie()`

Deletes the test cookie. Use this to clean up after yourself.

`adelete_test_cookie()` function was added.

`get_session_cookie_age()`

Returns the value of the setting *SESSION_COOKIE_AGE*. This can be overridden in a custom session backend.

`set_expiry`(value)

`aset_expiry`(value)

Asynchronous version: `aset_expiry()`

Sets the expiration time for the session. You can pass a number of different values:

- If `value` is an integer, the session will expire after that many seconds of inactivity. For example, calling `request.session.set_expiry(300)` would make the session expire in 5 minutes.
- If `value` is a `datetime` or `timedelta` object, the session will expire at that specific date/time.
- If `value` is `0`, the user's session cookie will expire when the user's web browser is closed.
- If `value` is `None`, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was modified.

`aset_expiry()` function was added.

`get_expiry_age()`

`aget_expiry_age()`

Asynchronous version: `aget_expiry_age()`

Returns the number of seconds until this session expires. For sessions with no custom expiration (or those set to expire at browser close), this will equal *SESSION_COOKIE_AGE*.

This function accepts two optional keyword arguments:

- `modification`: last modification of the session, as a `datetime` object. Defaults to the current time.
- `expiry`: expiry information for the session, as a `datetime` object, an `int` (in seconds), or `None`. Defaults to the value stored in the session by *`set_expiry()`*/*`aset_expiry()`*, if there is one, or `None`.

> **Note**
>
> This method is used by session backends to determine the session expiry age in seconds when saving the session. It is not really intended for usage outside of that context.
>
> In particular, while it is possible to determine the remaining lifetime of a session just when you have the correct `modification` value and the `expiry` is set as a `datetime` object, where you do have the `modification` value, it is more straight-forward to calculate the expiry by-hand:
>
> ```
> expires_at = modification + timedelta(seconds=settings.SESSION_COOKIE_AGE)
> ```

`aget_expiry_age()` function was added.

`get_expiry_date()`

`aget_expiry_date()`

Asynchronous version: `aget_expiry_date()`

Returns the date this session will expire. For sessions with no custom expiration (or those set to expire at browser close), this will equal the date *`SESSION_COOKIE_AGE`* seconds from now.

This function accepts the same keyword arguments as *`get_expiry_age()`*, and similar notes on usage apply.

`aget_expiry_date()` function was added.

`get_expire_at_browser_close()`

`aget_expire_at_browser_close()`

Asynchronous version: `aget_expire_at_browser_close()`

Returns either `True` or `False`, depending on whether the user's session cookie will expire when the user's web browser is closed.

`aget_expire_at_browser_close()` function was added.

`clear_expired()`

`aclear_expired()`

Asynchronous version: `aclear_expired()`

Removes expired sessions from the session store. This class method is called by *clearsessions*.

`aclear_expired()` function was added.

`cycle_key()`

`acycle_key()`

Asynchronous version: `acycle_key()`

Creates a new session key while retaining the current session data. *django.contrib.auth.login()* calls this method to mitigate against session fixation.

`acycle_key()` function was added.

Session serialization

By default, Django serializes session data using JSON. You can use the *SESSION_SERIALIZER* setting to customize the session serialization format. Even with the caveats described in Write your own serializer, we highly recommend sticking with JSON serialization especially if you are using the cookie backend.

For example, here's an attack scenario if you use `pickle` to serialize session data. If you're using the signed cookie session backend and *SECRET_KEY* (or any key of *SECRET_KEY_FALLBACKS*) is known by an attacker (there isn't an inherent vulnerability in Django that would cause it to leak), the attacker could insert a string into their session which, when unpickled, executes arbitrary code on the server. The technique for doing so is simple and easily available on the internet. Although the cookie session storage signs the cookie-stored data to prevent tampering, a *SECRET_KEY* leak immediately escalates to a remote code execution vulnerability.

Bundled serializers

`class serializers.JSONSerializer`

A wrapper around the JSON serializer from *django.core.signing*. Can only serialize basic data types.

In addition, as JSON supports only string keys, note that using non-string keys in `request.session` won't work as expected:

```
>>> # initial assignment
>>> request.session[0] = "bar"
>>> # subsequent requests following serialization & deserialization
>>> # of session data
>>> request.session[0]  # KeyError
>>> request.session["0"]
'bar'
```

Similarly, data that can't be encoded in JSON, such as non-UTF8 bytes like `'\xd9'` (which raises `UnicodeDecodeError`), can't be stored.

See the Write your own serializer section for more details on limitations of JSON serialization.

Write your own serializer

Note that the *JSONSerializer* cannot handle arbitrary Python data types. As is often the case, there is a trade-off between convenience and security. If you wish to store more advanced data types including `datetime` and `Decimal` in JSON backed sessions, you will need to write a custom serializer (or convert such values to a JSON serializable object before storing them in `request.session`). While serializing these values is often straightforward (*DjangoJSONEncoder* may be helpful), writing a decoder that can reliably get back the same thing that you put in is more fragile. For example, you run the risk of returning a `datetime` that was actually a string that just happened to be in the same format chosen for `datetime`s).

Your serializer class must implement two methods, `dumps(self, obj)` and `loads(self, data)`, to serialize and deserialize the dictionary of session data, respectively.

Session object guidelines

- Use normal Python strings as dictionary keys on `request.session`. This is more of a convention than a hard-and-fast rule.
- Session dictionary keys that begin with an underscore are reserved for internal use by Django.
- Don't override `request.session` with a new object, and don't access or set its attributes. Use it like a Python dictionary.

Examples

This simplistic view sets a `has_commented` variable to `True` after a user posts a comment. It doesn't let a user post a comment more than once:

```
def post_comment(request, new_comment):
    if request.session.get("has_commented", False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session["has_commented"] = True
    return HttpResponse("Thanks for your comment!")
```

This simplistic view logs in a "member" of the site:

```
def login(request):
    m = Member.objects.get(username=request.POST["username"])
    if m.check_password(request.POST["password"]):
        request.session["member_id"] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't match.")
```

...And this one logs a member out, according to `login()` above:

```
def logout(request):
    try:
        del request.session["member_id"]
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

The standard *django.contrib.auth.logout()* function actually does a bit more than this to prevent inadvertent data leakage. It calls the *flush()* method of `request.session`. We are using this example as a demonstration of how to work with session objects, not as a full `logout()` implementation.

Setting test cookies

As a convenience, Django provides a way to test whether the user's browser accepts cookies. Call the *set_test_cookie()* method of `request.session` in a view, and call *test_cookie_worked()* in a subsequent view – not in the same view call.

This awkward split between `set_test_cookie()` and `test_cookie_worked()` is necessary due to the way cookies work. When you set a cookie, you can't actually tell whether a browser accepted it until the browser's next request.

It's good practice to use *delete_test_cookie()* to clean up after yourself. Do this after you've verified that the test cookie worked.

Here's a typical usage example:

```
from django.http import HttpResponse
from django.shortcuts import render


def login(request):
    if request.method == "POST":
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render(request, "foo/login_form.html")
```

Support for setting test cookies in asynchronous view functions was added.

Using sessions out of views

> **Note**
>
> The examples in this section import the `SessionStore` object directly from the `django.contrib.sessions.backends.db` backend. In your own code, you should consider importing `SessionStore` from the session engine designated by *`SESSION_ENGINE`*, as below:
>
> ```
> >>> from importlib import import_module
> >>> from django.conf import settings
> >>> SessionStore = import_module(settings.SESSION_ENGINE).SessionStore
> ```

An API is available to manipulate session data outside of a view:

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since datetimes are not serializable in JSON.
>>> s["last_login"] = 1376587691
>>> s.create()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'
>>> s = SessionStore(session_key="2b1189a188b44ad18c35e113ac6ceead")
>>> s["last_login"]
1376587691
```

`SessionStore.create()` is designed to create a new session (i.e. one not loaded from the session store and with `session_key=None`). `save()` is designed to save an existing session (i.e. one loaded from the session store). Calling `save()` on a new session may also work but has a small chance of generating a `session_key` that collides with an existing one. `create()` calls `save()` and loops until an unused `session_key` is generated.

If you're using the `django.contrib.sessions.backends.db` backend, each session is a normal Django model. The `Session` model is defined in django/contrib/sessions/models.py. Because it's a normal model, you can access sessions using the normal Django database API:

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk="2b1189a188b44ad18c35e113ac6ceead")
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

Note that you'll need to call *`get_decoded()`* to get the session dictionary. This is necessary because the dictionary is stored in an encoded format:

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

When sessions are saved

By default, Django only saves to the session database when the session has been modified – that is if any of its dictionary values have been assigned or deleted:

```
# Session is modified.
request.session["foo"] = "bar"

# Session is modified.
del request.session["foo"]

# Session is modified.
request.session["foo"] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session["foo"]["bar"] = "baz"
```

In the last case of the above example, we can tell the session object explicitly that it has been modified by setting the `modified` attribute on the session object:

```
request.session.modified = True
```

To change this default behavior, set the *SESSION_SAVE_EVERY_REQUEST* setting to `True`. When set to `True`, Django will save the session to the database on every single request.

Note that the session cookie is only sent when a session has been created or modified. If *SESSION_SAVE_EVERY_REQUEST* is `True`, the session cookie will be sent on every request.

Similarly, the `expires` part of a session cookie is updated each time the session cookie is sent.

The session is not saved if the response's status code is 500.

Browser-length sessions vs. persistent sessions

You can control whether the session framework uses browser-length sessions vs. persistent sessions with the *SESSION_EXPIRE_AT_BROWSER_CLOSE* setting.

By default, *SESSION_EXPIRE_AT_BROWSER_CLOSE* is set to `False`, which means session cookies will be stored in users' browsers for as long as *SESSION_COOKIE_AGE*. Use this if you don't want people to have to log in

every time they open a browser.

If *SESSION_EXPIRE_AT_BROWSER_CLOSE* is set to `True`, Django will use browser-length cookies – cookies that expire as soon as the user closes their browser. Use this if you want people to have to log in every time they open a browser.

This setting is a global default and can be overwritten at a per-session level by explicitly calling the *set_expiry()* method of `request.session` as described above in using sessions in views.

> **Note**
>
> Some browsers (Chrome, for example) provide settings that allow users to continue browsing sessions after closing and reopening the browser. In some cases, this can interfere with the *SESSION_EXPIRE_AT_BROWSER_CLOSE* setting and prevent sessions from expiring on browser close. Please be aware of this while testing Django applications which have the *SESSION_EXPIRE_AT_BROWSER_CLOSE* setting enabled.

Clearing the session store

As users create new sessions on your website, session data can accumulate in your session store. If you're using the database backend, the `django_session` database table will grow. If you're using the file backend, your temporary directory will contain an increasing number of files.

To understand this problem, consider what happens with the database backend. When a user logs in, Django adds a row to the `django_session` database table. Django updates this row each time the session data changes. If the user logs out manually, Django deletes the row. But if the user does not log out, the row never gets deleted. A similar process happens with the file backend.

Django does not provide automatic purging of expired sessions. Therefore, it's your job to purge expired sessions on a regular basis. Django provides a clean-up management command for this purpose: *clearsessions*. It's recommended to call this command on a regular basis, for example as a daily cron job.

Note that the cache backend isn't vulnerable to this problem, because caches automatically delete stale data. Neither is the cookie backend, because the session data is stored by the users' browsers.

Settings

A few Django settings give you control over session behavior:

- *SESSION_CACHE_ALIAS*
- *SESSION_COOKIE_AGE*
- *SESSION_COOKIE_DOMAIN*
- *SESSION_COOKIE_HTTPONLY*
- *SESSION_COOKIE_NAME*

- *SESSION_COOKIE_PATH*
- *SESSION_COOKIE_SAMESITE*
- *SESSION_COOKIE_SECURE*
- *SESSION_ENGINE*
- *SESSION_EXPIRE_AT_BROWSER_CLOSE*
- *SESSION_FILE_PATH*
- *SESSION_SAVE_EVERY_REQUEST*
- *SESSION_SERIALIZER*

Session security

Subdomains within a site are able to set cookies on the client for the whole domain. This makes session fixation possible if cookies are permitted from subdomains not controlled by trusted users.

For example, an attacker could log into `good.example.com` and get a valid session for their account. If the attacker has control over `bad.example.com`, they can use it to send their session key to you since a subdomain is permitted to set cookies on `*.example.com`. When you visit `good.example.com`, you'll be logged in as the attacker and might inadvertently enter your sensitive personal data (e.g. credit card info) into the attacker's account.

Another possible attack would be if `good.example.com` sets its *SESSION_COOKIE_DOMAIN* to `"example.com"` which would cause session cookies from that site to be sent to `bad.example.com`.

Technical details

- The session dictionary accepts any `json` serializable value when using *JSONSerializer*.
- Session data is stored in a database table named `django_session` .
- Django only sends a cookie if it needs to. If you don't set any session data, it won't send a session cookie.

The `SessionStore` object

When working with sessions internally, Django uses a session store object from the corresponding session engine. By convention, the session store object class is named `SessionStore` and is located in the module designated by *SESSION_ENGINE*.

All `SessionStore` subclasses available in Django implement the following data manipulation methods:

- `exists()`
- `create()`
- `save()`
- `delete()`

- `load()`
- *`clear_expired()`*

An asynchronous interface for these methods is provided by wrapping them with `sync_to_async()`. They can be implemented directly if an async-native implementation is available:

- `aexists()`
- `acreate()`
- `asave()`
- `adelete()`
- `aload()`
- *`aclear_expired()`*

In order to build a custom session engine or to customize an existing one, you may create a new class inheriting from *`SessionBase`* or any other existing `SessionStore` class.

You can extend the session engines, but doing so with database-backed session engines generally requires some extra effort (see the next section for details).

`aexists()`, `acreate()`, `asave()`, `adelete()`, `aload()`, and `aclear_expired()` methods were added.

Extending database-backed session engines

Creating a custom database-backed session engine built upon those included in Django (namely `db` and `cached_db`) may be done by inheriting *`AbstractBaseSession`* and either `SessionStore` class.

`AbstractBaseSession` and `BaseSessionManager` are importable from `django.contrib.sessions.base_session` so that they can be imported without including `django.contrib.sessions` in *`INSTALLED_APPS`*.

class base_session.**AbstractBaseSession**

The abstract base session model.

session_key

Primary key. The field itself may contain up to 40 characters. The current implementation generates a 32-character string (a random sequence of digits and lowercase ASCII letters).

session_data

A string containing an encoded and serialized session dictionary.

expire_date

A datetime designating when the session expires.

Expired sessions are not available to a user, however, they may still be stored in the database until the *`clearsessions`* management command is run.

classmethod get_session_store_class()

Returns a session store class to be used with this session model.

get_decoded()

Returns decoded session data.

Decoding is performed by the session store class.

You can also customize the model manager by subclassing *BaseSessionManager*:

class base_session.BaseSessionManager

encode(session_dict)

Returns the given session dictionary serialized and encoded as a string.

Encoding is performed by the session store class tied to a model class.

save(session_key, session_dict, expire_date)

Saves session data for a provided session key, or deletes the session in case the data is empty.

Customization of `SessionStore` classes is achieved by overriding methods and properties described below:

class backends.db.SessionStore

Implements database-backed session store.

classmethod get_model_class()

Override this method to return a custom session model if you need one.

create_model_instance(data)

Returns a new instance of the session model object, which represents the current session state.

Overriding this method provides the ability to modify session model data before it's saved to database.

class backends.cached_db.SessionStore

Implements cached database-backed session store.

cache_key_prefix

A prefix added to a session key to build a cache key string.

Example

The example below shows a custom database-backed session engine that includes an additional database column to store an account ID (thus providing an option to query the database for all active sessions for an account):

```
from django.contrib.sessions.backends.db import SessionStore as DBStore
from django.contrib.sessions.base_session import AbstractBaseSession
```

(continues on next page)

(continued from previous page)

```
from django.db import models


class CustomSession(AbstractBaseSession):
    account_id = models.IntegerField(null=True, db_index=True)

    @classmethod
    def get_session_store_class(cls):
        return SessionStore


class SessionStore(DBStore):
    @classmethod
    def get_model_class(cls):
        return CustomSession

    def create_model_instance(self, data):
        obj = super().create_model_instance(data)
        try:
            account_id = int(data.get("_auth_user_id"))
        except (ValueError, TypeError):
            account_id = None
        obj.account_id = account_id
        return obj
```

If you are migrating from the Django's built-in `cached_db` session store to a custom one based on `cached_db`, you should override the cache key prefix in order to prevent a namespace clash:

```
class SessionStore(CachedDBStore):
    cache_key_prefix = "mysessions.custom_cached_db_backend"

    # ...
```

Session IDs in URLs

The Django sessions framework is entirely, and solely, cookie-based. It does not fall back to putting session IDs in URLs as a last resort, as PHP does. This is an intentional design decision. Not only does that behavior make URLs ugly, it makes your site vulnerable to session-ID theft via the "Referer" header.

3.4 Working with forms

About this document

This document provides an introduction to the basics of web forms and how they are handled in Django. For a more detailed look at specific areas of the forms API, see The Forms API, Form fields, and Form and field validation.

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms.

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

3.4.1 HTML forms

In HTML, a form is a collection of elements inside `<form>...</form>` that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form `<input>` elements to achieve these effects.

As well as its `<input>` elements, a form must specify two things:

- where: the URL to which the data corresponding to the user's input should be returned
- how: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several `<input>` elements: one of `type="text"` for the username, one of `type="password"` for the password, and one of `type="submit"` for the "Log in" button. It also contains some hidden text fields that the user doesn't see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the `<form>`'s `action` attribute - `/admin/` - and that it should be sent using the HTTP mechanism specified by the `method` attribute - `post`.

When the `<input type="submit" value="Log in">` element is triggered, the data is returned to `/admin/`.

`GET` and `POST`

`GET` and `POST` are the only HTTP methods to use when dealing with forms.

Django's login form is returned using the `POST` method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.

`GET`, by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains the address where the data must be sent, as well as the data keys and values. You can see this in action if you do a search in the Django documentation, which will produce a URL of the form `https://docs.djangoproject.com/search/?q=forms&release=1`.

`GET` and `POST` are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes in the database - should use `POST`. `GET` should be used only for requests that do not affect the state of the system.

`GET` would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A web application that uses `GET` requests for admin forms is a security risk: it can be easy for an attacker to mimic a form's request to gain access to sensitive parts of the system. `POST`, coupled with other protections like Django's CSRF protection offers more control over access.

On the other hand, `GET` is suitable for things like a web search form, because the URLs that represent a `GET` request can easily be bookmarked, shared, or resubmitted.

3.4.2 Django's role in forms

Handling forms is a complex business. Consider Django's admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.

Django's form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

It is possible to write code that does all of this manually, but Django can take care of it all for you.

3.4.3 Forms in Django

We've described HTML forms briefly, but an HTML `<form>` is just one part of the machinery required.

In the context of a web application, 'form' might refer to that HTML `<form>`, or to the Django *`Form`* that produces it, or to the structured data returned when it is submitted, or to the end-to-end working collection of these parts.

The Django `Form` class

At the heart of this system of components is Django's *`Form`* class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a *`Form`* class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form `<input>` elements. (A *`ModelForm`* maps a model class's fields to HTML form `<input>` elements via a *`Form`*; this is what the Django admin is based upon.)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A *`DateField`* and a *`FileField`* handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "widget" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

Instantiating, processing, and rendering forms

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)
2. pass it to the template context
3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but there are some key differences.

In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that's what we do when we want the user to populate it.

So when we handle a model instance in a view, we typically retrieve it from the database. When we're dealing with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or prepopulate it, for example with:

- data from a saved model instance (as in the case of admin forms for editing)
- data that we have collated from other sources
- data received from a previous HTML form submission

The last of these cases is the most interesting, because it's what makes it possible for users not just to read a website, but to send information back to it too.

3.4.4 Building a form

The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
    <input type="submit" value="OK">
</form>
```

This tells the browser to return the form data to the URL `/your-name/`, using the `POST` method. It will display a text field, labeled "Your name:", and a button marked "OK". If the template context contains a `current_name` variable, that will be used to pre-fill the `your_name` field.

You'll need a view that renders the template containing the HTML form, and that can supply the `current_name` field as appropriate.

When the form is submitted, the `POST` request which is sent to the server will contain the form data.

Now you'll also need a view corresponding to that `/your-name/` URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be prepopulated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Django to do most of this work for us.

Building a form in Django

The `Form` class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

Listing 10: `forms.py`

```
from django import forms


class NameForm(forms.Form):
    your_name = forms.CharField(label="Your name", max_length=100)
```

This defines a *Form* class with a single field (`your_name`). We've applied a human-friendly label to the field, which will appear in the `<label>` when it's rendered (although in this case, the *label* we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by *max_length*. This does two things. It puts a `maxlength="100"` on the HTML `<input>` (so the browser should prevent the user from entering more than that number of characters in the first place). It also means that when Django receives the form back from the browser, it will validate the length of the data.

A *Form* instance has an *is_valid()* method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:

- return `True`
- place the form's data in its *cleaned_data* attribute.

The whole form, when rendered for the first time, will look like:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required>
```

Note that it does not include the `<form>` tags, or a submit button. We'll have to provide those ourselves in the template.

The view

Form data sent back to a Django website is processed by a view, generally the same view which published the form. This allows us to reuse some of the same logic.

To handle the form we need to instantiate it in the view for the URL where we want it to be published:

Listing 11: `views.py`

```
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import NameForm


def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == "POST":
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
```

(continues on next page)

(continued from previous page)

```
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect("/thanks/")

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, "name.html", {"form": form})
```

If we arrive at this view with a `GET` request, it will create an empty form instance and place it in the template context to be rendered. This is what we can expect to happen the first time we visit the URL.

If the form is submitted using a `POST` request, the view will once again create a form instance and populate it with data from the request: `form = NameForm(request.POST)` This is called "binding data to the form" (it is now a bound form).

We call the form's `is_valid()` method; if it's not `True`, we go back to the template with the form. This time the form is no longer empty (unbound) so the HTML form will be populated with the data previously submitted, where it can be edited and corrected as required.

If `is_valid()` is `True`, we'll now be able to find all the validated form data in its `cleaned_data` attribute. We can use this data to update the database or do other processing before sending an HTTP redirect to the browser telling it where to go next.

The template

We don't need to do much in our `name.html` template:

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

All the form's fields and their attributes will be unpacked into HTML markup from that `{{ form }}` by Django's template language.

> **Forms and Cross Site Request Forgery protection**
>
> Django ships with an easy-to-use protection against Cross Site Request Forgeries. When submitting a form via `POST` with CSRF protection enabled you must use the *csrf_token* template tag as in the preced-

ing example. However, since CSRF protection is not directly tied to forms in templates, this tag is omitted from the following examples in this document.

HTML5 input types and browser validation

If your form includes a *URLField*, an *EmailField* or any integer field type, Django will use the `url`, `email` and `number` HTML5 input types. By default, browsers may apply their own validation on these fields, which may be stricter than Django's validation. If you would like to disable this behavior, set the `novalidate` attribute on the `form` tag, or specify a different widget on the field, like *TextInput*.

We now have a working web form, described by a Django *Form*, processed by a view, and rendered as an HTML `<form>`.

That's all you need to get started, but the forms framework puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

3.4.5 More about Django `Form` classes

All form classes are created as subclasses of either *django.forms.Form* or *django.forms.ModelForm*. You can think of `ModelForm` as a subclass of `Form`. `Form` and `ModelForm` actually inherit common functionality from a (private) `BaseForm` class, but this implementation detail is rarely important.

Models and Forms

In fact if your form is going to be used to directly add or edit a Django model, a ModelForm can save you a great deal of time, effort, and code, because it will build a form, along with the appropriate fields and their attributes, from a `Model` class.

Bound and unbound form instances

The distinction between Bound and unbound forms is important:

- An unbound form has no data associated with it. When rendered to the user, it will be empty or will contain default values.
- A bound form has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

The form's *is_bound* attribute will tell you whether a form has data bound to it or not.

More on fields

Consider a more useful form than our minimal example above, which we could use to implement "contact me" functionality on a personal website:

Listing 12: `forms.py`

```
from django import forms


class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Our earlier form used a single field, `your_name`, a *`CharField`*. In this case, our form has four fields: `subject`, `message`, `sender` and `cc_myself`. *`CharField`*, *`EmailField`* and *`BooleanField`* are just three of the available field types; a full list can be found in Form fields.

Widgets

Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such as `<input type="text">`.

In most cases, the field will have a sensible default widget. For example, by default, a *`CharField`* will have a *`TextInput`* widget, that produces an `<input type="text">` in the HTML. If you needed `<textarea>` instead, you'd specify the appropriate widget when defining your form field, as we have done for the `message` field.

Field data

Whatever the data submitted with a form, once it has been successfully validated by calling `is_valid()` (and `is_valid()` has returned `True`), the validated form data will be in the `form.cleaned_data` dictionary. This data will have been nicely converted into Python types for you.

> **Note**
>
> You can still access the unvalidated data directly from `request.POST` at this point, but the validated data is better.

In the contact form example above, `cc_myself` will be a boolean value. Likewise, fields such as *`IntegerField`* and *`FloatField`* convert values to a Python `int` and `float` respectively.

Here's how the form data could be processed in the view that handles this form:

Listing 13: `views.py`

```
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data["subject"]
    message = form.cleaned_data["message"]
    sender = form.cleaned_data["sender"]
    cc_myself = form.cleaned_data["cc_myself"]

    recipients = ["info@example.com"]
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect("/thanks/")
```

> **Tip**
>
> For more on sending email from Django, see Sending email.

Some field types need some extra handling. For example, files that are uploaded using a form need to be handled differently (they can be retrieved from `request.FILES`, rather than `request.POST`). For details of how to handle file uploads with your form, see Binding uploaded files to a form.

3.4.6 Working with form templates

All you need to do to get your form into a template is to place the form instance into the template context. So if your form is called `form` in the context, `{{ form }}` will render its `<label>` and `<input>` elements appropriately.

> **Additional form template furniture**
>
> Don't forget that a form's output does not include the surrounding `<form>` tags, or the form's `submit` control. You will have to provide these yourself.

Reusable form templates

The HTML output when rendering a form is itself generated via a template. You can control this by creating an appropriate template file and setting a custom *FORM_RENDERER* to use that *form_template_name* site-wide. You can also customize per-form by overriding the form's *template_name* attribute to render the form using the custom template, or by passing the template name directly to *Form.render()*.

The example below will result in `{{ form }}` being rendered as the output of the `form_snippet.html` template.

In your templates:

```
# In your template:
{{ form }}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

Then you can configure the *FORM_RENDERER* setting:

Listing 14: `settings.py`

```
from django.forms.renderers import TemplatesSetting


class CustomFormRenderer(TemplatesSetting):
    form_template_name = "form_snippet.html"


FORM_RENDERER = "project.settings.CustomFormRenderer"
```

. . . or for a single form:

```
class MyForm(forms.Form):
    template_name = "form_snippet.html"
    ...
```

. . . or for a single render of a form instance, passing in the template name to the *Form.render()*. Here's an example of this being used in a view:

```
def index(request):
    form = MyForm()
    rendered_form = form.render("form_snippet.html")
    context = {"form": rendered_form}
    return render(request, "index.html", context)
```

See Outputting forms as HTML for more details.

Reusable field group templates

Each field is available as an attribute of the form, using `{{ form.name_of_field }}` in a template. A field has a *as_field_group()* method which renders the related elements of the field as a group, its label, widget, errors, and help text.

This allows generic templates to be written that arrange fields elements in the required layout. For example:

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
  {{ form.subject.as_field_group }}
</div>
<div class="fieldWrapper">
  {{ form.message.as_field_group }}
</div>
<div class="fieldWrapper">
  {{ form.sender.as_field_group }}
</div>
<div class="fieldWrapper">
  {{ form.cc_myself.as_field_group }}
</div>
```

By default Django uses the `"django/forms/field.html"` template which is designed for use with the default `"django/forms/div.html"` form style.

The default template can be customized by setting *field_template_name* in your project-level *FORM_RENDERER*:

```
from django.forms.renderers import TemplatesSetting


class CustomFormRenderer(TemplatesSetting):
    field_template_name = "field_snippet.html"
```

. . . or on a single field:

```
class MyForm(forms.Form):
    subject = forms.CharField(template_name="my_custom_template.html")
    ...
```

. . . or on a per-request basis by calling *BoundField.render()* and supplying a template name:

```
def index(request):
    form = ContactForm()
    subject = form["subject"]
    context = {"subject": subject.render("my_custom_template.html")}
    return render(request, "index.html", context)
```

Rendering fields manually

More fine grained control over field rendering is also possible. Likely this will be in a custom field template, to allow the template to be written once and reused for each field. However, it can also be directly accessed from the field attribute on the form. For example:

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

Complete `<label>` elements can also be generated using the *label_tag()*. For example:

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

Rendering form error messages

The price of this flexibility is a bit more work. Until now we haven't had to worry about how to display form errors, because that's taken care of for us. In this example we have had to make sure we take care of any errors for each field and any errors for the form as a whole. Note `{{ form.non_field_errors }}` at the top of the form and the template lookup for errors on each field.

Using `{{ form.name_of_field.errors }}` displays a list of form errors, rendered as an unordered list. This might look like:

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

The list has a CSS class of `errorlist` to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

Non-field errors (and/or hidden field errors that are rendered at the top of the form when using helpers like `form.as_p()`) will be rendered with an additional class of `nonfield` to help distinguish them from field-specific errors. For example, `{{ form.non_field_errors }}` would look like:

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

See The Forms API for more on errors, styling, and working with form attributes in templates.

Looping over the form's fields

If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a `{% for %}` loop:

```
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
        {% if field.help_text %}
          <p class="help" id="{{ field.auto_id }}_helptext">
            {{ field.help_text|safe }}
          </p>
        {% endif %}
    </div>
{% endfor %}
```

Useful attributes on `{{ field }}` include:

`{{ field.errors }}`
: Outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a string containing the error message.

`{{ field.field }}`
: The *`Field`* instance from the form class that this *`BoundField`* wraps. You can use it to access *`Field`* attributes, e.g. `{{ char_field.field.max_length }}`.

`{{ field.help_text }}`
: Any help text that has been associated with the field.

`{{ field.html_name }}`
: The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

`{{ field.id_for_label }}`
: The ID that will be used for this field (`id_email` in the example above). If you are constructing the label manually, you may want to use this in lieu of `label_tag`. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID.

`{{ field.is_hidden }}`
: This attribute is `True` if the form field is a hidden field and `False` otherwise. It's not particularly useful as a template variable, but could be useful in conditional tests such as:

```
{% if field.is_hidden %}
   {# Do something special #}
```

(continues on next page)

(continued from previous page)

```
{% endif %}
```

`{{ field.label }}`
: The label of the field, e.g. `Email address`.

`{{ field.label_tag }}`
: The field's label wrapped in the appropriate HTML `<label>` tag. This includes the form's *`label_suffix`*. For example, the default `label_suffix` is a colon:

```
<label for="id_email">Email address:</label>
```

`{{ field.legend_tag }}`
: Similar to `field.label_tag` but uses a `<legend>` tag in place of `<label>`, for widgets with multiple inputs wrapped in a `<fieldset>`.

`{{ field.use_fieldset }}`
: This attribute is `True` if the form field's widget contains multiple inputs that should be semantically grouped in a `<fieldset>` with a `<legend>` to improve accessibility. An example use in a template:

```
{% if field.use_fieldset %}
  <fieldset>
  {% if field.label %}{{ field.legend_tag }}{% endif %}
{% else %}
  {% if field.label %}{{ field.label_tag }}{% endif %}
{% endif %}
{{ field }}
{% if field.use_fieldset %}</fieldset>{% endif %}
```

`{{ field.value }}`
: The value of the field. e.g `someone@example.com`.

See also

For a complete list of attributes and methods, see *`BoundField`*.

Looping over hidden and visible fields

If you're manually laying out a form in a template, as opposed to relying on Django's default form layout, you might want to treat `<input type="hidden">` fields differently from non-hidden fields. For example, because hidden fields don't display anything, putting error messages "next to" the field could cause confusion for your users – so errors for those fields should be handled differently.

Django provides two methods on a form that allow you to loop over the hidden and visible fields indepen-

dently: `hidden_fields()` and `visible_fields()`. Here's a modification of an earlier example that uses these two methods:

```
{# Include the hidden fields #}
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}
{# Include the visible fields #}
{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

This example does not handle any errors in the hidden fields. Usually, an error in a hidden field is a sign of form tampering, since normal form interaction won't alter them. However, you could easily insert some error displays for those form errors, as well.

3.4.7 Further topics

This covers the basics, but forms can do a whole lot more:

Formsets

class BaseFormSet

A formset is a layer of abstraction to work with multiple forms on the same page. It can be best compared to a data grid. Let's say you have the following form:

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
...
```

You might want to allow the user to create several articles at once. To create a formset out of an `ArticleForm` you would do:

```
>>> from django.forms import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

You now have created a formset class named `ArticleFormSet`. Instantiating the formset gives you the ability to iterate over the forms in the formset and display them as you would with a regular form:

```
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-title">Title:</label><input type="text" name="form-0-title"␣
↪id="id_form-0-title"></div>
<div><label for="id_form-0-pub_date">Pub date:</label><input type="text" name="form-0-
↪pub_date" id="id_form-0-pub_date"></div>
```

As you can see it only displayed one empty form. The number of empty forms that is displayed is controlled by the `extra` parameter. By default, *`formset_factory()`* defines one extra form; the following example will create a formset class to display two blank forms:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

Formsets can be iterated and indexed, accessing forms in the order they were created. You can reorder the forms by overriding the default `iteration` and `indexing` behavior if needed.

Using initial data with a formset

Initial data is what drives the main usability of a formset. As shown above you can define the number of extra forms. What this means is that you are telling the formset how many additional forms to show in addition to the number of forms it generates from the initial data. Let's take a look at an example:

```
>>> import datetime
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
>>> formset = ArticleFormSet(
...     initial=[
...         {
...             "title": "Django is now open source",
...             "pub_date": datetime.date.today(),
...         }
...     ]
... )

>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-title">Title:</label><input type="text" name="form-0-title"␣
↪value="Django is now open source" id="id_form-0-title"></div>
```

(continues on next page)

(continued from previous page)

```
<div><label for="id_form-0-pub_date">Pub date:</label><input type="text" name="form-0-
↪pub_date" value="2023-02-11" id="id_form-0-pub_date"></div>
<div><label for="id_form-1-title">Title:</label><input type="text" name="form-1-title"␣
↪id="id_form-1-title"></div>
<div><label for="id_form-1-pub_date">Pub date:</label><input type="text" name="form-1-
↪pub_date" id="id_form-1-pub_date"></div>
<div><label for="id_form-2-title">Title:</label><input type="text" name="form-2-title"␣
↪id="id_form-2-title"></div>
<div><label for="id_form-2-pub_date">Pub date:</label><input type="text" name="form-2-
↪pub_date" id="id_form-2-pub_date"></div>
```

There are now a total of three forms showing above. One for the initial data that was passed in and two extra forms. Also note that we are passing in a list of dictionaries as the initial data.

If you use an `initial` for displaying a formset, you should pass the same `initial` when processing that formset's submission so that the formset can detect which forms were changed by the user. For example, you might have something like: `ArticleFormSet(request.POST, initial=[...])`.

See also

Creating formsets from models with model formsets.

Limiting the maximum number of forms

The `max_num` parameter to *`formset_factory()`* gives you the ability to limit the number of forms the formset will display:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-title">Title:</label><input type="text" name="form-0-title"␣
↪id="id_form-0-title"></div>
<div><label for="id_form-0-pub_date">Pub date:</label><input type="text" name="form-0-
↪pub_date" id="id_form-0-pub_date"></div>
```

If the value of `max_num` is greater than the number of existing items in the initial data, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `max_num`. For example, if `extra=2` and `max_num=2` and the formset is initialized with one `initial` item, a form for the initial

item and one blank form will be displayed.

If the number of items in the initial data exceeds `max_num`, all initial data forms will be displayed regardless of the value of `max_num` and no extra forms will be displayed. For example, if `extra=3` and `max_num=1` and the formset is initialized with two initial items, two forms with the initial data will be displayed.

A `max_num` value of `None` (the default) puts a high limit on the number of forms displayed (1000). In practice this is equivalent to no limit.

By default, `max_num` only affects how many forms are displayed and does not affect validation. If `validate_max=True` is passed to the *`formset_factory()`*, then `max_num` will affect validation. See validate_max.

Limiting the maximum number of instantiated forms

The `absolute_max` parameter to *`formset_factory()`* allows limiting the number of forms that can be instantiated when supplying `POST` data. This protects against memory exhaustion attacks using forged `POST` requests:

```
>>> from django.forms.formsets import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, absolute_max=1500)
>>> data = {
...     "form-TOTAL_FORMS": "1501",
...     "form-INITIAL_FORMS": "0",
... }
>>> formset = ArticleFormSet(data)
>>> len(formset.forms)
1500
>>> formset.is_valid()
False
>>> formset.non_form_errors()
['Please submit at most 1000 forms.']
```

When `absolute_max` is `None`, it defaults to `max_num + 1000`. (If `max_num` is `None`, it defaults to 2000).

If `absolute_max` is less than `max_num`, a `ValueError` will be raised.

Formset validation

Validation with a formset is almost identical to a regular `Form`. There is an `is_valid` method on the formset to provide a convenient way to validate all forms in the formset:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
```

(continues on next page)

(continued from previous page)

```
>>> ArticleFormSet = formset_factory(ArticleForm)
>>> data = {
...     "form-TOTAL_FORMS": "1",
...     "form-INITIAL_FORMS": "0",
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
True
```

We passed in no data to the formset which is resulting in a valid form. The formset is smart enough to ignore extra forms that were not changed. If we provide an invalid article:

```
>>> data = {
...     "form-TOTAL_FORMS": "2",
...     "form-INITIAL_FORMS": "0",
...     "form-0-title": "Test",
...     "form-0-pub_date": "1904-06-16",
...     "form-1-title": "Test",
...     "form-1-pub_date": "",  # <-- this date is missing but required
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
```

As we can see, `formset.errors` is a list whose entries correspond to the forms in the formset. Validation was performed for each of the two forms, and the expected error message appears for the second item.

Just like when using a normal `Form`, each field in a formset's forms may include HTML attributes such as `maxlength` for browser validation. However, form fields of formsets won't include the `required` attribute as that validation may be incorrect when adding and deleting forms.

`BaseFormSet.total_error_count()`

To check how many errors there are in the formset, we can use the `total_error_count` method:

```
>>> # Using the previous example
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
>>> len(formset.errors)
2
```

(continues on next page)

(continued from previous page)

```
>>> formset.total_error_count()
1
```

We can also check if form data differs from the initial data (i.e. the form was sent without any data):

```
>>> data = {
...     "form-TOTAL_FORMS": "1",
...     "form-INITIAL_FORMS": "0",
...     "form-0-title": "",
...     "form-0-pub_date": "",
... }
>>> formset = ArticleFormSet(data)
>>> formset.has_changed()
False
```

Understanding the `ManagementForm`

You may have noticed the additional data (`form-TOTAL_FORMS`, `form-INITIAL_FORMS`) that was required in the formset's data above. This data is required for the `ManagementForm`. This form is used by the formset to manage the collection of forms contained in the formset. If you don't provide this management data, the formset will be invalid:

```
>>> data = {
...     "form-0-title": "Test",
...     "form-0-pub_date": "",
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
```

It is used to keep track of how many form instances are being displayed. If you are adding new forms via JavaScript, you should increment the count fields in this form as well. On the other hand, if you are using JavaScript to allow deletion of existing objects, then you need to ensure the ones being removed are properly marked for deletion by including `form-#-DELETE` in the `POST` data. It is expected that all forms are present in the `POST` data regardless.

The management form is available as an attribute of the formset itself. When rendering a formset in a template, you can include all the management data by rendering `{{ my_formset.management_form }}` (substituting the name of your formset as appropriate).

> **Note**
>
> As well as the `form-TOTAL_FORMS` and `form-INITIAL_FORMS` fields shown in the examples here, the management form also includes `form-MIN_NUM_FORMS` and `form-MAX_NUM_FORMS` fields. They are output with the rest of the management form, but only for the convenience of client-side code. These fields are not required and so are not shown in the example `POST` data.

`total_form_count` and `initial_form_count`

`BaseFormSet` has a couple of methods that are closely related to the `ManagementForm`, `total_form_count` and `initial_form_count`.

`total_form_count` returns the total number of forms in this formset. `initial_form_count` returns the number of forms in the formset that were pre-filled, and is also used to determine how many forms are required. You will probably never need to override either of these methods, so please be sure you understand what they do before doing so.

`empty_form`

`BaseFormSet` provides an additional attribute `empty_form` which returns a form instance with a prefix of `__prefix__` for easier use in dynamic forms with JavaScript.

`error_messages`

The `error_messages` argument lets you override the default messages that the formset will raise. Pass in a dictionary with keys matching the error messages you want to override. Error message keys include `'too_few_forms'`, `'too_many_forms'`, and `'missing_management_form'`. The `'too_few_forms'` and `'too_many_forms'` error messages may contain `%(num)d`, which will be replaced with `min_num` and `max_num`, respectively.

For example, here is the default error message when the management form is missing:

```
>>> formset = ArticleFormSet({})
>>> formset.is_valid()
False
>>> formset.non_form_errors()
['ManagementForm data is missing or has been tampered with. Missing fields: form-TOTAL_
↪FORMS, form-INITIAL_FORMS. You may need to file a bug report if the issue persists.']
```

And here is a custom error message:

```
>>> formset = ArticleFormSet(
...     {}, error_messages={"missing_management_form": "Sorry, something went wrong."}
... )
```

(continues on next page)

(continued from previous page)

```
>>> formset.is_valid()
False
>>> formset.non_form_errors()
['Sorry, something went wrong.']
```

Custom formset validation

A formset has a `clean` method similar to the one on a `Form` class. This is where you define your own validation that works at the formset level:

```
>>> from django.core.exceptions import ValidationError
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):
...     def clean(self):
...         """Checks that no two articles have the same title."""
...         if any(self.errors):
...             # Don't bother validating the formset unless each form is valid on its␣
↪own
...             return
...         titles = set()
...         for form in self.forms:
...             if self.can_delete and self._should_delete_form(form):
...                 continue
...             title = form.cleaned_data.get("title")
...             if title in titles:
...                 raise ValidationError("Articles in a set must have distinct titles.")
...             titles.add(title)
...

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> data = {
...     "form-TOTAL_FORMS": "2",
...     "form-INITIAL_FORMS": "0",
...     "form-0-title": "Test",
...     "form-0-pub_date": "1904-06-16",
...     "form-1-title": "Test",
...     "form-1-pub_date": "1912-06-23",
```

(continues on next page)

(continued from previous page)

```
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Articles in a set must have distinct titles.']
```

The formset `clean` method is called after all the `Form.clean` methods have been called. The errors will be found using the `non_form_errors()` method on the formset.

Non-form errors will be rendered with an additional class of `nonform` to help distinguish them from form-specific errors. For example, `{{ formset.non_form_errors }}` would look like:

```
<ul class="errorlist nonform">
    <li>Articles in a set must have distinct titles.</li>
</ul>
```

Validating the number of forms in a formset

Django provides a couple ways to validate the minimum or maximum number of submitted forms. Applications which need more customizable validation of the number of forms should use custom formset validation.

validate_max

If `validate_max=True` is passed to *formset_factory()*, validation will also check that the number of forms in the data set, minus those marked for deletion, is less than or equal to `max_num`.

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, max_num=1, validate_max=True)
>>> data = {
...     "form-TOTAL_FORMS": "2",
...     "form-INITIAL_FORMS": "0",
...     "form-0-title": "Test",
...     "form-0-pub_date": "1904-06-16",
...     "form-1-title": "Test 2",
...     "form-1-pub_date": "1912-06-23",
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
```

(continues on next page)

(continued from previous page)

```
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Please submit at most 1 form.']
```

`validate_max=True` validates against `max_num` strictly even if `max_num` was exceeded because the amount of initial data supplied was excessive.

The error message can be customized by passing the `'too_many_forms'` message to the error_messages argument.

> **Note**
>
> Regardless of `validate_max`, if the number of forms in a data set exceeds `absolute_max`, then the form will fail to validate as if `validate_max` were set, and additionally only the first `absolute_max` forms will be validated. The remainder will be truncated entirely. This is to protect against memory exhaustion attacks using forged POST requests. See Limiting the maximum number of instantiated forms.

validate_min

If `validate_min=True` is passed to *formset_factory()*, validation will also check that the number of forms in the data set, minus those marked for deletion, is greater than or equal to `min_num`.

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, min_num=3, validate_min=True)
>>> data = {
...     "form-TOTAL_FORMS": "2",
...     "form-INITIAL_FORMS": "0",
...     "form-0-title": "Test",
...     "form-0-pub_date": "1904-06-16",
...     "form-1-title": "Test 2",
...     "form-1-pub_date": "1912-06-23",
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
```

(continues on next page)

(continued from previous page)

```
['Please submit at least 3 forms.']
```

The error message can be customized by passing the `'too_few_forms'` message to the error_messages argument.

> **Note**
>
> Regardless of `validate_min`, if a formset contains no data, then `extra + min_num` empty forms will be displayed.

Dealing with ordering and deletion of forms

The *`formset_factory()`* provides two optional parameters `can_order` and `can_delete` to help with ordering of forms in formsets and deletion of forms from a formset.

`can_order`

`BaseFormSet.can_order`

Default: `False`

Lets you create a formset with the ability to order:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_order=True)
>>> formset = ArticleFormSet(
...     initial=[
...         {"title": "Article #1", "pub_date": datetime.date(2008, 5, 10)},
...         {"title": "Article #2", "pub_date": datetime.date(2008, 5, 11)},
...     ]
... )
>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-title">Title:</label><input type="text" name="form-0-title"␣
↪value="Article #1" id="id_form-0-title"></div>
<div><label for="id_form-0-pub_date">Pub date:</label><input type="text" name="form-0-
↪pub_date" value="2008-05-10" id="id_form-0-pub_date"></div>
<div><label for="id_form-0-ORDER">Order:</label><input type="number" name="form-0-ORDER"␣
↪value="1" id="id_form-0-ORDER"></div>
<div><label for="id_form-1-title">Title:</label><input type="text" name="form-1-title"␣
```

(continues on next page)

(continued from previous page)

```
↪value="Article #2" id="id_form-1-title"></div>
<div><label for="id_form-1-pub_date">Pub date:</label><input type="text" name="form-1-
↪pub_date" value="2008-05-11" id="id_form-1-pub_date"></div>
<div><label for="id_form-1-ORDER">Order:</label><input type="number" name="form-1-ORDER"␣
↪value="2" id="id_form-1-ORDER"></div>
<div><label for="id_form-2-title">Title:</label><input type="text" name="form-2-title"␣
↪id="id_form-2-title"></div>
<div><label for="id_form-2-pub_date">Pub date:</label><input type="text" name="form-2-
↪pub_date" id="id_form-2-pub_date"></div>
<div><label for="id_form-2-ORDER">Order:</label><input type="number" name="form-2-ORDER"␣
↪id="id_form-2-ORDER"></div>
```

This adds an additional field to each form. This new field is named `ORDER` and is an `forms.IntegerField`. For the forms that came from the initial data it automatically assigned them a numeric value. Let's look at what will happen when the user changes these values:

```
>>> data = {
...     "form-TOTAL_FORMS": "3",
...     "form-INITIAL_FORMS": "2",
...     "form-0-title": "Article #1",
...     "form-0-pub_date": "2008-05-10",
...     "form-0-ORDER": "2",
...     "form-1-title": "Article #2",
...     "form-1-pub_date": "2008-05-11",
...     "form-1-ORDER": "1",
...     "form-2-title": "Article #3",
...     "form-2-pub_date": "2008-05-01",
...     "form-2-ORDER": "0",
... }

>>> formset = ArticleFormSet(
...     data,
...     initial=[
...         {"title": "Article #1", "pub_date": datetime.date(2008, 5, 10)},
...         {"title": "Article #2", "pub_date": datetime.date(2008, 5, 11)},
...     ],
... )
>>> for form in formset.ordered_forms:
...     print(form.cleaned_data)
...
```

(continues on next page)

(continued from previous page)

```
{'title': 'Article #3', 'pub_date': datetime.date(2008, 5, 1), 'ORDER': 0}
{'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11), 'ORDER': 1}
{'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10), 'ORDER': 2}
```

BaseFormSet also provides an *ordering_widget* attribute and *get_ordering_widget()* method that control the widget used with *can_order*.

ordering_widget

BaseFormSet.**ordering_widget**

Default: *NumberInput*

Set `ordering_widget` to specify the widget class to be used with `can_order`:

```
>>> from django.forms import BaseFormSet, formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     ordering_widget = HiddenInput
...

>>> ArticleFormSet = formset_factory(
...     ArticleForm, formset=BaseArticleFormSet, can_order=True
... )
```

get_ordering_widget

BaseFormSet.**get_ordering_widget**()

Override `get_ordering_widget()` if you need to provide a widget instance for use with `can_order`:

```
>>> from django.forms import BaseFormSet, formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     def get_ordering_widget(self):
...         return HiddenInput(attrs={"class": "ordering"})
...

>>> ArticleFormSet = formset_factory(
...     ArticleForm, formset=BaseArticleFormSet, can_order=True
... )
```

can_delete

BaseFormSet.can_delete

Default: `False`

Lets you create a formset with the ability to select forms for deletion:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_delete=True)
>>> formset = ArticleFormSet(
...     initial=[
...         {"title": "Article #1", "pub_date": datetime.date(2008, 5, 10)},
...         {"title": "Article #2", "pub_date": datetime.date(2008, 5, 11)},
...     ]
... )
>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-title">Title:</label><input type="text" name="form-0-title"␣
↪value="Article #1" id="id_form-0-title"></div>
<div><label for="id_form-0-pub_date">Pub date:</label><input type="text" name="form-0-
↪pub_date" value="2008-05-10" id="id_form-0-pub_date"></div>
<div><label for="id_form-0-DELETE">Delete:</label><input type="checkbox" name="form-0-
↪DELETE" id="id_form-0-DELETE"></div>
<div><label for="id_form-1-title">Title:</label><input type="text" name="form-1-title"␣
↪value="Article #2" id="id_form-1-title"></div>
<div><label for="id_form-1-pub_date">Pub date:</label><input type="text" name="form-1-
↪pub_date" value="2008-05-11" id="id_form-1-pub_date"></div>
<div><label for="id_form-1-DELETE">Delete:</label><input type="checkbox" name="form-1-
↪DELETE" id="id_form-1-DELETE"></div>
<div><label for="id_form-2-title">Title:</label><input type="text" name="form-2-title"␣
↪id="id_form-2-title"></div>
<div><label for="id_form-2-pub_date">Pub date:</label><input type="text" name="form-2-
↪pub_date" id="id_form-2-pub_date"></div>
<div><label for="id_form-2-DELETE">Delete:</label><input type="checkbox" name="form-2-
↪DELETE" id="id_form-2-DELETE"></div>
```

Similar to `can_order` this adds a new field to each form named `DELETE` and is a `forms.BooleanField`. When data comes through marking any of the delete fields you can access them with `deleted_forms`:

```
>>> data = {
...     "form-TOTAL_FORMS": "3",
...     "form-INITIAL_FORMS": "2",
...     "form-0-title": "Article #1",
...     "form-0-pub_date": "2008-05-10",
...     "form-0-DELETE": "on",
...     "form-1-title": "Article #2",
...     "form-1-pub_date": "2008-05-11",
...     "form-1-DELETE": "",
...     "form-2-title": "",
...     "form-2-pub_date": "",
...     "form-2-DELETE": "",
... }

>>> formset = ArticleFormSet(
...     data,
...     initial=[
...         {"title": "Article #1", "pub_date": datetime.date(2008, 5, 10)},
...         {"title": "Article #2", "pub_date": datetime.date(2008, 5, 11)},
...     ],
... )
>>> [form.cleaned_data for form in formset.deleted_forms]
[{'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10), 'DELETE': True}]
```

If you are using a *ModelFormSet*, model instances for deleted forms will be deleted when you call `formset.save()`.

If you call `formset.save(commit=False)`, objects will not be deleted automatically. You'll need to call `delete()` on each of the *formset.deleted_objects* to actually delete them:

```
>>> instances = formset.save(commit=False)
>>> for obj in formset.deleted_objects:
...     obj.delete()
...
```

On the other hand, if you are using a plain `FormSet`, it's up to you to handle `formset.deleted_forms`, perhaps in your formset's `save()` method, as there's no general notion of what it means to delete a form.

BaseFormSet also provides a *deletion_widget* attribute and *get_deletion_widget()* method that control the widget used with *can_delete*.

deletion_widget

BaseFormSet.**deletion_widget**

Default: *CheckboxInput*

Set `deletion_widget` to specify the widget class to be used with `can_delete`:

```
>>> from django.forms import BaseFormSet, formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     deletion_widget = HiddenInput
...

>>> ArticleFormSet = formset_factory(
...     ArticleForm, formset=BaseArticleFormSet, can_delete=True
... )
```

get_deletion_widget

BaseFormSet.**get_deletion_widget**()

Override `get_deletion_widget()` if you need to provide a widget instance for use with `can_delete`:

```
>>> from django.forms import BaseFormSet, formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     def get_deletion_widget(self):
...         return HiddenInput(attrs={"class": "deletion"})
...

>>> ArticleFormSet = formset_factory(
...     ArticleForm, formset=BaseArticleFormSet, can_delete=True
... )
```

can_delete_extra

BaseFormSet.**can_delete_extra**

Default: `True`

While setting `can_delete=True`, specifying `can_delete_extra=False` will remove the option to delete extra forms.

Adding additional fields to a formset

If you need to add additional fields to the formset this can be easily accomplished. The formset base class provides an `add_fields` method. You can override this method to add your own fields or even redefine the default fields/attributes of the order and deletion fields:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     def add_fields(self, form, index):
...         super().add_fields(form, index)
...         form.fields["my_field"] = forms.CharField()
...

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-title">Title:</label><input type="text" name="form-0-title"␣
↪id="id_form-0-title"></div>
<div><label for="id_form-0-pub_date">Pub date:</label><input type="text" name="form-0-
↪pub_date" id="id_form-0-pub_date"></div>
<div><label for="id_form-0-my_field">My field:</label><input type="text" name="form-0-my_
↪field" id="id_form-0-my_field"></div>
```

Passing custom parameters to formset forms

Sometimes your form class takes custom parameters, like `MyArticleForm`. You can pass this parameter when instantiating the formset:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm

>>> class MyArticleForm(ArticleForm):
...     def __init__(self, *args, user, **kwargs):
...         self.user = user
...         super().__init__(*args, **kwargs)
...
```

(continues on next page)

(continued from previous page)

```
>>> ArticleFormSet = formset_factory(MyArticleForm)
>>> formset = ArticleFormSet(form_kwargs={"user": request.user})
```

The `form_kwargs` may also depend on the specific form instance. The formset base class provides a `get_form_kwargs` method. The method takes a single argument - the index of the form in the formset. The index is `None` for the empty_form:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory

>>> class BaseArticleFormSet(BaseFormSet):
...     def get_form_kwargs(self, index):
...         kwargs = super().get_form_kwargs(index)
...         kwargs["custom_kwarg"] = index
...         return kwargs
...

>>> ArticleFormSet = formset_factory(MyArticleForm, formset=BaseArticleFormSet)
>>> formset = ArticleFormSet()
```

Customizing a formset's prefix

In the rendered HTML, formsets include a prefix on each field's name. By default, the prefix is `'form'`, but it can be customized using the formset's `prefix` argument.

For example, in the default case, you might see:

```
<label for="id_form-0-title">Title:</label>
<input type="text" name="form-0-title" id="id_form-0-title">
```

But with `ArticleFormset(prefix='article')` that becomes:

```
<label for="id_article-0-title">Title:</label>
<input type="text" name="article-0-title" id="id_article-0-title">
```

This is useful if you want to use more than one formset in a view.

Using a formset in views and templates

Formsets have the following attributes and methods associated with rendering:

`BaseFormSet.renderer`

Specifies the renderer to use for the formset. Defaults to the renderer specified by the *FORM_RENDERER*

setting.

`BaseFormSet.template_name`

The name of the template rendered if the formset is cast into a string, e.g. via `print(formset)` or in a template via `{{ formset }}`.

By default, a property returning the value of the renderer's *formset_template_name*. You may set it as a string template name in order to override that for a particular formset class.

This template will be used to render the formset's management form, and then each form in the formset as per the template defined by the form's *template_name*.

`BaseFormSet.template_name_div`

The name of the template used when calling *as_div()*. By default this is `"django/forms/formsets/div.html"`. This template renders the formset's management form and then each form in the formset as per the form's *as_div()* method.

`BaseFormSet.template_name_p`

The name of the template used when calling *as_p()*. By default this is `"django/forms/formsets/p.html"`. This template renders the formset's management form and then each form in the formset as per the form's *as_p()* method.

`BaseFormSet.template_name_table`

The name of the template used when calling *as_table()*. By default this is `"django/forms/formsets/table.html"`. This template renders the formset's management form and then each form in the formset as per the form's *as_table()* method.

`BaseFormSet.template_name_ul`

The name of the template used when calling *as_ul()*. By default this is `"django/forms/formsets/ul.html"`. This template renders the formset's management form and then each form in the formset as per the form's *as_ul()* method.

`BaseFormSet.get_context()`

Returns the context for rendering a formset in a template.

The available context is:

- `formset` : The instance of the formset.

`BaseFormSet.render`(template_name=None, context=None, renderer=None)

The render method is called by `__str__` as well as the *as_div()*, *as_p()*, *as_ul()*, and *as_table()* methods. All arguments are optional and will default to:

- `template_name`: *template_name*
- `context`: Value returned by *get_context()*
- `renderer`: Value returned by *renderer*

`BaseFormSet.as_div()`

Renders the formset with the *template_name_div* template.

`BaseFormSet.as_p()`

Renders the formset with the *template_name_p* template.

`BaseFormSet.as_table()`

Renders the formset with the *template_name_table* template.

`BaseFormSet.as_ul()`

Renders the formset with the *template_name_ul* template.

Using a formset inside a view is not very different from using a regular `Form` class. The only thing you will want to be aware of is making sure to use the management form inside the template. Let's look at a sample view:

```
from django.forms import formset_factory
from django.shortcuts import render
from myapp.forms import ArticleForm


def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == "POST":
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
            pass
    else:
        formset = ArticleFormSet()
    return render(request, "manage_articles.html", {"formset": formset})
```

The `manage_articles.html` template might look like this:

```
<form method="post">
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
        {{ form }}
        {% endfor %}
    </table>
</form>
```

However there's a slight shortcut for the above by letting the formset itself deal with the management form:

```
<form method="post">
    <table>
        {{ formset }}
    </table>
</form>
```

The above ends up calling the *BaseFormSet.render()* method on the formset class. This renders the formset using the template specified by the *template_name* attribute. Similar to forms, by default the formset will be rendered `as_div`, with other helper methods of `as_p`, `as_ul`, and `as_table` being available. The rendering of the formset can be customized by specifying the `template_name` attribute, or more generally by overriding the default template.

Manually rendered `can_delete` and `can_order`

If you manually render fields in the template, you can render `can_delete` parameter with `{{ form.DELETE }}`:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        <ul>
            <li>{{ form.title }}</li>
            <li>{{ form.pub_date }}</li>
            {% if formset.can_delete %}
                <li>{{ form.DELETE }}</li>
            {% endif %}
        </ul>
    {% endfor %}
</form>
```

Similarly, if the formset has the ability to order (`can_order=True`), it is possible to render it with `{{ form.ORDER }}`.

Using more than one formset in a view

You are able to use more than one formset in a view if you like. Formsets borrow much of its behavior from forms. With that said you are able to use `prefix` to prefix formset form field names with a given value to allow more than one formset to be sent to a view without name clashing. Let's take a look at how this might be accomplished:

```
from django.forms import formset_factory
from django.shortcuts import render
```

(continues on next page)

(continued from previous page)

```
from myapp.forms import ArticleForm, BookForm


def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    BookFormSet = formset_factory(BookForm)
    if request.method == "POST":
        article_formset = ArticleFormSet(request.POST, request.FILES, prefix="articles")
        book_formset = BookFormSet(request.POST, request.FILES, prefix="books")
        if article_formset.is_valid() and book_formset.is_valid():
            # do something with the cleaned_data on the formsets.
            pass
    else:
        article_formset = ArticleFormSet(prefix="articles")
        book_formset = BookFormSet(prefix="books")
    return render(
        request,
        "manage_articles.html",
        {
            "article_formset": article_formset,
            "book_formset": book_formset,
        },
    )
```

You would then render the formsets as normal. It is important to point out that you need to pass `prefix` on both the POST and non-POST cases so that it is rendered and processed correctly.

Each formset's prefix replaces the default `form` prefix that's added to each field's `name` and `id` HTML attributes.

Creating forms from models

`ModelForm`

`class ModelForm`

If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a `BlogComment` model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a `Form` class from a Django model.

For example:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ["pub_date", "headline", "content", "reporter"]
...

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Field types

The generated `Form` class will have a form field for every model field specified, in the order specified in the `fields` attribute.

Each model field has a corresponding default form field. For example, a `CharField` on a model is represented as a `CharField` on a form. A model `ManyToManyField` is represented as a `MultipleChoiceField`. Here is the full list of conversions:

Model field	Form field
AutoField	Not represented in the form
BigAutoField	Not represented in the form
BigIntegerField	*IntegerField* with `min_value` set to -9223372036854775808 and `max_value` set to 922337
BinaryField	*CharField*, if *editable* is set to `True` on the model field, otherwise not represented in the
BooleanField	*BooleanField*, or *NullBooleanField* if `null=True`.
CharField	*CharField* with `max_length` set to the model field's `max_length` and *empty_value* set to N
DateField	*DateField*
DateTimeField	*DateTimeField*
DecimalField	*DecimalField*
DurationField	*DurationField*
EmailField	*EmailField*
FileField	*FileField*
FilePathField	*FilePathField*

cont

Table 1 – continued from previous page

Model field	Form field
FloatField	*FloatField*
ForeignKey	*ModelChoiceField* (see below)
ImageField	*ImageField*
IntegerField	*IntegerField*
IPAddressField	IPAddressField
GenericIPAddressField	*GenericIPAddressField*
JSONField	*JSONField*
ManyToManyField	*ModelMultipleChoiceField* (see below)
PositiveBigIntegerField	*IntegerField*
PositiveIntegerField	*IntegerField*
PositiveSmallIntegerField	*IntegerField*
SlugField	*SlugField*
SmallAutoField	Not represented in the form
SmallIntegerField	*IntegerField*
TextField	*CharField* with `widget=forms.Textarea`
TimeField	*TimeField*
URLField	*URLField*
UUIDField	*UUIDField*

As you might expect, the `ForeignKey` and `ManyToManyField` model field types are special cases:

- `ForeignKey` is represented by `django.forms.ModelChoiceField`, which is a `ChoiceField` whose choices are a model `QuerySet`.
- `ManyToManyField` is represented by `django.forms.ModelMultipleChoiceField`, which is a `MultipleChoiceField` whose choices are a model `QuerySet`.

In addition, each generated form field has attributes set as follows:

- If the model field has `blank=True`, then `required` is set to `False` on the form field. Otherwise, `required=True`.
- The form field's `label` is set to the `verbose_name` of the model field, with the first character capitalized.
- The form field's `help_text` is set to the `help_text` of the model field.
- If the model field has `choices` set, then the form field's `widget` will be set to `Select`, with choices coming from the model field's `choices`. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has `blank=False` and an explicit `default` value (the `default` value will be initially selected instead).

Finally, note that you can override the form field used for a given model field. See Overriding the default fields below.

A full example

Consider this set of models:

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = {
    "MR": "Mr.",
    "MRS": "Mrs.",
    "MS": "Ms.",
}


class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name


class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]


class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ["name", "authors"]
```

With these models, the `ModelForm` subclasses above would be roughly equivalent to this (the only difference being the `save()` method, which we'll discuss in a moment.):

```
from django import forms


class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(
        max_length=3,
        widget=forms.Select(choices=TITLE_CHOICES),
    )
    birth_date = forms.DateField(required=False)


class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

Validation on a `ModelForm`

There are two main steps involved in validating a `ModelForm`:

1. Validating the form
2. Validating the model instance

Just like normal form validation, model form validation is triggered implicitly when calling *`is_valid()`* or accessing the *`errors`* attribute and explicitly when calling `full_clean()`, although you will typically not use the latter method in practice.

`Model` validation (*`Model.full_clean()`*) is triggered from within the form validation step, right after the form's `clean()` method is called.

> **Warning**
>
> The cleaning process modifies the model instance passed to the `ModelForm` constructor in various ways. For instance, any date fields on the model are converted into actual date objects. Failed validation may leave the underlying model instance in an inconsistent state and therefore it's not recommended to reuse it.

Overriding the `clean()` method

You can override the `clean()` method on a model form to provide additional validation in the same way you can on a normal form.

A model form instance attached to a model object will contain an `instance` attribute that gives its methods

access to that specific model instance.

> **Warning**
>
> The `ModelForm.clean()` method sets a flag that makes the model validation step validate the uniqueness of model fields that are marked as `unique`, `unique_together` or `unique_for_date|month|year`.
>
> If you would like to override the `clean()` method and maintain this validation, you must call the parent class's `clean()` method.

Interaction with model validation

As part of the validation process, `ModelForm` will call the `clean()` method of each field on your model that has a corresponding field on your form. If you have excluded any model fields, validation will not be run on those fields. See the form validation documentation for more on how field cleaning and validation work.

The model's `clean()` method will be called before any uniqueness checks are made. See Validating objects for more information on the model's `clean()` hook.

Considerations regarding model's `error_messages`

Error messages defined at the *`form field`* level or at the form Meta level always take precedence over the error messages defined at the *`model field`* level.

Error messages defined on *`model fields`* are only used when the `ValidationError` is raised during the model validation step and no corresponding error messages are defined at the form level.

You can override the error messages from `NON_FIELD_ERRORS` raised by model validation by adding the *`NON_FIELD_ERRORS`* key to the `error_messages` dictionary of the `ModelForm`'s inner `Meta` class:

```
from django.core.exceptions import NON_FIELD_ERRORS
from django.forms import ModelForm


class ArticleForm(ModelForm):
    class Meta:
        error_messages = {
            NON_FIELD_ERRORS: {
                "unique_together": "%(model_name)s's %(field_labels)s are not unique.",
            }
        }
```

The `save()` method

Every `ModelForm` also has a `save()` method. This method creates and saves a database object from the data bound to the form. A subclass of `ModelForm` can accept an existing model instance as the keyword argument `instance`; if this is supplied, `save()` will update that instance. If it's not supplied, `save()` will create a new instance of the specified model:

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Note that if the form hasn't been validated, calling `save()` will do so by checking `form.errors`. A `ValueError` will be raised if the data in the form doesn't validate – i.e., if `form.errors` evaluates to `True`.

If an optional field doesn't appear in the form's data, the resulting model instance uses the model field *`default`*, if there is one, for that field. This behavior doesn't apply to fields that use *`CheckboxInput`*, *`CheckboxSelectMultiple`*, or *`SelectMultiple`* (or any custom widget whose *`value_omitted_from_data()`* method always returns `False`) since an unchecked checkbox and unselected `<select multiple>` don't appear in the data of an HTML form submission. Use a custom form field or widget if you're designing an API and want the default fallback behavior for a field that uses one of these widgets.

This `save()` method accepts an optional `commit` keyword argument, which accepts either `True` or `False`. If you call `save()` with `commit=False`, then it will return an object that hasn't yet been saved to the database. In this case, it's up to you to call `save()` on the resulting model instance. This is useful if you want to do custom processing on the object before saving it, or if you want to use one of the specialized model saving options. `commit` is `True` by default.

Another side effect of using `commit=False` is seen when your model has a many-to-many relation with another model. If your model has a many-to-many relation and you specify `commit=False` when you save a form, Django cannot immediately save the form data for the many-to-many relation. This is because it isn't possible to save many-to-many data for an instance until the instance exists in the database.

To work around this problem, every time you save a form using `commit=False`, Django adds a `save_m2m()` method to your `ModelForm` subclass. After you've manually saved the instance produced by the form, you

can invoke `save_m2m()` to save the many-to-many form data. For example:

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = "some_value"

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

Calling `save_m2m()` is only required if you use `save(commit=False)`. When you use a `save()` on a form, all data – including many-to-many data – is saved without the need for any additional method calls. For example:

```
# Create a form instance with POST data.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do anything else.
>>> new_author = f.save()
```

Other than the `save()` and `save_m2m()` methods, a `ModelForm` works exactly the same way as any other `forms` form. For example, the `is_valid()` method is used to check for validity, the `is_multipart()` method is used to determine whether a form requires multipart file upload (and hence whether `request.FILES` must be passed to the form), etc. See Binding uploaded files to a form for more information.

Selecting the fields to use

It is strongly recommended that you explicitly set all fields that should be edited in the form using the `fields` attribute. Failure to do so can easily lead to security problems when a form unexpectedly allows a user to set certain fields, especially when new fields are added to a model. Depending on how the form is rendered, the problem may not even be visible on the web page.

The alternative approach would be to include all fields automatically, or remove only some. This fundamental approach is known to be much less secure and has led to serious exploits on major websites (e.g. GitHub).

There are, however, two shortcuts available for cases where you can guarantee these security concerns do

not apply to you:

1. Set the `fields` attribute to the special value `'__all__'` to indicate that all fields in the model should be used. For example:

```
from django.forms import ModelForm


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = "__all__"
```

2. Set the `exclude` attribute of the `ModelForm`'s inner `Meta` class to a list of fields to be excluded from the form.

   For example:

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ["title"]
```

   Since the `Author` model has the 3 fields `name`, `title` and `birth_date`, this will result in the fields `name` and `birth_date` being present on the form.

If either of these are used, the order the fields appear in the form will be the order the fields are defined in the model, with `ManyToManyField` instances appearing last.

In addition, Django applies the following rule: if you set `editable=False` on the model field, any form created from the model via `ModelForm` will not include that field.

Note

Any fields not included in a form by the above logic will not be set by the form's `save()` method. Also, if you manually add the excluded fields back to the form, they will not be initialized from the model instance.

Django will prevent any attempt to save an incomplete model, so if the model does not allow the missing fields to be empty, and does not provide a default value for the missing fields, any attempt to `save()` a `ModelForm` with missing fields will fail. To avoid this failure, you must instantiate your model with initial values for the missing, but required fields:

```
author = Author(title="Mr")
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

Alternatively, you can use `save(commit=False)` and manually set any extra required fields:

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = "Mr"
author.save()
```

See the section on saving forms for more details on using `save(commit=False)`.

Overriding the default fields

The default field types, as described in the Field types table above, are sensible defaults. If you have a `DateField` in your model, chances are you'd want that to be represented as a `DateField` in your form. But `ModelForm` gives you the flexibility of changing the form field for a given model.

To specify a custom widget for a field, use the `widgets` attribute of the inner `Meta` class. This should be a dictionary mapping field names to widget classes or instances.

For example, if you want the `CharField` for the `name` attribute of `Author` to be represented by a `<textarea>` instead of its default `<input type="text">`, you can override the field's widget:

```
from django.forms import ModelForm, Textarea
from myapp.models import Author


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]
        widgets = {
            "name": Textarea(attrs={"cols": 80, "rows": 20}),
        }
```

The `widgets` dictionary accepts either widget instances (e.g., `Textarea(...)`) or classes (e.g., `Textarea`). Note that the `widgets` dictionary is ignored for a model field with a non-empty `choices` attribute. In this case, you must override the form field to use a different widget.

Similarly, you can specify the `labels`, `help_texts` and `error_messages` attributes of the inner `Meta` class if you want to further customize a field.

For example if you wanted to customize the wording of all user facing strings for the `name` field:

```
from django.utils.translation import gettext_lazy as _
```

(continues on next page)

(continued from previous page)

```
class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]
        labels = {
            "name": _("Writer"),
        }
        help_texts = {
            "name": _("Some useful help text."),
        }
        error_messages = {
            "name": {
                "max_length": _("This writer's name is too long."),
            },
        }
```

You can also specify `field_classes` or `formfield_callback` to customize the type of fields instantiated by the form.

For example, if you wanted to use `MySlugFormField` for the `slug` field, you could do the following:

```
from django.forms import ModelForm
from myapp.models import Article


class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ["pub_date", "headline", "content", "reporter", "slug"]
        field_classes = {
            "slug": MySlugFormField,
        }
```

or:

```
from django.forms import ModelForm
from myapp.models import Article


def formfield_for_dbfield(db_field, **kwargs):
    if db_field.name == "slug":
```

(continues on next page)

(continued from previous page)

```
        return MySlugFormField()
    return db_field.formfield(**kwargs)


class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ["pub_date", "headline", "content", "reporter", "slug"]
        formfield_callback = formfield_for_dbfield
```

Finally, if you want complete control over of a field – including its type, validators, required, etc. – you can do this by declaratively specifying fields like you would in a regular `Form`.

If you want to specify a field's validators, you can do so by defining the field declaratively and setting its `validators` parameter:

```
from django.forms import CharField, ModelForm
from myapp.models import Article


class ArticleForm(ModelForm):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = Article
        fields = ["pub_date", "headline", "content", "reporter", "slug"]
```

Note

When you explicitly instantiate a form field like this, it is important to understand how `ModelForm` and regular `Form` are related.

`ModelForm` is a regular `Form` which can automatically generate certain fields. The fields that are automatically generated depend on the content of the `Meta` class and on which fields have already been defined declaratively. Basically, `ModelForm` will only generate fields that are missing from the form, or in other words, fields that weren't defined declaratively.

Fields defined declaratively are left as-is, therefore any customizations made to `Meta` attributes such as `widgets`, `labels`, `help_texts`, or `error_messages` are ignored; these only apply to fields that are generated automatically.

Similarly, fields defined declaratively do not draw their attributes like `max_length` or `required` from the corresponding model. If you want to maintain the behavior specified in the model, you must set the

relevant arguments explicitly when declaring the form field.

For example, if the `Article` model looks like this:

```
class Article(models.Model):
    headline = models.CharField(
        max_length=200,
        null=True,
        blank=True,
        help_text="Use puns liberally",
    )
    content = models.TextField()
```

and you want to do some custom validation for `headline`, while keeping the `blank` and `help_text` values as specified, you might define `ArticleForm` like this:

```
class ArticleForm(ModelForm):
    headline = MyFormField(
        max_length=200,
        required=False,
        help_text="Use puns liberally",
    )

    class Meta:
        model = Article
        fields = ["headline", "content"]
```

You must ensure that the type of the form field can be used to set the contents of the corresponding model field. When they are not compatible, you will get a `ValueError` as no implicit conversion takes place.

See the form field documentation for more information on fields and their arguments.

Enabling localization of fields

By default, the fields in a `ModelForm` will not localize their data. To enable localization for fields, you can use the `localized_fields` attribute on the `Meta` class.

```
>>> from django.forms import ModelForm
>>> from myapp.models import Author
>>> class AuthorForm(ModelForm):
...     class Meta:
...         model = Author
...         localized_fields = ['birth_date']
```

If `localized_fields` is set to the special value `'__all__'`, all fields will be localized.

Form inheritance

As with basic forms, you can extend and reuse `ModelForm` classes by inheriting them. This is useful if you need to declare extra fields or extra methods on a parent class for use in a number of forms derived from models. For example, using the previous `ArticleForm` class:

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self): ...
...
```

This creates a form that behaves identically to `ArticleForm`, except there's some extra validation and cleaning for the `pub_date` field.

You can also subclass the parent's `Meta` inner class if you want to change the `Meta.fields` or `Meta.exclude` lists:

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ["body"]
...
```

This adds the extra method from the `EnhancedArticleForm` and modifies the original `ArticleForm.Meta` to remove one field.

There are a couple of things to note, however.

- Normal Python name resolution rules apply. If you have multiple base classes that declare a `Meta` inner class, only the first one will be used. This means the child's `Meta`, if it exists, otherwise the `Meta` of the first parent, etc.
- It's possible to inherit from both `Form` and `ModelForm` simultaneously, however, you must ensure that `ModelForm` appears first in the MRO. This is because these classes rely on different metaclasses and a class can only have one metaclass.
- It's possible to declaratively remove a `Field` inherited from a parent class by setting the name to be `None` on the subclass.

  You can only use this technique to opt out from a field defined declaratively by a parent class; it won't prevent the `ModelForm` metaclass from generating a default field. To opt-out from default fields, see Selecting the fields to use.

Providing initial values

As with regular forms, it's possible to specify initial data for forms by specifying an `initial` parameter when instantiating the form. Initial values provided this way will override both initial values from the form field and values from an attached model instance. For example:

```
>>> article = Article.objects.get(pk=1)
>>> article.headline
'My headline'
>>> form = ArticleForm(initial={"headline": "Initial headline"}, instance=article)
>>> form["headline"].value()
'Initial headline'
```

ModelForm factory function

You can create forms from a given model using the standalone function *modelform_factory()*, instead of using a class definition. This may be more convenient if you do not have many customizations to make:

```
>>> from django.forms import modelform_factory
>>> from myapp.models import Book
>>> BookForm = modelform_factory(Book, fields=["author", "title"])
```

This can also be used to make modifications to existing forms, for example by specifying the widgets to be used for a given field:

```
>>> from django.forms import Textarea
>>> Form = modelform_factory(Book, form=BookForm, widgets={"title": Textarea()})
```

The fields to include can be specified using the `fields` and `exclude` keyword arguments, or the corresponding attributes on the `ModelForm` inner `Meta` class. Please see the `ModelForm` Selecting the fields to use documentation.

. . . or enable localization for specific fields:

```
>>> Form = modelform_factory(Author, form=AuthorForm, localized_fields=["birth_date"])
```

Model formsets

class models.BaseModelFormSet

Like regular formsets, Django provides a couple of enhanced formset classes to make working with Django models more convenient. Let's reuse the `Author` model from above:

```
>>> from django.forms import modelformset_factory
>>> from myapp.models import Author
>>> AuthorFormSet = modelformset_factory(Author, fields=["name", "title"])
```

Using `fields` restricts the formset to use only the given fields. Alternatively, you can take an "opt-out" approach, specifying which fields to exclude:

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=["birth_date"])
```

This will create a formset that is capable of working with the data associated with the `Author` model. It works just like a regular formset:

```
>>> formset = AuthorFormSet()
>>> print(formset)
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS"><input␣
↪type="hidden" name="form-INITIAL_FORMS" value="0" id="id_form-INITIAL_FORMS"><input␣
↪type="hidden" name="form-MIN_NUM_FORMS" value="0" id="id_form-MIN_NUM_FORMS"><input␣
↪type="hidden" name="form-MAX_NUM_FORMS" value="1000" id="id_form-MAX_NUM_FORMS">
<div><label for="id_form-0-name">Name:</label><input id="id_form-0-name" type="text"␣
↪name="form-0-name" maxlength="100"></div>
<div><label for="id_form-0-title">Title:</label><select name="form-0-title" id="id_form-
↪0-title">
<option value="" selected>---------</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select><input type="hidden" name="form-0-id" id="id_form-0-id"></div>
```

> **Note**
>
> *`modelformset_factory()`* uses *`formset_factory()`* to generate formsets. This means that a model formset is an extension of a basic formset that knows how to interact with a particular model.

> **Note**
>
> When using multi-table inheritance, forms generated by a formset factory will contain a parent link field (by default `<parent_model_name>_ptr`) instead of an `id` field.

Changing the queryset

By default, when you create a formset from a model, the formset will use a queryset that includes all objects in the model (e.g., `Author.objects.all()`). You can override this behavior by using the `queryset` argument:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith="O"))
```

Alternatively, you can create a subclass that sets `self.queryset` in `__init__`:

```
from django.forms import BaseModelFormSet
from myapp.models import Author


class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.queryset = Author.objects.filter(name__startswith="O")
```

Then, pass your `BaseAuthorFormSet` class to the factory function:

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=["name", "title"], formset=BaseAuthorFormSet
... )
```

If you want to return a formset that doesn't include any preexisting instances of the model, you can specify an empty QuerySet:

```
>>> AuthorFormSet(queryset=Author.objects.none())
```

Changing the form

By default, when you use `modelformset_factory`, a model form will be created using *`modelform_factory()`*. Often, it can be useful to specify a custom model form. For example, you can create a custom model form that has custom validation:

```
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title"]

    def clean_name(self):
        # custom validation for the name field
        ...
```

Then, pass your model form to the factory function:

```
AuthorFormSet = modelformset_factory(Author, form=AuthorForm)
```

It is not always necessary to define a custom model form. The `modelformset_factory` function has several arguments which are passed through to `modelform_factory`, which are described below.

Specifying widgets to use in the form with `widgets`

Using the `widgets` parameter, you can specify a dictionary of values to customize the `ModelForm`'s widget class for a particular field. This works the same way as the `widgets` dictionary on the inner `Meta` class of a `ModelForm` works:

```
>>> AuthorFormSet = modelformset_factory(
...     Author,
...     fields=["name", "title"],
...     widgets={"name": Textarea(attrs={"cols": 80, "rows": 20})},
... )
```

Enabling localization for fields with `localized_fields`

Using the `localized_fields` parameter, you can enable localization for fields in the form.

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=['name', 'title', 'birth_date'],
...     localized_fields=['birth_date'])
```

If `localized_fields` is set to the special value `'__all__'`, all fields will be localized.

Providing initial values

As with regular formsets, it's possible to specify initial data for forms in the formset by specifying an `initial` parameter when instantiating the model formset class returned by *`modelformset_factory()`*. However, with model formsets, the initial values only apply to extra forms, those that aren't attached to an existing model instance. If the length of `initial` exceeds the number of extra forms, the excess initial data is ignored. If the extra forms with initial data aren't changed by the user, they won't be validated or saved.

Saving objects in the formset

As with a `ModelForm`, you can save the data as a model object. This is done with the formset's `save()` method:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

The `save()` method returns the instances that have been saved to the database. If a given instance's data didn't change in the bound data, the instance won't be saved to the database and won't be included in the return value (`instances`, in the above example).

When fields are missing from the form (for example because they have been excluded), these fields will not be set by the `save()` method. You can find more information about this restriction, which also holds for regular model forms, in Selecting the fields to use.

Pass `commit=False` to return the unsaved model instances:

```
# don't save to the database
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # do something with instance
...     instance.save()
...
```

This gives you the ability to attach data to the instances before saving them to the database. If your formset contains a `ManyToManyField`, you'll also need to call `formset.save_m2m()` to ensure the many-to-many relationships are saved properly.

After calling `save()`, your model formset will have three new attributes containing the formset's changes:

`models.BaseModelFormSet.changed_objects`

`models.BaseModelFormSet.deleted_objects`

`models.BaseModelFormSet.new_objects`

Limiting the number of editable objects

As with regular formsets, you can use the `max_num` and `extra` parameters to *`modelformset_factory()`* to limit the number of extra forms displayed.

`max_num` does not prevent existing objects from being displayed:

```
>>> Author.objects.order_by("name")
<QuerySet [<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>
↪]>

>>> AuthorFormSet = modelformset_factory(Author, fields=["name"], max_num=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by("name"))
>>> [x.name for x in formset.get_queryset()]
['Charles Baudelaire', 'Paul Verlaine', 'Walt Whitman']
```

Also, `extra=0` doesn't prevent creation of new model instances as you can add additional forms with JavaScript or send additional POST data. See Preventing new objects creation on how to do this.

If the value of `max_num` is greater than the number of existing related objects, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `max_num`:

```
>>> AuthorFormSet = modelformset_factory(Author, fields=["name"], max_num=4, extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by("name"))
>>> for form in formset:
...     print(form)
...
<div><label for="id_form-0-name">Name:</label><input id="id_form-0-name" type="text"␣
↪name="form-0-name" value="Charles Baudelaire" maxlength="100"><input type="hidden"␣
↪name="form-0-id" value="1" id="id_form-0-id"></div>
<div><label for="id_form-1-name">Name:</label><input id="id_form-1-name" type="text"␣
↪name="form-1-name" value="Paul Verlaine" maxlength="100"><input type="hidden" name=
↪"form-1-id" value="3" id="id_form-1-id"></div>
<div><label for="id_form-2-name">Name:</label><input id="id_form-2-name" type="text"␣
↪name="form-2-name" value="Walt Whitman" maxlength="100"><input type="hidden" name=
↪"form-2-id" value="2" id="id_form-2-id"></div>
<div><label for="id_form-3-name">Name:</label><input id="id_form-3-name" type="text"␣
↪name="form-3-name" maxlength="100"><input type="hidden" name="form-3-id" id="id_form-3-
↪id"></div>
```

A `max_num` value of `None` (the default) puts a high limit on the number of forms displayed (1000). In practice this is equivalent to no limit.

Preventing new objects creation

Using the `edit_only` parameter, you can prevent creation of any new objects:

```
>>> AuthorFormSet = modelformset_factory(
...     Author,
...     fields=["name", "title"],
...     edit_only=True,
... )
```

Here, the formset will only edit existing `Author` instances. No other objects will be created or edited.

Using a model formset in a view

Model formsets are very similar to formsets. Let's say we want to present a formset to edit `Author` model instances:

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author
```

(continues on next page)

(continued from previous page)

```
def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=["name", "title"])
    if request.method == "POST":
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = AuthorFormSet()
    return render(request, "manage_authors.html", {"formset": formset})
```

As you can see, the view logic of a model formset isn't drastically different than that of a "normal" formset. The only difference is that we call `formset.save()` to save the data into the database. (This was described above, in Saving objects in the formset.)

Overriding `clean()` on a `ModelFormSet`

Just like with a `ModelForm`, by default the `clean()` method of a `ModelFormSet` will validate that none of the items in the formset violate the unique constraints on your model (either `unique`, `unique_together` or `unique_for_date|month|year`). If you want to override the `clean()` method on a `ModelFormSet` and maintain this validation, you must call the parent class's `clean` method:

```
from django.forms import BaseModelFormSet


class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super().clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

Also note that by the time you reach this step, individual model instances have already been created for each `Form`. Modifying a value in `form.cleaned_data` is not sufficient to affect the saved value. If you wish to modify a value in `ModelFormSet.clean()` you must modify `form.instance`:

```
from django.forms import BaseModelFormSet
```

(continues on next page)

(continued from previous page)

```
class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super().clean()

        for form in self.forms:
            name = form.cleaned_data["name"].upper()
            form.cleaned_data["name"] = name
            # update the instance value.
            form.instance.name = name
```

Using a custom queryset

As stated earlier, you can override the default queryset used by the model formset:

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author


def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=["name", "title"])
    queryset = Author.objects.filter(name__startswith="O")
    if request.method == "POST":
        formset = AuthorFormSet(
            request.POST,
            request.FILES,
            queryset=queryset,
        )
        if formset.is_valid():
            formset.save()
            # Do something.
    else:
        formset = AuthorFormSet(queryset=queryset)
    return render(request, "manage_authors.html", {"formset": formset})
```

Note that we pass the `queryset` argument in both the `POST` and `GET` cases in this example.

Using the formset in the template

There are three ways to render a formset in a Django template.

First, you can let the formset do most of the work:

```
<form method="post">
    {{ formset }}
</form>
```

Second, you can manually render the formset, but let the form deal with itself:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>
```

When you manually render the forms yourself, be sure to render the management form as shown above. See the management form documentation.

Third, you can manually render each field:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }} {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

If you opt to use this third method and you don't iterate over the fields with a `{% for %}` loop, you'll need to render the primary key field. For example, if you were rendering the `name` and `age` fields of a model:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form.id }}
        <ul>
            <li>{{ form.name }}</li>
            <li>{{ form.age }}</li>
        </ul>
```

(continues on next page)

(continued from previous page)

```
    {% endfor %}
</form>
```

Notice how we need to explicitly render `{{ form.id }}`. This ensures that the model formset, in the `POST` case, will work correctly. (This example assumes a primary key named `id`. If you've explicitly defined your own primary key that isn't called `id`, make sure it gets rendered.)

Inline formsets

class models.**BaseInlineFormSet**

Inline formsets is a small abstraction layer on top of model formsets. These simplify the case of working with related objects via a foreign key. Suppose you have these two models:

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=100)


class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
```

If you want to create a formset that allows you to edit books belonging to a particular author, you could do this:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=["title"])
>>> author = Author.objects.get(name="Mike Royko")
>>> formset = BookFormSet(instance=author)
```

`BookFormSet`'s prefix is `'book_set'` (`<model name>_set`). If `Book`'s `ForeignKey` to `Author` has a *`related_name`*, that's used instead.

> **Note**
>
> *`inlineformset_factory()`* uses *`modelformset_factory()`* and marks `can_delete=True`.

See also

Manually rendered can_delete and can_order.

Overriding methods on an `InlineFormSet`

When overriding methods on `InlineFormSet`, you should subclass *`BaseInlineFormSet`* rather than *`BaseModelFormSet`*.

For example, if you want to override `clean()`:

```
from django.forms import BaseInlineFormSet


class CustomInlineFormSet(BaseInlineFormSet):
    def clean(self):
        super().clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

See also Overriding clean() on a ModelFormSet.

Then when you create your inline formset, pass in the optional argument `formset`:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(
...     Author, Book, fields=["title"], formset=CustomInlineFormSet
... )
>>> author = Author.objects.get(name="Mike Royko")
>>> formset = BookFormSet(instance=author)
```

More than one foreign key to the same model

If your model contains more than one foreign key to the same model, you'll need to resolve the ambiguity manually using `fk_name`. For example, consider the following model:

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name="from_friends",
```

(continues on next page)

(continued from previous page)

```
    )
    to_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name="friends",
    )
    length_in_months = models.IntegerField()
```

To resolve this, you can use `fk_name` to *`inlineformset_factory()`*:

```
>>> FriendshipFormSet = inlineformset_factory(
...     Friend, Friendship, fk_name="from_friend", fields=["to_friend", "length_in_months
↪"]
... )
```

Using an inline formset in a view

You may want to provide a view that allows a user to edit the related objects of a model. Here's how you can do that:

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book, fields=["title"])
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something. Should generally end with a redirect. For example:
            return HttpResponseRedirect(author.get_absolute_url())
    else:
        formset = BookInlineFormSet(instance=author)
    return render(request, "manage_books.html", {"formset": formset})
```

Notice how we pass `instance` in both the `POST` and `GET` cases.

Specifying widgets to use in the inline form

`inlineformset_factory` uses `modelformset_factory` and passes most of its arguments to `modelformset_factory`. This means you can use the `widgets` parameter in much the same way as passing it to `modelformset_factory`. See Specifying widgets to use in the form with widgets above.

Form Assets (the `Media` class)

Rendering an attractive and easy-to-use web form requires more than just HTML - it also requires CSS stylesheets, and if you want to use fancy widgets, you may also need to include some JavaScript on each page. The exact combination of CSS and JavaScript that is required for any given page will depend upon the widgets that are in use on that page.

This is where asset definitions come in. Django allows you to associate different files – like stylesheets and scripts – with the forms and widgets that require those assets. For example, if you want to use a calendar to render DateFields, you can define a custom Calendar widget. This widget can then be associated with the CSS and JavaScript that is required to render the calendar. When the Calendar widget is used on a form, Django is able to identify the CSS and JavaScript files that are required, and provide the list of file names in a form suitable for inclusion on your web page.

> **Assets and Django Admin**
>
> The Django Admin application defines a number of customized widgets for calendars, filtered selections, and so on. These widgets define asset requirements, and the Django Admin uses the custom widgets in place of the Django defaults. The Admin templates will only include those files that are required to render the widgets on any given page.
>
> If you like the widgets that the Django Admin application uses, feel free to use them in your own application! They're all stored in `django.contrib.admin.widgets`.

> **Which JavaScript toolkit?**
>
> Many JavaScript toolkits exist, and many of them include widgets (such as calendar widgets) that can be used to enhance your application. Django has deliberately avoided blessing any one JavaScript toolkit. Each toolkit has its own relative strengths and weaknesses - use whichever toolkit suits your requirements. Django is able to integrate with any JavaScript toolkit.

Assets as a static definition

The easiest way to define assets is as a static definition. Using this method, the declaration is an inner `Media` class. The properties of the inner class define the requirements.

Here's an example:

```
from django import forms


class CalendarWidget(forms.TextInput):
    class Media:
```

(continues on next page)

(continued from previous page)

```
        css = {
            "all": ["pretty.css"],
        }
        js = ["animations.js", "actions.js"]
```

This code defines a `CalendarWidget`, which will be based on `TextInput`. Every time the CalendarWidget is used on a form, that form will be directed to include the CSS file `pretty.css`, and the JavaScript files `animations.js` and `actions.js`.

This static definition is converted at runtime into a widget property named `media`. The list of assets for a `CalendarWidget` instance can be retrieved through this property:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
```

Here's a list of all possible `Media` options. There are no required options.

css

A dictionary describing the CSS files required for various forms of output media.

The values in the dictionary should be a tuple/list of file names. See the section on paths for details of how to specify paths to these files.

The keys in the dictionary are the output media types. These are the same types accepted by CSS files in media declarations: 'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' and 'tv'. If you need to have different stylesheets for different media types, provide a list of CSS files for each output medium. The following example would provide two CSS options – one for the screen, and one for print:

```
class Media:
    css = {
        "screen": ["pretty.css"],
        "print": ["newspaper.css"],
    }
```

If a group of CSS files are appropriate for multiple output media types, the dictionary key can be a comma separated list of output media types. In the following example, TV's and projectors will have the same media requirements:

```
class Media:
    css = {
        "screen": ["pretty.css"],
        "tv,projector": ["lo_res.css"],
        "print": ["newspaper.css"],
    }
```

If this last CSS definition were to be rendered, it would become the following HTML:

```
<link href="https://static.example.com/pretty.css" media="screen" rel="stylesheet">
<link href="https://static.example.com/lo_res.css" media="tv,projector" rel="stylesheet">
<link href="https://static.example.com/newspaper.css" media="print" rel="stylesheet">
```

`js`

A tuple describing the required JavaScript files. See the section on paths for details of how to specify paths to these files.

`Script` objects

class `Script`(src, **attributes)

Represents a script file.

The first parameter, `src`, is the string path to the script file. See the section on paths for details on how to specify paths to these files.

The optional keyword arguments, `**attributes`, are HTML attributes that are set on the rendered `<script>` tag.

See Paths as objects for usage examples.

`extend`

A boolean defining inheritance behavior for `Media` declarations.

By default, any object using a static `Media` definition will inherit all the assets associated with the parent widget. This occurs regardless of how the parent defines its own requirements. For example, if we were to extend our basic Calendar widget from the example above:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             "all": ["fancy.css"],
...         }
```

(continues on next page)

(continued from previous page)

```
...         js = ["whizbang.js"]
...

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<link href="https://static.example.com/fancy.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

The FancyCalendar widget inherits all the assets from its parent widget. If you don't want `Media` to be inherited in this way, add an `extend=False` declaration to the `Media` declaration:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
...         css = {
...             "all": ["fancy.css"],
...         }
...         js = ["whizbang.js"]
...

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/fancy.css" media="all" rel="stylesheet">
<script src="https://static.example.com/whizbang.js"></script>
```

If you require even more control over inheritance, define your assets using a dynamic property. Dynamic properties give you complete control over which files are inherited, and which are not.

`Media` as a dynamic property

If you need to perform some more sophisticated manipulation of asset requirements, you can define the `media` property directly. This is done by defining a widget property that returns an instance of `forms.Media`. The constructor for `forms.Media` accepts `css` and `js` keyword arguments in the same format as that used in a static media definition.

For example, the static definition for our Calendar Widget could also be defined in a dynamic fashion:

```
class CalendarWidget(forms.TextInput):
    @property
```

(continues on next page)

(continued from previous page)

```
    def media(self):
        return forms.Media(
            css={"all": ["pretty.css"]}, js=["animations.js", "actions.js"]
        )
```

See the section on Media objects for more details on how to construct return values for dynamic `media` properties.

Paths in asset definitions

Paths as strings

String paths used to specify assets can be either relative or absolute. If a path starts with `/`, `http://` or `https://`, it will be interpreted as an absolute path, and left as-is. All other paths will be prepended with the value of the appropriate prefix. If the *`django.contrib.staticfiles`* app is installed, it will be used to serve assets.

Whether or not you use *`django.contrib.staticfiles`*, the *`STATIC_URL`* and *`STATIC_ROOT`* settings are required to render a complete web page.

To find the appropriate prefix to use, Django will check if the *`STATIC_URL`* setting is not `None` and automatically fall back to using *`MEDIA_URL`*. For example, if the *`MEDIA_URL`* for your site was `'https://uploads.example.com/'` and *`STATIC_URL`* was `None`:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             "all": ["/css/pretty.css"],
...         }
...         js = ["animations.js", "https://othersite.com/actions.js"]
...

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" media="all" rel="stylesheet">
<script src="https://uploads.example.com/animations.js"></script>
<script src="https://othersite.com/actions.js"></script>
```

But if *`STATIC_URL`* is `'https://static.example.com/'`:

```
>>> w = CalendarWidget()
>>> print(w.media)
```

(continues on next page)

(continued from previous page)

```
<link href="/css/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://othersite.com/actions.js"></script>
```

Or if *staticfiles* is configured using the *ManifestStaticFilesStorage*:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.27e20196a850.js"></script>
<script src="https://othersite.com/actions.js"></script>
```

Paths as objects

Assets may also be object-based, using *Script*. Furthermore, these allow you to pass custom HTML attributes:

```
class Media:
    js = [
        Script(
            "https://cdn.example.com/something.min.js",
            **{
                "crossorigin": "anonymous",
                "async": True,
            },
        ),
    ]
```

If this Media definition were to be rendered, it would become the following HTML:

```
<script src="https://cdn.example.com/something.min.js"
        crossorigin="anonymous"
        async>
</script>
```

The object class `Script` was added.

`Media` objects

When you interrogate the `media` attribute of a widget or form, the value that is returned is a `forms.Media` object. As we have already seen, the string representation of a `Media` object is the HTML required to include the relevant files in the `<head>` block of your HTML page.

However, `Media` objects have some other interesting properties.

Subsets of assets

If you only want files of a particular type, you can use the subscript operator to filter out a medium of interest. For example:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>

>>> print(w.media["css"])
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
```

When you use the subscript operator, the value that is returned is a new `Media` object – but one that only contains the media of interest.

Combining `Media` objects

`Media` objects can also be added together. When two `Media` objects are added, the resulting `Media` object contains the union of the assets specified by both:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             "all": ["pretty.css"],
...         }
...         js = ["animations.js", "actions.js"]
...

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ["whizbang.js"]
...
```

(continues on next page)

(continued from previous page)

```
>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

Order of assets

The order in which assets are inserted into the DOM is often important. For example, you may have a script that depends on jQuery. Therefore, combining `Media` objects attempts to preserve the relative order in which assets are defined in each `Media` class.

For example:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         js = ["jQuery.js", "calendar.js", "noConflict.js"]
...
>>> class TimeWidget(forms.TextInput):
...     class Media:
...         js = ["jQuery.js", "time.js", "noConflict.js"]
...
>>> w1 = CalendarWidget()
>>> w2 = TimeWidget()
>>> print(w1.media + w2.media)
<script src="https://static.example.com/jQuery.js"></script>
<script src="https://static.example.com/calendar.js"></script>
<script src="https://static.example.com/time.js"></script>
<script src="https://static.example.com/noConflict.js"></script>
```

Combining `Media` objects with assets in a conflicting order results in a `MediaOrderConflictWarning`.

`Media` on Forms

Widgets aren't the only objects that can have `media` definitions – forms can also define `media`. The rules for `media` definitions on forms are the same as the rules for widgets: declarations can be static or dynamic; path and inheritance rules for those declarations are exactly the same.

Regardless of whether you define a `media` declaration, all Form objects have a `media` property. The default value for this property is the result of adding the `media` definitions for all widgets that are part of the form:

```
>>> from django import forms
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...

>>> f = ContactForm()
>>> f.media
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

If you want to associate additional assets with a form – for example, CSS for form layout – add a `Media` declaration to the form:

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...     class Media:
...         css = {
...             "all": ["layout.css"],
...         }
...

>>> f = ContactForm()
>>> f.media
<link href="https://static.example.com/pretty.css" media="all" rel="stylesheet">
<link href="https://static.example.com/layout.css" media="all" rel="stylesheet">
<script src="https://static.example.com/animations.js"></script>
<script src="https://static.example.com/actions.js"></script>
<script src="https://static.example.com/whizbang.js"></script>
```

See also

The Forms Reference
: Covers the full API reference, including form fields, form widgets, and form and field validation.

3.5 Templates

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted. For a hands-on example of creating HTML pages with templates, see Tutorial 3.

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships built-in backends for its own template system, creatively called the Django template language (DTL), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties. You can also write your own custom backend, see Custom template backend

Django defines a standard API for loading and rendering templates regardless of the backend. Loading consists of finding the template for a given identifier and preprocessing it, usually compiling it to an in-memory representation. Rendering means interpolating the template with context data and returning the resulting string.

The Django template language is Django's own template system. Until Django 1.8 it was the only built-in option available. It's a good template library even though it's fairly opinionated and sports a few idiosyncrasies. If you don't have a pressing reason to choose another backend, you should use the DTL, especially if you're writing a pluggable application and you intend to distribute templates. Django's contrib apps that include templates, like django.contrib.admin, use the DTL.

For historical reasons, both the generic support for template engines and the implementation of the Django template language live in the `django.template` namespace.

> **Warning**
>
> The template system isn't safe against untrusted template authors. For example, a site shouldn't allow its users to provide their own templates, since template authors can do things like perform XSS attacks and access properties of template variables that may contain sensitive information.

3.5.1 The Django template language

Syntax

> **About this section**
>
> This is an overview of the Django template language's syntax. For details see the language syntax reference.

A Django template is a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags. Everything else is output as is.

The syntax of the Django template language involves four constructs.

Variables

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by `{{` and `}}` like this:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

With a context of `{'first_name': 'John', 'last_name': 'Doe'}`, this template renders to:

```
My first name is John. My last name is Doe.
```

Dictionary lookup, attribute lookup and list-index lookups are implemented with a dot notation:

```
{{ my_dict.key }}
{{ my_object.attribute }}
{{ my_list.0 }}
```

If a variable resolves to a callable, the template system will call it with no arguments and use its result instead of the callable.

Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by `{%` and `%}` like this:

```
{% csrf_token %}
```

Most tags accept arguments:

```
{% cycle 'odd' 'even' %}
```

Some tags require beginning and ending tags:

```
{% if user.is_authenticated %}Hello, {{ user.username }}.{% endif %}
```

A reference of built-in tags is available as well as instructions for writing custom tags.

Filters

Filters transform the values of variables and tag arguments.

They look like this:

```
{{ django|title }}
```

With a context of `{'django': 'the web framework for perfectionists with deadlines'}`, this template renders to:

```
The Web Framework For Perfectionists With Deadlines
```

Some filters take an argument:

```
{{ my_date|date:"Y-m-d" }}
```

A reference of built-in filters is available as well as instructions for writing custom filters.

Comments

Comments look like this:

```
{# this won't be rendered #}
```

A *{% comment %}* tag provides multi-line comments.

Components

> **About this section**
>
> This is an overview of the Django template language's APIs. For details see the API reference.

Engine

django.template.Engine encapsulates an instance of the Django template system. The main reason for instantiating an *Engine* directly is to use the Django template language outside of a Django project.

django.template.backends.django.DjangoTemplates is a thin wrapper adapting *django.template.Engine* to Django's template backend API.

Template

django.template.Template represents a compiled template. Templates are obtained with *Engine.get_template()* or *Engine.from_string()*.

Likewise `django.template.backends.django.Template` is a thin wrapper adapting *django.template.Template* to the common template API.

Context

django.template.Context holds some metadata in addition to the context data. It is passed to *Template.render()* for rendering a template.

django.template.RequestContext is a subclass of *Context* that stores the current *HttpRequest* and runs template context processors.

The common API doesn't have an equivalent concept. Context data is passed in a plain `dict` and the current *HttpRequest* is passed separately if needed.

Loaders

Template loaders are responsible for locating templates, loading them, and returning *Template* objects.

Django provides several built-in template loaders and supports custom template loaders.

Context processors

Context processors are functions that receive the current *HttpRequest* as an argument and return a `dict` of data to be added to the rendering context.

Their main use is to add common data shared by all templates to the context without repeating code in every view.

Django provides many built-in context processors, and you can implement your own additional context processors, too.

3.5.2 Support for template engines

Configuration

Templates engines are configured with the *TEMPLATES* setting. It's a list of configurations, one for each engine. The default value is empty. The `settings.py` generated by the *startproject* command defines a more useful value:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
```

(continues on next page)

(continued from previous page)

```
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            # ... some options here ...
        },
    },
]
```

BACKEND is a dotted Python path to a template engine class implementing Django's template backend API. The built-in backends are *django.template.backends.django.DjangoTemplates* and *django.template.backends.jinja2.Jinja2*.

Since most engines load templates from files, the top-level configuration for each engine contains two common settings:

- *DIRS* defines a list of directories where the engine should look for template source files, in search order.
- *APP_DIRS* tells whether the engine should look for templates inside installed applications. Each backend defines a conventional name for the subdirectory inside applications where its templates should be stored.

While uncommon, it's possible to configure several instances of the same backend with different options. In that case you should define a unique *NAME* for each engine.

OPTIONS contains backend-specific settings.

Usage

The `django.template.loader` module defines two functions to load templates.

`get_template`(template_name, using=None)

This function loads the template with the given name and returns a `Template` object.

The exact type of the return value depends on the backend that loaded the template. Each backend has its own `Template` class.

`get_template()` tries each template engine in order until one succeeds. If the template cannot be found, it raises *TemplateDoesNotExist*. If the template is found but contains invalid syntax, it raises *TemplateSyntaxError*.

How templates are searched and loaded depends on each engine's backend and configuration.

If you want to restrict the search to a particular template engine, pass the engine's *NAME* in the `using` argument.

`select_template`(template_name_list, using=None)

`select_template()` is just like `get_template()`, except it takes a list of template names. It tries each name in order and returns the first template that exists.

If loading a template fails, the following two exceptions, defined in `django.template`, may be raised:

exception `TemplateDoesNotExist`(msg, tried=None, backend=None, chain=None)

This exception is raised when a template cannot be found. It accepts the following optional arguments for populating the template postmortem on the debug page:

`backend`

The template backend instance from which the exception originated.

`tried`

A list of sources that were tried when finding the template. This is formatted as a list of tuples containing `(origin, status)`, where `origin` is an origin-like object and `status` is a string with the reason the template wasn't found.

`chain`

A list of intermediate *`TemplateDoesNotExist`* exceptions raised when trying to load a template. This is used by functions, such as *`get_template()`*, that try to load a given template from multiple engines.

exception `TemplateSyntaxError`(msg)

This exception is raised when a template was found but contains errors.

`Template` objects returned by `get_template()` and `select_template()` must provide a `render()` method with the following signature:

`Template.render`(context=None, request=None)

Renders this template with a given context.

If `context` is provided, it must be a `dict`. If it isn't provided, the engine will render the template with an empty context.

If `request` is provided, it must be an *`HttpRequest`*. Then the engine must make it, as well as the CSRF token, available in the template. How this is achieved is up to each backend.

Here's an example of the search algorithm. For this example the *`TEMPLATES`* setting is:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [
            "/home/html/example.com",
            "/home/html/default",
        ],
    },
    {
        "BACKEND": "django.template.backends.jinja2.Jinja2",
        "DIRS": [
```

(continues on next page)

(continued from previous page)

```
            "/home/html/jinja2",
        ],
    },
]
```

If you call `get_template('story_detail.html')`, here are the files Django will look for, in order:

- `/home/html/example.com/story_detail.html` (`'django'` engine)
- `/home/html/default/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_detail.html` (`'jinja2'` engine)

If you call `select_template(['story_253_detail.html', 'story_detail.html'])`, here's what Django will look for:

- `/home/html/example.com/story_253_detail.html` (`'django'` engine)
- `/home/html/default/story_253_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_253_detail.html` (`'jinja2'` engine)
- `/home/html/example.com/story_detail.html` (`'django'` engine)
- `/home/html/default/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_detail.html` (`'jinja2'` engine)

When Django finds a template that exists, it stops looking.

> **Use `django.template.loader.select_template()` for more flexibility**
>
> You can use *`select_template()`* for flexible template loading. For example, if you've written a news story and want some stories to have custom templates, use something like `select_template(['story_%s_detail.html' % story.id, 'story_detail.html'])`. That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

It's possible – and preferable – to organize templates in subdirectories inside each directory containing templates. The convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, use a slash, like so:

```
get_template("news/story_detail.html")
```

Using the same *`TEMPLATES`* option as above, this will attempt to load the following templates:

- `/home/html/example.com/news/story_detail.html` (`'django'` engine)
- `/home/html/default/news/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/news/story_detail.html` (`'jinja2'` engine)

In addition, to cut down on the repetitive nature of loading and rendering templates, Django provides a shortcut function which automates the process.

render_to_string(template_name, context=None, request=None, using=None)

`render_to_string()` loads a template like *get_template()* and calls its `render()` method immediately. It takes the following arguments.

template_name

The name of the template to load and render. If it's a list of template names, Django uses *select_template()* instead of *get_template()* to find the template.

context

A `dict` to be used as the template's context for rendering.

request

An optional *HttpRequest* that will be available during the template's rendering process.

using

An optional template engine *NAME*. The search for the template will be restricted to that engine.

Usage example:

```
from django.template.loader import render_to_string

rendered = render_to_string("my_template.html", {"foo": "bar"})
```

See also the *render()* shortcut which calls *render_to_string()* and feeds the result into an *HttpResponse* suitable for returning from a view.

Finally, you can use configured engines directly:

engines

Template engines are available in `django.template.engines`:

```
from django.template import engines

django_engine = engines["django"]
template = django_engine.from_string("Hello {{ name }}!")
```

The lookup key — `'django'` in this example — is the engine's *NAME*.

Built-in backends

`class DjangoTemplates`

Set *BACKEND* to `'django.template.backends.django.DjangoTemplates'` to configure a Django template engine.

When *APP_DIRS* is `True`, `DjangoTemplates` engines look for templates in the `templates` subdirectory of installed applications. This generic name was kept for backwards-compatibility.

`DjangoTemplates` engines accept the following *OPTIONS*:

- `'autoescape'`: a boolean that controls whether HTML autoescaping is enabled.

  It defaults to `True`.

  > **Warning**
  >
  > Only set it to `False` if you're rendering non-HTML templates!

- `'context_processors'`: a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a `dict` of items to be merged into the context.

  It defaults to an empty list.

  See *RequestContext* for more information.

- `'debug'`: a boolean that turns on/off template debug mode. If it is `True`, the fancy error page will display a detailed report for any exception raised during template rendering. This report contains the relevant snippet of the template with the appropriate line highlighted.

  It defaults to the value of the *DEBUG* setting.

- `'loaders'`: a list of dotted Python paths to template loader classes. Each `Loader` class knows how to import templates from a particular source. Optionally, a tuple can be used instead of a string. The first item in the tuple should be the `Loader` class name, and subsequent items are passed to the `Loader` during initialization.

  The default depends on the values of *DIRS* and *APP_DIRS*.

  See Loader types for details.

- `'string_if_invalid'`: the output, as a string, that the template system should use for invalid (e.g. misspelled) variables.

  It defaults to an empty string.

  See How invalid variables are handled for details.

- `'file_charset'`: the charset used to read template files on disk.

It defaults to `'utf-8'`.

- `'libraries'`: A dictionary of labels and dotted Python paths of template tag modules to register with the template engine. This can be used to add new libraries or provide alternate labels for existing ones. For example:

```
OPTIONS = {
    "libraries": {
        "myapp_tags": "path.to.myapp.tags",
        "admin.urls": "django.contrib.admin.templatetags.admin_urls",
    },
}
```

Libraries can be loaded by passing the corresponding dictionary key to the *{% load %}* tag.

- `'builtins'`: A list of dotted Python paths of template tag modules to add to built-ins. For example:

```
OPTIONS = {
    "builtins": ["myapp.builtins"],
}
```

Tags and filters from built-in libraries can be used without first calling the *{% load %}* tag.

class Jinja2

Requires Jinja2 to be installed:

```
$ python -m pip install Jinja2
```

Set *BACKEND* to `'django.template.backends.jinja2.Jinja2'` to configure a Jinja2 engine.

When *APP_DIRS* is `True`, `Jinja2` engines look for templates in the `jinja2` subdirectory of installed applications.

The most important entry in *OPTIONS* is `'environment'`. It's a dotted Python path to a callable returning a Jinja2 environment. It defaults to `'jinja2.Environment'`. Django invokes that callable and passes other options as keyword arguments. Furthermore, Django adds defaults that differ from Jinja2's for a few options:

- `'autoescape'`: `True`
- `'loader'`: a loader configured for *DIRS* and *APP_DIRS*
- `'auto_reload'`: `settings.DEBUG`
- `'undefined'`: `DebugUndefined if settings.DEBUG else Undefined`

`Jinja2` engines also accept the following *OPTIONS*:

- `'context_processors'`: a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument

and return a `dict` of items to be merged into the context.

It defaults to an empty list.

> **Using context processors with Jinja2 templates is discouraged.**
>
> Context processors are useful with Django templates because Django templates don't support calling functions with arguments. Since Jinja2 doesn't have that limitation, it's recommended to put the function that you would use as a context processor in the global variables available to the template using `jinja2.Environment` as described below. You can then call that function in the template:
>
> ```
> {{ function(request) }}
> ```
>
> Some Django templates context processors return a fixed value. For Jinja2 templates, this layer of indirection isn't necessary since you can add constants directly in `jinja2.Environment`.
>
> The original use case for adding context processors for Jinja2 involved:
>
> - Making an expensive computation that depends on the request.
> - Needing the result in every template.
> - Using the result multiple times in each template.
>
> Unless all of these conditions are met, passing a function to the template is more in line with the design of Jinja2.

The default configuration is purposefully kept to a minimum. If a template is rendered with a request (e.g. when using *`render()`*), the `Jinja2` backend adds the globals `request`, `csrf_input`, and `csrf_token` to the context. Apart from that, this backend doesn't create a Django-flavored environment. It doesn't know about Django filters and tags. In order to use Django-specific APIs, you must configure them into the environment.

For example, you can create `myproject/jinja2.py` with this content:

```
from django.templatetags.static import static
from django.urls import reverse

from jinja2 import Environment


def environment(**options):
    env = Environment(**options)
    env.globals.update(
        {
            "static": static,
            "url": reverse,
        }
```

(continues on next page)

(continued from previous page)

```
    )
    return env
```

and set the `'environment'` option to `'myproject.jinja2.environment'`.

Then you could use the following constructs in Jinja2 templates:

```
<img src="{{ static('path/to/company-logo.png') }}" alt="Company Logo">

<a href="{{ url('admin:index') }}">Administration</a>
```

The concepts of tags and filters exist both in the Django template language and in Jinja2 but they're used differently. Since Jinja2 supports passing arguments to callables in templates, many features that require a template tag or filter in Django templates can be achieved by calling a function in Jinja2 templates, as shown in the example above. Jinja2's global namespace removes the need for template context processors. The Django template language doesn't have an equivalent of Jinja2 tests.

3.6 Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the class-based views reference documentation.

3.6.1 Introduction to class-based views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

- Organization of code related to specific HTTP methods (`GET`, `POST`, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

The relationship and history of generic views, class-based views, and class-based generic views

In the beginning there was only the view function contract, Django passed your function an *`HttpRequest`* and expected back an *`HttpResponse`*. This was the extent of what Django provided.

Early on it was recognized that there were common idioms and patterns found in view development. Function-based generic views were introduced to abstract these patterns and ease view development for the common cases.

The problem with function-based generic views is that while they covered the simple cases well, there was no way to extend or customize them beyond some configuration options, limiting their usefulness in many real-world applications.

Class-based generic views were created with the same objective as function-based generic views, to make view development easier. However, the way the solution is implemented, through the use of mixins, provides a toolkit that results in class-based generic views being more extensible and flexible than their function-based counterparts.

If you have tried function based generic views in the past and found them lacking, you should not think of class-based generic views as a class-based equivalent, but rather as a fresh approach to solving the original problems that generic views were meant to solve.

The toolkit of base classes and mixins that Django uses to build class-based generic views are built for maximum flexibility, and as such have many hooks in the form of default method implementations and attributes that you are unlikely to be concerned with in the simplest use cases. For example, instead of limiting you to a class-based attribute for `form_class`, the implementation uses a `get_form` method, which calls a `get_form_class` method, which in its default implementation returns the `form_class` attribute of the class. This gives you several options for specifying what form to use, from an attribute, to a fully dynamic, callable hook. These options seem to add hollow complexity for simple situations, but without them, more advanced designs would be limited.

Using class-based views

At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So where the code to handle HTTP `GET` in a view function would look something like:

```
from django.http import HttpResponse


def my_view(request):
    if request.method == "GET":
        # <view logic>
        return HttpResponse("result")
```

In a class-based view, this would become:

```
from django.http import HttpResponse
from django.views import View


class MyView(View):
    def get(self, request):
```

(continues on next page)

(continued from previous page)

```
        # <view logic>
        return HttpResponse("result")
```

Because Django's URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an *as_view()* class method which returns a function that can be called when a request arrives for a URL matching the associated pattern. The function creates an instance of the class, calls *setup()* to initialize its attributes, and then calls its *dispatch()* method. `dispatch` looks at the request to determine whether it is a `GET`, `POST`, etc, and relays the request to a matching method if one is defined, or raises *HttpResponseNotAllowed* if not:

```
# urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [
    path("about/", MyView.as_view()),
]
```

It is worth noting that what your method returns is identical to what you return from a function-based view, namely some form of *HttpResponse*. This means that http shortcuts or *TemplateResponse* objects are valid to use inside a class-based view.

While a minimal class-based view does not require any class attributes to perform its job, class attributes are useful in many class-based designs, and there are two ways to configure or set class attributes.

The first is the standard Python way of subclassing and overriding attributes and methods in the subclass. So that if your parent class had an attribute `greeting` like this:

```
from django.http import HttpResponse
from django.views import View


class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

You can override that in a subclass:

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Another option is to configure class attributes as keyword arguments to the *as_view()* call in the URLconf:

```
urlpatterns = [
    path("about/", GreetingView.as_view(greeting="G'day")),
]
```

> **Note**
>
> While your class is instantiated for each request dispatched to it, class attributes set through the *as_view()* entry point are configured only once at the time your URLs are imported.

Using mixins

Mixins are a form of multiple inheritance where behaviors and attributes of multiple parent classes can be combined.

For example, in the generic class-based views there is a mixin called *TemplateResponseMixin* whose primary purpose is to define the method *render_to_response()*. When combined with the behavior of the *View* base class, the result is a *TemplateView* class that will dispatch requests to the appropriate matching methods (a behavior defined in the `View` base class), and that has a *render_to_response()* method that uses a *template_name* attribute to return a *TemplateResponse* object (a behavior defined in the `TemplateResponseMixin`).

Mixins are an excellent way of reusing code across multiple classes, but they come with some cost. The more your code is scattered among mixins, the harder it will be to read a child class and know what exactly it is doing, and the harder it will be to know which methods from which mixins to override if you are subclassing something that has a deep inheritance tree.

Note also that you can only inherit from one generic view - that is, only one parent class may inherit from *View* and the rest (if any) should be mixins. Trying to inherit from more than one class that inherits from `View` - for example, trying to use a form at the top of a list and combining *ProcessFormView* and *ListView* - won't work as expected.

Handling forms with class-based views

A basic function-based view that handles forms may look something like this:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import MyForm


def myview(request):
```

(continues on next page)

(continued from previous page)

```
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect("/success/")
    else:
        form = MyForm(initial={"key": "value"})

    return render(request, "form_template.html", {"form": form})
```

A similar class-based view might look like:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views import View

from .forms import MyForm


class MyFormView(View):
    form_class = MyForm
    initial = {"key": "value"}
    template_name = "form_template.html"

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {"form": form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect("/success/")

        return render(request, self.template_name, {"form": form})
```

This is a minimal case, but you can see that you would then have the option of customizing this view by overriding any of the class attributes, e.g. `form_class`, via URLconf configuration, or subclassing and overriding one or more of the methods (or both!).

Decorating class-based views

The extension of class-based views isn't limited to using mixins. You can also use decorators. Since class-based views aren't functions, decorating them works differently depending on if you're using `as_view()` or creating a subclass.

Decorating in URLconf

You can adjust class-based views by decorating the result of the *as_view()* method. The easiest place to do this is in the URLconf where you deploy your view:

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = [
    path("about/", login_required(TemplateView.as_view(template_name="secret.html"))),
    path("vote/", permission_required("polls.can_vote")(VoteView.as_view())),
]
```

This approach applies the decorator on a per-instance basis. If you want every instance of a view to be decorated, you need to take a different approach.

Decorating the class

To decorate every instance of a class-based view, you need to decorate the class definition itself. To do this you apply the decorator to the *dispatch()* method of the class.

A method on a class isn't quite the same as a standalone function, so you can't just apply a function decorator to the method – you need to transform it into a method decorator first. The `method_decorator` decorator transforms a function decorator into a method decorator so that it can be used on an instance method. For example:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView



class ProtectedView(TemplateView):
    template_name = "secret.html"

    @method_decorator(login_required)
```

(continues on next page)

(continued from previous page)

```
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
```

Or, more succinctly, you can decorate the class instead and pass the name of the method to be decorated as the keyword argument `name`:

```
@method_decorator(login_required, name="dispatch")
class ProtectedView(TemplateView):
    template_name = "secret.html"
```

If you have a set of common decorators used in several places, you can define a list or tuple of decorators and use this instead of invoking `method_decorator()` multiple times. These two classes are equivalent:

```
decorators = [never_cache, login_required]


@method_decorator(decorators, name="dispatch")
class ProtectedView(TemplateView):
    template_name = "secret.html"


@method_decorator(never_cache, name="dispatch")
@method_decorator(login_required, name="dispatch")
class ProtectedView(TemplateView):
    template_name = "secret.html"
```

The decorators will process a request in the order they are passed to the decorator. In the example, `never_cache()` will process the request before `login_required()`.

In this example, every instance of `ProtectedView` will have login protection. These examples use `login_required`, however, the same behavior can be obtained by using *`LoginRequiredMixin`*.

> **Note**
>
> `method_decorator` passes `*args` and `**kwargs` as parameters to the decorated method on the class. If your method does not accept a compatible set of parameters it will raise a `TypeError` exception.

3.6.2 Built-in class-based generic views

Writing web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but web developers also experience this boredom at the view level.

Django's generic views were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

We can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of any object. Then the model in question can be passed as an extra argument to the URLconf.

Django ships with generic views to do the following:

- Display list and detail pages for a single object. If we were creating an application to manage conferences then a `TalkListView` and a `RegisteredUserListView` would be examples of list views. A single talk page is an example of what we call a "detail" view.
- Present date-based objects in year/month/day archive pages, associated detail, and "latest" pages.
- Allow users to create, update, and delete objects – with or without authorization.

Taken together, these views provide interfaces to perform the most common tasks developers encounter.

Extending generic views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

This is one of the reasons generic views were redesigned for the 1.3 release - previously, they were view functions with a bewildering array of options; now, rather than passing in a large amount of configuration in the URLconf, the recommended way to extend generic views is to subclass them, and override their attributes or methods.

That said, generic views will have a limit. If you find you're struggling to implement your view as a subclass of a generic view, then you may find it more effective to write just the code you need, using your own class-based or functional views.

More examples of generic views are available in some third party applications, or you could write your own as needed.

Generic views of objects

TemplateView certainly is useful, but Django's generic views really shine when it comes to presenting views of your database content. Because it's such a common task, Django comes with a handful of built-in generic views to help generate list and detail views of objects.

Let's start by looking at some examples of showing a list of objects or an individual object.

We'll be using these models:

```
# models.py
from django.db import models


class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):
        return self.name


class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to="author_headshots")

    def __str__(self):
        return self.name


class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField("Author")
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    publication_date = models.DateField()
```

Now we need to define a view:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher
```

(continues on next page)

(continued from previous page)

```
class PublisherListView(ListView):
    model = Publisher
```

Finally hook that view into your urls:

```
# urls.py
from django.urls import path
from books.views import PublisherListView

urlpatterns = [
    path("publishers/", PublisherListView.as_view()),
]
```

That's all the Python code we need to write. We still need to write a template, however. We could explicitly tell the view which template to use by adding a `template_name` attribute to the view, but in the absence of an explicit template Django will infer one from the object's name. In this case, the inferred template will be `"books/publisher_list.html"` – the "books" part comes from the name of the app that defines the model, while the "publisher" bit is the lowercased version of the model's name.

> **Note**
>
> Thus, when (for example) the `APP_DIRS` option of a `DjangoTemplates` backend is set to True in *TEMPLATES*, a template location could be: /path/to/project/books/templates/books/publisher_list.html

This template will be rendered against a context containing a variable called `object_list` that contains all the publisher objects. A template might look like this:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

That's really all there is to it. All the cool features of generic views come from changing the attributes set

on the generic view. The generic views reference documents all the generic views and their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

Making "friendly" template contexts

You might have noticed that our sample publisher list template stores all the publishers in a variable named `object_list`. While this works just fine, it isn't all that "friendly" to template authors: they have to "just know" that they're dealing with publishers here.

Well, if you're dealing with a model object, this is already done for you. When you are dealing with an object or queryset, Django is able to populate the context using the lowercased version of the model class' name. This is provided in addition to the default `object_list` entry, but contains exactly the same data, i.e. `publisher_list`.

If this still isn't a good match, you can manually set the name of the context variable. The `context_object_name` attribute on a generic view specifies the context variable to use:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher


class PublisherListView(ListView):
    model = Publisher
    context_object_name = "my_favorite_publishers"
```

Providing a useful `context_object_name` is always a good idea. Your coworkers who design templates will thank you.

Adding extra context

Often you need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the books on each publisher detail page. The *DetailView* generic view provides the publisher to the context, but how do we get additional information in that template?

The answer is to subclass *DetailView* and provide your own implementation of the `get_context_data` method. The default implementation adds the object being displayed to the template, but you can override it to send more:

```
from django.views.generic import DetailView
from books.models import Book, Publisher


```

(continues on next page)

(continued from previous page)

```
class PublisherDetailView(DetailView):
    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super().get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context["book_list"] = Book.objects.all()
        return context
```

> **Note**
>
> Generally, `get_context_data` will merge the context data of all parent classes with those of the current class. To preserve this behavior in your own classes where you want to alter the context, you should be sure to call `get_context_data` on the super class. When no two classes try to define the same key, this will give the expected results. However if any class attempts to override a key after parent classes have set it (after the call to super), any children of that class will also need to explicitly set it after super if they want to be sure to override all parents. If you're having trouble, review the method resolution order of your view.
>
> Another consideration is that the context data from class-based generic views will override data provided by context processors; see *`get_context_data()`* for an example.

Viewing subsets of objects

Now let's take a closer look at the `model` argument we've been using all along. The `model` argument, which specifies the database model that the view will operate upon, is available on all the generic views that operate on a single object or a collection of objects. However, the `model` argument is not the only way to specify the objects that the view will operate upon – you can also specify the list of objects using the `queryset` argument:

```
from django.views.generic import DetailView
from books.models import Publisher


class PublisherDetailView(DetailView):
    context_object_name = "publisher"
    queryset = Publisher.objects.all()
```

Specifying `model = Publisher` is shorthand for saying `queryset = Publisher.objects.all()`. However, by using `queryset` to define a filtered list of objects you can be more specific about the objects that will be visible in the view (see Making queries for more information about *`QuerySet`* objects, and see the class-based

views reference for the complete details).

To pick an example, we might want to order a list of books by publication date, with the most recent first:

```
from django.views.generic import ListView
from books.models import Book


class BookListView(ListView):
    queryset = Book.objects.order_by("-publication_date")
    context_object_name = "book_list"
```

That's a pretty minimal example, but it illustrates the idea nicely. You'll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

```
from django.views.generic import ListView
from books.models import Book


class AcmeBookListView(ListView):
    context_object_name = "book_list"
    queryset = Book.objects.filter(publisher__name="ACME Publishing")
    template_name = "books/acme_list.html"
```

Notice that along with a filtered `queryset`, we're also using a custom template name. If we didn't, the generic view would use the same template as the "vanilla" object list, which might not be what we want.

Also notice that this isn't a very elegant way of doing publisher-specific books. If we want to add another publisher page, we'd need another handful of lines in the URLconf, and more than a few publishers would get unreasonable. We'll deal with this problem in the next section.

> **Note**
>
> If you get a 404 when requesting `/books/acme/`, check to ensure you actually have a Publisher with the name 'ACME Publishing'. Generic views have an `allow_empty` parameter for this case. See the class-based-views reference for more details.

Dynamic filtering

Another common need is to filter down the objects given in a list page by some key in the URL. Earlier we hard-coded the publisher's name in the URLconf, but what if we wanted to write a view that displayed all the books by some arbitrary publisher?

Handily, the `ListView` has a *`get_queryset()`* method we can override. By default, it returns the value of the `queryset` attribute, but we can use it to add more logic.

The key part to making this work is that when class-based views are called, various useful things are stored on `self`; as well as the request (`self.request`) this includes the positional (`self.args`) and name-based (`self.kwargs`) arguments captured according to the URLconf.

Here, we have a URLconf with a single captured group:

```
# urls.py
from django.urls import path
from books.views import PublisherBookListView

urlpatterns = [
    path("books/<publisher>/", PublisherBookListView.as_view()),
]
```

Next, we'll write the `PublisherBookListView` view itself:

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher


class PublisherBookListView(ListView):
    template_name = "books/books_by_publisher.html"

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.kwargs["publisher"])
        return Book.objects.filter(publisher=self.publisher)
```

Using `get_queryset` to add logic to the queryset selection is as convenient as it is powerful. For instance, if we wanted, we could use `self.request.user` to filter using the current user, or other more complex logic.

We can also add the publisher into the context at the same time, so we can use it in the template:

```
# ...


def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super().get_context_data(**kwargs)
    # Add in the publisher
```

(continues on next page)

(continued from previous page)

```
    context["publisher"] = self.publisher
    return context
```

Performing extra work

The last common pattern we'll look at involves doing some extra work before or after calling the generic view.

Imagine we had a `last_accessed` field on our `Author` model that we were using to keep track of the last time anybody looked at that author:

```
# models.py
from django.db import models


class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to="author_headshots")
    last_accessed = models.DateTimeField()
```

The generic `DetailView` class wouldn't know anything about this field, but once again we could write a custom view to keep that field updated.

First, we'd need to add an author detail bit in the URLconf to point to a custom view:

```
from django.urls import path
from books.views import AuthorDetailView

urlpatterns = [
    # ...
    path("authors/<int:pk>/", AuthorDetailView.as_view(), name="author-detail"),
]
```

Then we'd write our new view – `get_object` is the method that retrieves the object – so we override it and wrap the call:

```
from django.utils import timezone
from django.views.generic import DetailView
from books.models import Author
```

(continues on next page)

(continued from previous page)

```
class AuthorDetailView(DetailView):
    queryset = Author.objects.all()

    def get_object(self):
        obj = super().get_object()
        # Record the last accessed date
        obj.last_accessed = timezone.now()
        obj.save()
        return obj
```

> **Note**
>
> The URLconf here uses the named group `pk` - this name is the default name that `DetailView` uses to find the value of the primary key used to filter the queryset.
>
> If you want to call the group something else, you can set *`pk_url_kwarg`* on the view.

3.6.3 Form handling with class-based views

Form processing generally has 3 paths:

- Initial GET (blank or prepopulated form)
- POST with invalid data (typically redisplay form with errors)
- POST with valid data (process the data and typically redirect)

Implementing this yourself often results in a lot of repeated boilerplate code (see Using a form in a view). To help avoid this, Django provides a collection of generic class-based views for form processing.

Basic forms

Given a contact form:

Listing 15: `forms.py`

```
from django import forms


class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)
```

(continues on next page)

(continued from previous page)

```
    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

The view can be constructed using a `FormView`:

Listing 16: `views.py`

```
from myapp.forms import ContactForm
from django.views.generic.edit import FormView


class ContactFormView(FormView):
    template_name = "contact.html"
    form_class = ContactForm
    success_url = "/thanks/"

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super().form_valid(form)
```

Notes:

- FormView inherits *TemplateResponseMixin* so *template_name* can be used here.
- The default implementation for *form_valid()* simply redirects to the *success_url*.

Model forms

Generic views really shine when working with models. These generic views will automatically create a *ModelForm*, so long as they can work out which model class to use:

- If the *model* attribute is given, that model class will be used.
- If *get_object()* returns an object, the class of that object will be used.
- If a *queryset* is given, the model for that queryset will be used.

Model form views provide a *form_valid()* implementation that saves the model automatically. You can override this if you have any special requirements; see below for examples.

You don't even need to provide a `success_url` for *CreateView* or *UpdateView* - they will use *get_absolute_url()* on the model object if available.

If you want to use a custom *ModelForm* (for instance to add extra validation), set *form_class* on your view.

> **Note**
>
> When specifying a custom form class, you must still specify the model, even though the *form_class* may be a *ModelForm*.

First we need to add *get_absolute_url()* to our `Author` class:

Listing 17: `models.py`

```
from django.db import models
from django.urls import reverse


class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse("author-detail", kwargs={"pk": self.pk})
```

Then we can use *CreateView* and friends to do the actual work. Notice how we're just configuring the generic class-based views here; we don't have to write any logic ourselves:

Listing 18: `views.py`

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView, DeleteView, UpdateView
from myapp.models import Author


class AuthorCreateView(CreateView):
    model = Author
    fields = ["name"]


class AuthorUpdateView(UpdateView):
    model = Author
    fields = ["name"]


class AuthorDeleteView(DeleteView):
    model = Author
    success_url = reverse_lazy("author-list")
```

> **Note**
>
> We have to use *reverse_lazy()* instead of `reverse()`, as the urls are not loaded when the file is imported.

The `fields` attribute works the same way as the `fields` attribute on the inner `Meta` class on *ModelForm*. Unless you define the form class in another way, the attribute is required and the view will raise an *ImproperlyConfigured* exception if it's not.

If you specify both the *fields* and *form_class* attributes, an *ImproperlyConfigured* exception will be raised.

Finally, we hook these new views into the URLconf:

Listing 19: `urls.py`

```
from django.urls import path
from myapp.views import AuthorCreateView, AuthorDeleteView, AuthorUpdateView

urlpatterns = [
    # ...
    path("author/add/", AuthorCreateView.as_view(), name="author-add"),
    path("author/<int:pk>/", AuthorUpdateView.as_view(), name="author-update"),
    path("author/<int:pk>/delete/", AuthorDeleteView.as_view(), name="author-delete"),
]
```

> **Note**
>
> These views inherit *SingleObjectTemplateResponseMixin* which uses *template_name_suffix* to construct the *template_name* based on the model.
>
> In this example:
>
> - *CreateView* and *UpdateView* use `myapp/author_form.html`
> - *DeleteView* uses `myapp/author_confirm_delete.html`
>
> If you wish to have separate templates for *CreateView* and *UpdateView*, you can set either *template_name* or *template_name_suffix* on your view class.

Models and `request.user`

To track the user that created an object using a *CreateView*, you can use a custom *ModelForm* to do this. First, add the foreign key relation to the model:

Listing 20: `models.py`

```
from django.contrib.auth.models import User
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)

    # ...
```

In the view, ensure that you don't include `created_by` in the list of fields to edit, and override *`form_valid()`* to add the user:

Listing 21: `views.py`

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic.edit import CreateView
from myapp.models import Author


class AuthorCreateView(LoginRequiredMixin, CreateView):
    model = Author
    fields = ["name"]

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super().form_valid(form)
```

`LoginRequiredMixin` prevents users who aren't logged in from accessing the form. If you omit that, you'll need to handle unauthorized users in *`form_valid()`*.

Content negotiation example

Here is an example showing how you might go about implementing a form that works with an API-based workflow as well as 'normal' form POSTs:

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

```

(continues on next page)

(continued from previous page)

```
class JsonableResponseMixin:
    """
    Mixin to add JSON support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """

    def form_invalid(self, form):
        response = super().form_invalid(form)
        if self.request.accepts("text/html"):
            return response
        else:
            return JsonResponse(form.errors, status=400)

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super().form_valid(form)
        if self.request.accepts("text/html"):
            return response
        else:
            data = {
                "pk": self.object.pk,
            }
            return JsonResponse(data)


class AuthorCreateView(JsonableResponseMixin, CreateView):
    model = Author
    fields = ["name"]
```

The above example assumes that if the client supports `text/html`, that they would prefer it. However, this may not always be true. When requesting a `.css` file, many browsers will send the header `Accept: text/css,*/*;q=0.1`, indicating that they would prefer CSS, but anything else is fine. This means `request.accepts("text/html")` will be `True`.

To determine the correct format, taking into consideration the client's preference, use *`django.http.HttpRequest.get_preferred_type()`*:

```
class JsonableResponseMixin:
    """
    Mixin to add JSON support to a form.
    Must be used with an object-based FormView (e.g. CreateView).
    """

    accepted_media_types = ["text/html", "application/json"]

    def dispatch(self, request, *args, **kwargs):
        if request.get_preferred_type(self.accepted_media_types) is None:
            # No format in common.
            return HttpResponse(
                status_code=406, headers={"Accept": ",".join(self.accepted_media_types)}
            )

        return super().dispatch(request, *args, **kwargs)

    def form_invalid(self, form):
        response = super().form_invalid(form)
        accepted_type = request.get_preferred_type(self.accepted_media_types)
        if accepted_type == "text/html":
            return response
        elif accepted_type == "application/json":
            return JsonResponse(form.errors, status=400)

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super().form_valid(form)
        accepted_type = request.get_preferred_type(self.accepted_media_types)
        if accepted_type == "text/html":
            return response
        elif accepted_type == "application/json":
            data = {
                "pk": self.object.pk,
            }
            return JsonResponse(data)
```

The *HttpRequest.get_preferred_type()* method was added.

3.6.4 Using mixins with class-based views

> **Caution**
>
> This is an advanced topic. A working knowledge of Django's class-based views is advised before exploring these techniques.

Django's built-in class-based views provide a lot of functionality, but some of it you may want to use separately. For instance, you may want to write a view that renders a template to make the HTTP response, but you can't use *TemplateView*; perhaps you need to render a template only on `POST`, with `GET` doing something else entirely. While you could use *TemplateResponse* directly, this will likely result in duplicate code.

For this reason, Django also provides a number of mixins that provide more discrete functionality. Template rendering, for instance, is encapsulated in the *TemplateResponseMixin*. The Django reference documentation contains full documentation of all the mixins.

Context and template responses

Two central mixins are provided that help in providing a consistent interface to working with templates in class-based views.

TemplateResponseMixin

Every built in view which returns a *TemplateResponse* will call the *render_to_response()* method that `TemplateResponseMixin` provides. Most of the time this will be called for you (for instance, it is called by the `get()` method implemented by both *TemplateView* and *DetailView*); similarly, it's unlikely that you'll need to override it, although if you want your response to return something not rendered via a Django template then you'll want to do it. For an example of this, see the JSONResponseMixin example.

`render_to_response()` itself calls *get_template_names()*, which by default will look up *template_name* on the class-based view; two other mixins (*SingleObjectTemplateResponseMixin* and *MultipleObjectTemplateResponseMixin*) override this to provide more flexible defaults when dealing with actual objects.

ContextMixin

Every built in view which needs context data, such as for rendering a template (including `TemplateResponseMixin` above), should call *get_context_data()* passing any data they want to ensure is in there as keyword arguments. `get_context_data()` returns a dictionary; in `ContextMixin` it returns its keyword arguments, but it is common to override this to add more members to the dictionary. You can also use the *extra_context* attribute.

Building up Django's generic class-based views

Let's look at how two of Django's generic class-based views are built out of mixins providing discrete functionality. We'll consider *DetailView*, which renders a "detail" view of an object, and *ListView*, which will render a list of objects, typically from a queryset, and optionally paginate them. This will introduce us to four mixins which between them provide useful functionality when working with either a single Django object, or multiple objects.

There are also mixins involved in the generic edit views (*FormView*, and the model-specific views *CreateView*, *UpdateView* and *DeleteView*), and in the date-based generic views. These are covered in the mixin reference documentation.

`DetailView`: working with a single Django object

To show the detail of an object, we basically need to do two things: we need to look up the object and then we need to make a *TemplateResponse* with a suitable template, and that object as context.

To get the object, *DetailView* relies on *SingleObjectMixin*, which provides a *get_object()* method that figures out the object based on the URL of the request (it looks for `pk` and `slug` keyword arguments as declared in the URLConf, and looks the object up either from the *model* attribute on the view, or the *queryset* attribute if that's provided). `SingleObjectMixin` also overrides *get_context_data()*, which is used across all Django's built in class-based views to supply context data for template renders.

To then make a *TemplateResponse*, *DetailView* uses *SingleObjectTemplateResponseMixin*, which extends *TemplateResponseMixin*, overriding *get_template_names()* as discussed above. It actually provides a fairly sophisticated set of options, but the main one that most people are going to use is `<app_label>/<model_name>_detail.html`. The `_detail` part can be changed by setting *template_name_suffix* on a subclass to something else. (For instance, the generic edit views use `_form` for create and update views, and `_confirm_delete` for delete views.)

`ListView`: working with many Django objects

Lists of objects follow roughly the same pattern: we need a (possibly paginated) list of objects, typically a *QuerySet*, and then we need to make a *TemplateResponse* with a suitable template using that list of objects.

To get the objects, *ListView* uses *MultipleObjectMixin*, which provides both *get_queryset()* and *paginate_queryset()*. Unlike with *SingleObjectMixin*, there's no need to key off parts of the URL to figure out the queryset to work with, so the default uses the *queryset* or *model* attribute on the view class. A common reason to override *get_queryset()* here would be to dynamically vary the objects, such as depending on the current user or to exclude posts in the future for a blog.

MultipleObjectMixin also overrides *get_context_data()* to include appropriate context variables for pagination (providing dummies if pagination is disabled). It relies on `object_list` being passed in as a keyword argument, which *ListView* arranges for it.

To make a *TemplateResponse*, *ListView* then uses *MultipleObjectTemplateResponseMixin*; as with *SingleObjectTemplateResponseMixin* above, this overrides `get_template_names()` to provide *a range*

of options, with the most commonly-used being `<app_label>/<model_name>_list.html`, with the `_list` part again being taken from the *template_name_suffix* attribute. (The date based generic views use suffixes such as `_archive`, `_archive_year` and so on to use different templates for the various specialized date-based list views.)

Using Django's class-based view mixins

Now we've seen how Django's generic class-based views use the provided mixins, let's look at other ways we can combine them. We're still going to be combining them with either built-in class-based views, or other generic class-based views, but there are a range of rarer problems you can solve than are provided for by Django out of the box.

> **Warning**
>
> Not all mixins can be used together, and not all generic class based views can be used with all other mixins. Here we present a few examples that do work; if you want to bring together other functionality then you'll have to consider interactions between attributes and methods that overlap between the different classes you're using, and how method resolution order will affect which versions of the methods will be called in what order.
>
> The reference documentation for Django's class-based views and class-based view mixins will help you in understanding which attributes and methods are likely to cause conflict between different classes and mixins.
>
> If in doubt, it's often better to back off and base your work on *View* or *TemplateView*, perhaps with *SingleObjectMixin* and *MultipleObjectMixin*. Although you will probably end up writing more code, it is more likely to be clearly understandable to someone else coming to it later, and with fewer interactions to worry about you will save yourself some thinking. (Of course, you can always dip into Django's implementation of the generic class-based views for inspiration on how to tackle problems.)

Using `SingleObjectMixin` with View

If we want to write a class-based view that responds only to `POST`, we'll subclass *View* and write a `post()` method in the subclass. However if we want our processing to work on a particular object, identified from the URL, we'll want the functionality provided by *SingleObjectMixin*.

We'll demonstrate this with the `Author` model we used in the generic class-based views introduction.

Listing 22: `views.py`

```
from django.http import HttpResponseForbidden, HttpResponseRedirect
from django.urls import reverse
from django.views import View
from django.views.generic.detail import SingleObjectMixin
```

(continues on next page)

(continued from previous page)

```
from books.models import Author


class RecordInterestView(SingleObjectMixin, View):
    """Records the current user's interest in an author."""

    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(
            reverse("author-detail", kwargs={"pk": self.object.pk})
        )
```

In practice you'd probably want to record the interest in a key-value store rather than in a relational database, so we've left that bit out. The only bit of the view that needs to worry about using *SingleObjectMixin* is where we want to look up the author we're interested in, which it does with a call to `self.get_object()`. Everything else is taken care of for us by the mixin.

We can hook this into our URLs easily enough:

Listing 23: `urls.py`

```
from django.urls import path
from books.views import RecordInterestView

urlpatterns = [
    # ...
    path(
        "author/<int:pk>/interest/",
        RecordInterestView.as_view(),
        name="author-interest",
    ),
]
```

Note the `pk` named group, which *get_object()* uses to look up the `Author` instance. You could also use a

slug, or any of the other features of *SingleObjectMixin*.

Using SingleObjectMixin with ListView

ListView provides built-in pagination, but you might want to paginate a list of objects that are all linked (by a foreign key) to another object. In our publishing example, you might want to paginate through all the books by a particular publisher.

One way to do this is to combine *ListView* with *SingleObjectMixin*, so that the queryset for the paginated list of books can hang off the publisher found as the single object. In order to do this, we need to have two different querysets:

`Book` queryset for use by *ListView*
: Since we have access to the `Publisher` whose books we want to list, we override `get_queryset()` and use the `Publisher`'s reverse foreign key manager.

`Publisher` queryset for use in *get_object()*
: We'll rely on the default implementation of `get_object()` to fetch the correct `Publisher` object. However, we need to explicitly pass a `queryset` argument because otherwise the default implementation of `get_object()` would call `get_queryset()` which we have overridden to return `Book` objects instead of `Publisher` ones.

> **Note**
>
> We have to think carefully about `get_context_data()`. Since both *SingleObjectMixin* and *ListView* will put things in the context data under the value of `context_object_name` if it's set, we'll instead explicitly ensure the `Publisher` is in the context data. *ListView* will add in the suitable `page_obj` and `paginator` for us providing we remember to call `super()`.

Now we can write a new `PublisherDetailView`:

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher


class PublisherDetailView(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Publisher.objects.all())
        return super().get(request, *args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["publisher"] = self.object
        return context

    def get_queryset(self):
        return self.object.book_set.all()
```

Notice how we set `self.object` within `get()` so we can use it again later in `get_context_data()` and `get_queryset()`. If you don't set `template_name`, the template will default to the normal *ListView* choice, which in this case would be `"books/book_list.html"` because it's a list of books; *ListView* knows nothing about *SingleObjectMixin*, so it doesn't have any clue this view is anything to do with a `Publisher`.

The `paginate_by` is deliberately small in the example so you don't have to create lots of books to see the pagination working! Here's the template you'd want to use:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publisher {{ publisher.name }}</h2>

    <ol>
      {% for book in page_obj %}
        <li>{{ book.title }}</li>
      {% endfor %}
    </ol>

    <div class="pagination">
        <span class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number }}">previous</a>
            {% endif %}

            <span class="current">
                Page {{ page_obj.number }} of {{ paginator.num_pages }}.
            </span>

            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">next</a>
            {% endif %}
```

(continues on next page)

(continued from previous page)

```
        </span>
    </div>
{% endblock %}
```

Avoid anything more complex

Generally you can use *TemplateResponseMixin* and *SingleObjectMixin* when you need their functionality. As shown above, with a bit of care you can even combine `SingleObjectMixin` with *ListView*. However things get increasingly complex as you try to do so, and a good rule of thumb is:

> **Hint**
>
> Each of your views should use only mixins or views from one of the groups of generic class-based views: detail, list, editing and date. For example it's fine to combine *TemplateView* (built in view) with *MultipleObjectMixin* (generic list), but you're likely to have problems combining `SingleObjectMixin` (generic detail) with `MultipleObjectMixin` (generic list).

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. First, let's look at a naive attempt to combine *DetailView* with *FormMixin* to enable us to `POST` a Django *Form* to the same URL as we're displaying an object using *DetailView*.

Using `FormMixin` with `DetailView`

Think back to our earlier example of using *View* and *SingleObjectMixin* together. We were recording a user's interest in a particular author; say now that we want to let them leave a message saying why they like them. Again, let's assume we're not going to store this in a relational database but instead in something more esoteric that we won't worry about here.

At this point it's natural to reach for a *Form* to encapsulate the information sent from the user's browser to Django. Say also that we're heavily invested in REST, so we want to use the same URL for displaying the author as for capturing the message from the user. Let's rewrite our `AuthorDetailView` to do that.

We'll keep the `GET` handling from *DetailView*, although we'll have to add a *Form* into the context data so we can render it in the template. We'll also want to pull in form processing from *FormMixin*, and write a bit of code so that on `POST` the form gets called appropriately.

> **Note**
>
> We use *FormMixin* and implement `post()` ourselves rather than try to mix *DetailView* with *FormView* (which provides a suitable `post()` already) because both of the views implement `get()`, and things would get much more confusing.

Our new `AuthorDetailView` looks like this:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseForbidden
from django.urls import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin
from books.models import Author


class AuthorInterestForm(forms.Form):
    message = forms.CharField()


class AuthorDetailView(FormMixin, DetailView):
    model = Author
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse("author-detail", kwargs={"pk": self.object.pk})

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()
        self.object = self.get_object()
        form = self.get_form()
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        # Here, we would record the user's interest using the message
        # passed in form.cleaned_data['message']
        return super().form_valid(form)
```

`get_success_url()` provides somewhere to redirect to, which gets used in the default implementation of

`form_valid()`. We have to provide our own `post()` as noted earlier.

A better solution

The number of subtle interactions between *FormMixin* and *DetailView* is already testing our ability to manage things. It's unlikely you'd want to write this kind of class yourself.

In this case, you could write the `post()` method yourself, keeping *DetailView* as the only generic functionality, although writing *Form* handling code involves a lot of duplication.

Alternatively, it would still be less work than the above approach to have a separate view for processing the form, which could use *FormView* distinct from *DetailView* without concerns.

An alternative better solution

What we're really trying to do here is to use two different class based views from the same URL. So why not do just that? We have a very clear division here: `GET` requests should get the *DetailView* (with the *Form* added to the context data), and `POST` requests should get the *FormView*. Let's set up those views first.

The `AuthorDetailView` view is almost the same as when we first introduced AuthorDetailView; we have to write our own `get_context_data()` to make the `AuthorInterestForm` available to the template. We'll skip the `get_object()` override from before for clarity:

```
from django import forms
from django.views.generic import DetailView
from books.models import Author


class AuthorInterestForm(forms.Form):
    message = forms.CharField()


class AuthorDetailView(DetailView):
    model = Author

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["form"] = AuthorInterestForm()
        return context
```

Then the `AuthorInterestFormView` is a *FormView*, but we have to bring in *SingleObjectMixin* so we can find the author we're talking about, and we have to remember to set `template_name` to ensure that form errors will render the same template as `AuthorDetailView` is using on `GET`:

```
from django.http import HttpResponseForbidden
from django.urls import reverse
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin


class AuthorInterestFormView(SingleObjectMixin, FormView):
    template_name = "books/author_detail.html"
    form_class = AuthorInterestForm
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()
        self.object = self.get_object()
        return super().post(request, *args, **kwargs)

    def get_success_url(self):
        return reverse("author-detail", kwargs={"pk": self.object.pk})
```

Finally we bring this together in a new `AuthorView` view. We already know that calling *as_view()* on a class-based view gives us something that behaves exactly like a function based view, so we can do that at the point we choose between the two subviews.

You can pass through keyword arguments to *as_view()* in the same way you would in your URLconf, such as if you wanted the `AuthorInterestFormView` behavior to also appear at another URL but using a different template:

```
from django.views import View


class AuthorView(View):
    def get(self, request, *args, **kwargs):
        view = AuthorDetailView.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterestFormView.as_view()
        return view(request, *args, **kwargs)
```

This approach can also be used with any other generic class-based views or your own class-based views inheriting directly from *View* or *TemplateView*, as it keeps the different views as separate as possible.

More than just HTML

Where class-based views shine is when you want to do the same thing many times. Suppose you're writing an API, and every view should return JSON instead of rendered HTML.

We can create a mixin class to use in all of our views, handling the conversion to JSON once.

For example, a JSON mixin might look something like this:

```
from django.http import JsonResponse


class JSONResponseMixin:
    """
    A mixin that can be used to render a JSON response.
    """

    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return JsonResponse(self.get_data(context), **response_kwargs)

    def get_data(self, context):
        """
        Returns an object that will be serialized as JSON by json.dumps().
        """
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return context
```

> **Note**
>
> Check out the Serializing Django objects documentation for more information on how to correctly transform Django models and querysets into JSON.

This mixin provides a `render_to_json_response()` method with the same signature as *`render_to_response()`*. To use it, we need to mix it into a `TemplateView` for example, and override `render_to_response()` to call `render_to_json_response()` instead:

```
from django.views.generic import TemplateView


class JSONView(JSONResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

Equally we could use our mixin with one of the generic views. We can make our own version of *DetailView* by mixing `JSONResponseMixin` with the *BaseDetailView* – (the *DetailView* before template rendering behavior has been mixed in):

```
from django.views.generic.detail import BaseDetailView


class JSONDetailView(JSONResponseMixin, BaseDetailView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

This view can then be deployed in the same way as any other *DetailView*, with exactly the same behavior – except for the format of the response.

If you want to be really adventurous, you could even mix a *DetailView* subclass that is able to return both HTML and JSON content, depending on some property of the HTTP request, such as a query argument or an HTTP header. Mix in both the `JSONResponseMixin` and a *SingleObjectTemplateResponseMixin*, and override the implementation of *render_to_response()* to defer to the appropriate rendering method depending on the type of response that the user requested:

```
from django.views.generic.detail import SingleObjectTemplateResponseMixin


class HybridDetailView(
    JSONResponseMixin, SingleObjectTemplateResponseMixin, BaseDetailView
):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get("format") == "json":
            return self.render_to_json_response(context)
        else:
            return super().render_to_response(context)
```

Because of the way that Python resolves method overloading, the call to `super().render_to_response(context)` ends up calling the *render_to_response()* implementation of *TemplateResponseMixin*.

3.6.5 Basic examples

Django provides base view classes which will suit a wide range of applications. All views inherit from the *View* class, which handles linking the view into the URLs, HTTP method dispatching and other common features. *RedirectView* provides a HTTP redirect, and *TemplateView* extends the base class to make it also render a template.

3.6.6 Usage in your URLconf

The most direct way to use generic views is to create them directly in your URLconf. If you're only changing a few attributes on a class-based view, you can pass them into the *as_view()* method call itself:

```
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
    path("about/", TemplateView.as_view(template_name="about.html")),
]
```

Any arguments passed to *as_view()* will override attributes set on the class. In this example, we set `template_name` on the `TemplateView`. A similar overriding pattern can be used for the `url` attribute on *RedirectView*.

3.6.7 Subclassing generic views

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the `template_name`) or methods (such as `get_context_data`) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, `about.html`. Django has a generic view to do this - *TemplateView* - so we can subclass it, and override the template name:

```
# some_app/views.py
from django.views.generic import TemplateView


class AboutView(TemplateView):
    template_name = "about.html"
```

Then we need to add this new view into our URLconf. *TemplateView* is a class, not a function, so we point the URL to the *as_view()* class method instead, which provides a function-like entry to class-based views:

```
# urls.py
from django.urls import path
from some_app.views import AboutView
```

(continues on next page)

(continued from previous page)

```
urlpatterns = [
    path("about/", AboutView.as_view()),
]
```

For more information on how to use the built in generic views, consult the next topic on generic class-based views.

Supporting other HTTP methods

Suppose somebody wants to access our book library over HTTP using the views as an API. The API client would connect every now and then and download book data for the books published since last visit. But if no new books appeared since then, it is a waste of CPU time and bandwidth to fetch the books from the database, render a full response and send it to the client. It might be preferable to ask the API when the most recent book was published.

We map the URL to book list view in the URLconf:

```
from django.urls import path
from books.views import BookListView

urlpatterns = [
    path("books/", BookListView.as_view()),
]
```

And the view:

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book


class BookListView(ListView):
    model = Book

    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest("publication_date")
        response = HttpResponse(
            # RFC 1123 date format.
            headers={
                "Last-Modified": last_book.publication_date.strftime(
                    "%a, %d %b %Y %H:%M:%S GMT"
```

(continues on next page)

(continued from previous page)

```
                )
            },
        )
        return response
```

If the view is accessed from a `GET` request, an object list is returned in the response (using the `book_list.html` template). But if the client issues a `HEAD` request, the response has an empty body and the `Last-Modified` header indicates when the most recent book was published. Based on this information, the client may or may not download the full object list.

3.6.8 Asynchronous class-based views

As well as the synchronous (`def`) method handlers already shown, `View` subclasses may define asynchronous (`async def`) method handlers to leverage asynchronous code using `await`:

```
import asyncio
from django.http import HttpResponse
from django.views import View


class AsyncView(View):
    async def get(self, request, *args, **kwargs):
        # Perform io-blocking view logic using await, sleep for example.
        await asyncio.sleep(1)
        return HttpResponse("Hello async world!")
```

Within a single view-class, all user-defined method handlers must be either synchronous, using `def`, or all asynchronous, using `async def`. An `ImproperlyConfigured` exception will be raised in `as_view()` if `def` and `async def` declarations are mixed.

Django will automatically detect asynchronous views and run them in an asynchronous context. You can read more about Django's asynchronous support, and how to best use async views, in Asynchronous support.

3.7 Migrations

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into.

3.7.1 The Commands

There are several commands which you will use to interact with migrations and Django's handling of database schema:

- *migrate*, which is responsible for applying and unapplying migrations.
- *makemigrations*, which is responsible for creating new migrations based on the changes you have made to your models.
- *sqlmigrate*, which displays the SQL statements for a migration.
- *showmigrations*, which lists a project's migrations and their status.

You should think of migrations as a version control system for your database schema. `makemigrations` is responsible for packaging up your model changes into individual migration files - analogous to commits - and `migrate` is responsible for applying those to your database.

The migration files for each app live in a "migrations" directory inside of that app, and are designed to be committed to, and distributed as part of, its codebase. You should be making them once on your development machine and then running the same migrations on your colleagues' machines, your staging machines, and eventually your production machines.

> **Note**
>
> It is possible to override the name of the package which contains the migrations on a per-app basis by modifying the *MIGRATION_MODULES* setting.

Migrations will run the same way on the same dataset and produce consistent results, meaning that what you see in development and staging is, under the same circumstances, exactly what will happen in production.

Django will make migrations for any change to your models or fields - even options that don't affect the database - as the only way it can reconstruct a field correctly is to have all the changes in the history, and you might need those options in some data migrations later on (for example, if you've set custom validators).

3.7.2 Backend Support

Migrations are supported on all backends that Django ships with, as well as any third-party backends if they have programmed in support for schema alteration (done via the SchemaEditor class).

However, some databases are more capable than others when it comes to schema migrations; some of the caveats are covered below.

PostgreSQL

PostgreSQL is the most capable of all the databases here in terms of schema support.

MySQL

MySQL lacks support for transactions around schema alteration operations, meaning that if a migration fails to apply you will have to manually unpick the changes in order to try again (it's impossible to roll back to an earlier point).

MySQL 8.0 introduced significant performance enhancements for DDL operations, making them more efficient and reducing the need for full table rebuilds. However, it cannot guarantee a complete absence of locks or interruptions. In situations where locks are still necessary, the duration of these operations will be proportionate to the number of rows involved.

Finally, MySQL has a relatively small limit on the combined size of all columns an index covers. This means that indexes that are possible on other backends will fail to be created under MySQL.

SQLite

SQLite has very little built-in schema alteration support, and so Django attempts to emulate it by:

- Creating a new table with the new schema
- Copying the data across
- Dropping the old table
- Renaming the new table to match the original name

This process generally works well, but it can be slow and occasionally buggy. It is not recommended that you run and migrate SQLite in a production environment unless you are very aware of the risks and its limitations; the support Django ships with is designed to allow developers to use SQLite on their local machines to develop less complex Django projects without the need for a full database.

3.7.3 Workflow

Django can create migrations for you. Make changes to your models - say, add a field and remove a model - and then run *`makemigrations`*:

```
$ python manage.py makemigrations
Migrations for 'books':
  books/migrations/0003_auto.py:
    ~ Alter field author on book
```

Your models will be scanned and compared to the versions currently contained in your migration files, and then a new set of migrations will be written out. Make sure to read the output to see what `makemigrations` thinks you have changed - it's not perfect, and for complex changes it might not be detecting what you expect.

Once you have your new migration files, you should apply them to your database to make sure they work as expected:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: books
Running migrations:
  Rendering model states... DONE
  Applying books.0003_auto... OK
```

Once the migration is applied, commit the migration and the models change to your version control system as a single commit - that way, when other developers (or your production servers) check out the code, they'll get both the changes to your models and the accompanying migration at the same time.

If you want to give the migration(s) a meaningful name instead of a generated one, you can use the *makemigrations --name* option:

```
$ python manage.py makemigrations --name changed_my_model your_app_label
```

Version control

Because migrations are stored in version control, you'll occasionally come across situations where you and another developer have both committed a migration to the same app at the same time, resulting in two migrations with the same number.

Don't worry - the numbers are just there for developers' reference, Django just cares that each migration has a different name. Migrations specify which other migrations they depend on - including earlier migrations in the same app - in the file, so it's possible to detect when there's two new migrations for the same app that aren't ordered.

When this happens, Django will prompt you and give you some options. If it thinks it's safe enough, it will offer to automatically linearize the two migrations for you. If not, you'll have to go in and modify the migrations yourself - don't worry, this isn't difficult, and is explained more in Migration files below.

3.7.4 Transactions

On databases that support DDL transactions (SQLite and PostgreSQL), all migration operations will run inside a single transaction by default. In contrast, if a database doesn't support DDL transactions (e.g. MySQL, Oracle) then all operations will run without a transaction.

You can prevent a migration from running in a transaction by setting the `atomic` attribute to `False`. For example:

```
from django.db import migrations

```

(continues on next page)

(continued from previous page)

```
class Migration(migrations.Migration):
    atomic = False
```

It's also possible to execute parts of the migration inside a transaction using *atomic()* or by passing `atomic=True` to *RunPython*. See Non-atomic migrations for more details.

3.7.5 Dependencies

While migrations are per-app, the tables and relationships implied by your models are too complex to be created for one app at a time. When you make a migration that requires something else to run - for example, you add a `ForeignKey` in your `books` app to your `authors` app - the resulting migration will contain a dependency on a migration in `authors`.

This means that when you run the migrations, the `authors` migration runs first and creates the table the `ForeignKey` references, and then the migration that makes the `ForeignKey` column runs afterward and creates the constraint. If this didn't happen, the migration would try to create the `ForeignKey` column without the table it's referencing existing and your database would throw an error.

This dependency behavior affects most migration operations where you restrict to a single app. Restricting to a single app (either in `makemigrations` or `migrate`) is a best-efforts promise, and not a guarantee; any other apps that need to be used to get dependencies correct will be.

Apps without migrations must not have relations (`ForeignKey`, `ManyToManyField`, etc.) to apps with migrations. Sometimes it may work, but it's not supported.

Swappable dependencies

`django.db.migrations.swappable_dependency(value)`

The `swappable_dependency()` function is used in migrations to declare "swappable" dependencies on migrations in the app of the swapped-in model, currently, on the first migration of this app. As a consequence, the swapped-in model should be created in the initial migration. The argument `value` is a string `"<app label>.<model>"` describing an app label and a model name, e.g. `"myapp.MyModel"`.

By using `swappable_dependency()`, you inform the migration framework that the migration relies on another migration which sets up a swappable model, allowing for the possibility of substituting the model with a different implementation in the future. This is typically used for referencing models that are subject to customization or replacement, such as the custom user model (`settings.AUTH_USER_MODEL`, which defaults to `"auth.User"`) in Django's authentication system.

3.7.6 Migration files

Migrations are stored as an on-disk format, referred to here as "migration files". These files are actually normal Python files with an agreed-upon object layout, written in a declarative style.

A basic migration file looks like this:

```
from django.db import migrations, models


class Migration(migrations.Migration):
    dependencies = [("migrations", "0001_initial")]

    operations = [
        migrations.DeleteModel("Tribble"),
        migrations.AddField("Author", "rating", models.IntegerField(default=0)),
    ]
```

What Django looks for when it loads a migration file (as a Python module) is a subclass of `django.db.migrations.Migration` called `Migration`. It then inspects this object for four attributes, only two of which are used most of the time:

- `dependencies`, a list of migrations this one depends on.
- `operations`, a list of `Operation` classes that define what this migration does.

The operations are the key; they are a set of declarative instructions which tell Django what schema changes need to be made. Django scans them and builds an in-memory representation of all of the schema changes to all apps, and uses this to generate the SQL which makes the schema changes.

That in-memory structure is also used to work out what the differences are between your models and the current state of your migrations; Django runs through all the changes, in order, on an in-memory set of models to come up with the state of your models last time you ran `makemigrations`. It then uses these models to compare against the ones in your `models.py` files to work out what you have changed.

You should rarely, if ever, need to edit migration files by hand, but it's entirely possible to write them manually if you need to. Some of the more complex operations are not autodetectable and are only available via a hand-written migration, so don't be scared about editing them if you have to.

Custom fields

You can't modify the number of positional arguments in an already migrated custom field without raising a `TypeError`. The old migration will call the modified `__init__` method with the old signature. So if you need a new argument, please create a keyword argument and add something like `assert 'argument_name' in kwargs` in the constructor.

Model managers

You can optionally serialize managers into migrations and have them available in *RunPython* operations. This is done by defining a `use_in_migrations` attribute on the manager class:

```
class MyManager(models.Manager):
    use_in_migrations = True


class MyModel(models.Model):
    objects = MyManager()
```

If you are using the *from_queryset()* function to dynamically generate a manager class, you need to inherit from the generated class to make it importable:

```
class MyManager(MyBaseManager.from_queryset(CustomQuerySet)):
    use_in_migrations = True


class MyModel(models.Model):
    objects = MyManager()
```

Please refer to the notes about Historical models in migrations to see the implications that come along.

Initial migrations

`Migration.initial`

The "initial migrations" for an app are the migrations that create the first version of that app's tables. Usually an app will have one initial migration, but in some cases of complex model interdependencies it may have two or more.

Initial migrations are marked with an `initial = True` class attribute on the migration class. If an `initial` class attribute isn't found, a migration will be considered "initial" if it is the first migration in the app (i.e. if it has no dependencies on any other migration in the same app).

When the *migrate --fake-initial* option is used, these initial migrations are treated specially. For an initial migration that creates one or more tables (`CreateModel` operation), Django checks that all of those tables already exist in the database and fake-applies the migration if so. Similarly, for an initial migration that adds one or more fields (`AddField` operation), Django checks that all of the respective columns already exist in the database and fake-applies the migration if so. Without `--fake-initial`, initial migrations are treated no differently from any other migration.

History consistency

As previously discussed, you may need to linearize migrations manually when two development branches are joined. While editing migration dependencies, you can inadvertently create an inconsistent history state where a migration has been applied but some of its dependencies haven't. This is a strong indication that the dependencies are incorrect, so Django will refuse to run migrations or make new migrations until it's fixed. When using multiple databases, you can use the *`allow_migrate()`* method of database routers to control which databases *`makemigrations`* checks for consistent history.

3.7.7 Adding migrations to apps

New apps come preconfigured to accept migrations, and so you can add migrations by running *`makemigrations`* once you've made some changes.

If your app already has models and database tables, and doesn't have migrations yet (for example, you created it against a previous Django version), you'll need to convert it to use migrations by running:

```
$ python manage.py makemigrations your_app_label
```

This will make a new initial migration for your app. Now, run `python manage.py migrate --fake-initial`, and Django will detect that you have an initial migration and that the tables it wants to create already exist, and will mark the migration as already applied. (Without the *`migrate --fake-initial`* flag, the command would error out because the tables it wants to create already exist.)

Note that this only works given two things:

- You have not changed your models since you made their tables. For migrations to work, you must make the initial migration first and then make changes, as Django compares changes against migration files, not the database.
- You have not manually edited your database - Django won't be able to detect that your database doesn't match your models, you'll just get errors when migrations try to modify those tables.

3.7.8 Reversing migrations

Migrations can be reversed with *`migrate`* by passing the number of the previous migration. For example, to reverse migration `books.0003`:

```
$ python manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto... OK
```

If you want to reverse all migrations applied for an app, use the name `zero`:

```
$ python manage.py migrate books zero
Operations to perform:
  Unapply all migrations: books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0002_auto... OK
  Unapplying books.0001_initial... OK
```

A migration is irreversible if it contains any irreversible operations. Attempting to reverse such migrations will raise `IrreversibleError`:

```
$ python manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto...Traceback (most recent call last):
django.db.migrations.exceptions.IrreversibleError: Operation <RunSQL  sql='DROP TABLE␣
↪demo_books'> in books.0003_auto is not reversible
```

3.7.9 Historical models

When you run migrations, Django is working from historical versions of your models stored in the migration files. If you write Python code using the *RunPython* operation, or if you have `allow_migrate` methods on your database routers, you need to use these historical model versions rather than importing them directly.

> **Warning**
>
> If you import models directly rather than using the historical models, your migrations may work initially but will fail in the future when you try to rerun old migrations (commonly, when you set up a new installation and run through all the migrations to set up the database).
>
> This means that historical model problems may not be immediately obvious. If you run into this kind of failure, it's OK to edit the migration to use the historical models rather than direct imports and commit those changes.

Because it's impossible to serialize arbitrary Python code, these historical models will not have any custom methods that you have defined. They will, however, have the same fields, relationships, managers (limited to those with `use_in_migrations = True`) and `Meta` options (also versioned, so they may be different from your current ones).

> **Warning**
>
> This means that you will NOT have custom `save()` methods called on objects when you access them in migrations, and you will NOT have any custom constructors or instance methods. Plan appropriately!

References to functions in field options such as `upload_to` and `limit_choices_to` and model manager declarations with managers having `use_in_migrations = True` are serialized in migrations, so the functions and classes will need to be kept around for as long as there is a migration referencing them. Any custom model fields will also need to be kept, since these are imported directly by migrations.

In addition, the concrete base classes of the model are stored as pointers, so you must always keep base classes around for as long as there is a migration that contains a reference to them. On the plus side, methods and managers from these base classes inherit normally, so if you absolutely need access to these you can opt to move them into a superclass.

To remove old references, you can squash migrations or, if there aren't many references, copy them into the migration files.

3.7.10 Considerations when removing model fields

Similar to the "references to historical functions" considerations described in the previous section, removing custom model fields from your project or third-party app will cause a problem if they are referenced in old migrations.

To help with this situation, Django provides some model field attributes to assist with model field deprecation using the system checks framework.

Add the `system_check_deprecated_details` attribute to your model field similar to the following:

```
class IPAddressField(Field):
    system_check_deprecated_details = {
        "msg": (
            "IPAddressField has been deprecated. Support for it (except "
            "in historical migrations) will be removed in Django 1.9."
        ),
        "hint": "Use GenericIPAddressField instead.",  # optional
        "id": "fields.W900",  # pick a unique ID for your field.
    }
```

After a deprecation period of your choosing (two or three feature releases for fields in Django itself), change the `system_check_deprecated_details` attribute to `system_check_removed_details` and update the dictionary similar to:

```
class IPAddressField(Field):
    system_check_removed_details = {
        "msg": (
            "IPAddressField has been removed except for support in "
            "historical migrations."
        ),
        "hint": "Use GenericIPAddressField instead.",
        "id": "fields.E900",  # pick a unique ID for your field.
    }
```

You should keep the field's methods that are required for it to operate in database migrations such as `__init__()`, `deconstruct()`, and `get_internal_type()`. Keep this stub field for as long as any migrations which reference the field exist. For example, after squashing migrations and removing the old ones, you should be able to remove the field completely.

3.7.11 Data Migrations

As well as changing the database schema, you can also use migrations to change the data in the database itself, in conjunction with the schema if you want.

Migrations that alter data are usually called "data migrations"; they're best written as separate migrations, sitting alongside your schema migrations.

Django can't automatically generate data migrations for you, as it does with schema migrations, but it's not very hard to write them. Migration files in Django are made up of Operations, and the main operation you use for data migrations is *RunPython*.

To start, make an empty migration file you can work from (Django will put the file in the right place, suggest a name, and add dependencies for you):

```
python manage.py makemigrations --empty yourappname
```

Then, open up the file; it should look something like this:

```
# Generated by Django A.B on YYYY-MM-DD HH:MM
from django.db import migrations


class Migration(migrations.Migration):
    dependencies = [
        ("yourappname", "0001_initial"),
    ]

    operations = []
```

Now, all you need to do is create a new function and have *RunPython* use it. *RunPython* expects a callable as its argument which takes two arguments - the first is an app registry that has the historical versions of all your models loaded into it to match where in your history the migration sits, and the second is a SchemaEditor, which you can use to manually effect database schema changes (but beware, doing this can confuse the migration autodetector!)

Let's write a migration that populates our new `name` field with the combined values of `first_name` and `last_name` (we've come to our senses and realized that not everyone has first and last names). All we need to do is use the historical model and iterate over the rows:

```
from django.db import migrations


def combine_names(apps, schema_editor):
    # We can't import the Person model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Person = apps.get_model("yourappname", "Person")
    for person in Person.objects.all():
        person.name = f"{person.first_name} {person.last_name}"
        person.save()


class Migration(migrations.Migration):
    dependencies = [
        ("yourappname", "0001_initial"),
    ]

    operations = [
        migrations.RunPython(combine_names),
    ]
```

Once that's done, we can run `python manage.py migrate` as normal and the data migration will run in place alongside other migrations.

You can pass a second callable to *RunPython* to run whatever logic you want executed when migrating backwards. If this callable is omitted, migrating backwards will raise an exception.

Accessing models from other apps

When writing a `RunPython` function that uses models from apps other than the one in which the migration is located, the migration's `dependencies` attribute should include the latest migration of each app that is involved, otherwise you may get an error similar to: `LookupError: No installed app with label 'myappname'` when you try to retrieve the model in the `RunPython` function using `apps.get_model()`.

In the following example, we have a migration in `app1` which needs to use models in `app2`. We aren't con-

cerned with the details of `move_m1` other than the fact it will need to access models from both apps. Therefore we've added a dependency that specifies the last migration of `app2`:

```
class Migration(migrations.Migration):
    dependencies = [
        ("app1", "0001_initial"),
        # added dependency to enable using models from app2 in move_m1
        ("app2", "0004_foobar"),
    ]

    operations = [
        migrations.RunPython(move_m1),
    ]
```

More advanced migrations

If you're interested in the more advanced migration operations, or want to be able to write your own, see the migration operations reference and the "how-to" on writing migrations.

3.7.12 Squashing migrations

You are encouraged to make migrations freely and not worry about how many you have; the migration code is optimized to deal with hundreds at a time without much slowdown. However, eventually you will want to move back from having several hundred migrations to just a few, and that's where squashing comes in.

Squashing is the act of reducing an existing set of many migrations down to one (or sometimes a few) migrations which still represent the same changes.

Django does this by taking all of your existing migrations, extracting their `Operation`s and putting them all in sequence, and then running an optimizer over them to try and reduce the length of the list - for example, it knows that *`CreateModel`* and *`DeleteModel`* cancel each other out, and it knows that *`AddField`* can be rolled into *`CreateModel`*.

Once the operation sequence has been reduced as much as possible - the amount possible depends on how closely intertwined your models are and if you have any *`RunSQL`* or *`RunPython`* operations (which can't be optimized through unless they are marked as `elidable`) - Django will then write it back out into a new set of migration files.

These files are marked to say they replace the previously-squashed migrations, so they can coexist with the old migration files, and Django will intelligently switch between them depending where you are in the history. If you're still part-way through the set of migrations that you squashed, it will keep using them until it hits the end and then switch to the squashed history, while new installs will use the new squashed migration and skip all the old ones.

This enables you to squash and not mess up systems currently in production that aren't fully up-to-date yet. The recommended process is to squash, keeping the old files, commit and release, wait until all systems are

upgraded with the new release (or if you're a third-party project, ensure your users upgrade releases in order without skipping any), and then remove the old files, commit and do a second release.

The command that backs all this is *squashmigrations* - pass it the app label and migration name you want to squash up to, and it'll get to work:

```
$ ./manage.py squashmigrations myapp 0004
Will squash the following migrations:
 - 0001_initial
 - 0002_some_change
 - 0003_another_change
 - 0004_undo_something
Do you wish to proceed? [y/N] y
Optimizing...
  Optimized from 12 operations to 7 operations.
Created new squashed migration /home/andrew/Programs/DjangoTest/test/migrations/0001_
↪squashed_0004_undo_something.py
  You should commit this migration but leave the old ones in place;
  the new migration will be used for new installs. Once you are sure
  all instances of the codebase have applied the migrations you squashed,
  you can delete them.
```

Use the *squashmigrations --squashed-name* option if you want to set the name of the squashed migration rather than use an autogenerated one.

Note that model interdependencies in Django can get very complex, and squashing may result in migrations that do not run; either mis-optimized (in which case you can try again with `--no-optimize`, though you should also report an issue), or with a `CircularDependencyError`, in which case you can manually resolve it.

To manually resolve a `CircularDependencyError`, break out one of the ForeignKeys in the circular dependency loop into a separate migration, and move the dependency on the other app with it. If you're unsure, see how *makemigrations* deals with the problem when asked to create brand new migrations from your models. In a future release of Django, *squashmigrations* will be updated to attempt to resolve these errors itself.

Once you've squashed your migration, you should then commit it alongside the migrations it replaces and distribute this change to all running instances of your application, making sure that they run `migrate` to store the change in their database.

You must then transition the squashed migration to a normal migration by:

- Deleting all the migration files it replaces.
- Updating all migrations that depend on the deleted migrations to depend on the squashed migration instead.
- Removing the `replaces` attribute in the `Migration` class of the squashed migration (this is how Django

tells that it is a squashed migration).

> **Note**
>
> Once you've squashed a migration, you should not then re-squash that squashed migration until you have fully transitioned it to a normal migration.

> **Pruning references to deleted migrations**
>
> If it is likely that you may reuse the name of a deleted migration in the future, you should remove references to it from Django's migrations table with the *`migrate --prune`* option.

3.7.13 Serializing values

Migrations are Python files containing the old definitions of your models - thus, to write them, Django must take the current state of your models and serialize them out into a file.

While Django can serialize most things, there are some things that we just can't serialize out into a valid Python representation - there's no Python standard for how a value can be turned back into code (`repr()` only works for basic values, and doesn't specify import paths).

Django can serialize the following:

- `int`, `float`, `bool`, `str`, `bytes`, `None`, `NoneType`
- `list`, `set`, `tuple`, `dict`, `range`.
- `datetime.date`, `datetime.time`, and `datetime.datetime` instances (include those that are timezone-aware)
- `decimal.Decimal` instances
- `enum.Enum` and `enum.Flag` instances
- `uuid.UUID` instances
- `functools.partial()` and `functools.partialmethod` instances which have serializable `func`, `args`, and `keywords` values.
- Pure and concrete path objects from `pathlib`. Concrete paths are converted to their pure path equivalent, e.g. `pathlib.PosixPath` to `pathlib.PurePosixPath`.
- `os.PathLike` instances, e.g. `os.DirEntry`, which are converted to `str` or `bytes` using `os.fspath()`.
- `LazyObject` instances which wrap a serializable value.
- Enumeration types (e.g. `TextChoices` or `IntegerChoices`) instances.
- Any Django field

- Any function or method reference (e.g. `datetime.datetime.today`) (must be in module's top-level scope)
  - Functions may be decorated if wrapped properly, i.e. using `functools.wraps()`
  - The `functools.cache()` and `functools.lru_cache()` decorators are explicitly supported
- Unbound methods used from within the class body
- Any class reference (must be in module's top-level scope)
- Anything with a custom `deconstruct()` method (see below)

Django cannot serialize:

- Nested classes
- Arbitrary class instances (e.g. `MyClass(4.3, 5.7)`)
- Lambdas

Custom serializers

You can serialize other types by writing a custom serializer. For example, if Django didn't serialize `Decimal` by default, you could do this:

```
from decimal import Decimal

from django.db.migrations.serializer import BaseSerializer
from django.db.migrations.writer import MigrationWriter


class DecimalSerializer(BaseSerializer):
    def serialize(self):
        return repr(self.value), {"from decimal import Decimal"}


MigrationWriter.register_serializer(Decimal, DecimalSerializer)
```

The first argument of `MigrationWriter.register_serializer()` is a type or iterable of types that should use the serializer.

The `serialize()` method of your serializer must return a string of how the value should appear in migrations and a set of any imports that are needed in the migration.

Adding a `deconstruct()` method

You can let Django serialize your own custom class instances by giving the class a `deconstruct()` method. It takes no arguments, and should return a tuple of three things `(path, args, kwargs)`:

- `path` should be the Python path to the class, with the class name included as the last part (for example, `myapp.custom_things.MyClass`). If your class is not available at the top level of a module it is not serializable.
- `args` should be a list of positional arguments to pass to your class' `__init__` method. Everything in this list should itself be serializable.
- `kwargs` should be a dict of keyword arguments to pass to your class' `__init__` method. Every value should itself be serializable.

> **Note**
>
> This return value is different from the `deconstruct()` method for custom fields which returns a tuple of four items.

Django will write out the value as an instantiation of your class with the given arguments, similar to the way it writes out references to Django fields.

To prevent a new migration from being created each time *makemigrations* is run, you should also add a `__eq__()` method to the decorated class. This function will be called by Django's migration framework to detect changes between states.

As long as all of the arguments to your class' constructor are themselves serializable, you can use the `@deconstructible` class decorator from `django.utils.deconstruct` to add the `deconstruct()` method:

```
from django.utils.deconstruct import deconstructible


@deconstructible
class MyCustomClass:
    def __init__(self, foo=1):
        self.foo = foo
        ...

    def __eq__(self, other):
        return self.foo == other.foo
```

The decorator adds logic to capture and preserve the arguments on their way into your constructor, and then returns those arguments exactly when deconstruct() is called.

3.7.14 Supporting multiple Django versions

If you are the maintainer of a third-party app with models, you may need to ship migrations that support multiple Django versions. In this case, you should always run *makemigrations* with the lowest Django version you wish to support.

The migrations system will maintain backwards-compatibility according to the same policy as the rest of Django, so migration files generated on Django X.Y should run unchanged on Django X.Y+1. The migrations system does not promise forwards-compatibility, however. New features may be added, and migration files generated with newer versions of Django may not work on older versions.

See also

The Migrations Operations Reference
: Covers the schema operations API, special operations, and writing your own operations.

The Writing Migrations "how-to"
: Explains how to structure and write database migrations for different scenarios you might encounter.

3.8 Managing files

This document describes Django's file access APIs for files such as those uploaded by a user. The lower level APIs are general enough that you could use them for other purposes. If you want to handle "static files" (JS, CSS, etc.), see How to manage static files (e.g. images, JavaScript, CSS).

By default, Django stores files locally, using the *MEDIA_ROOT* and *MEDIA_URL* settings. The examples below assume that you're using these defaults.

However, Django provides ways to write custom file storage systems that allow you to completely customize where and how Django stores files. The second half of this document describes how these storage systems work.

3.8.1 Using files in models

When you use a *FileField* or *ImageField*, Django provides a set of APIs you can use to deal with that file.

Consider the following model, using an *ImageField* to store a photo:

```
from django.db import models


class Car(models.Model):
    name = models.CharField(max_length=255)
```

(continues on next page)

(continued from previous page)

```
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to="cars")
    specs = models.FileField(upload_to="specs")
```

Any `Car` instance will have a `photo` attribute that you can use to get at the details of the attached photo:

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: cars/chevy.jpg>
>>> car.photo.name
'cars/chevy.jpg'
>>> car.photo.path
'/media/cars/chevy.jpg'
>>> car.photo.url
'https://media.example.com/cars/chevy.jpg'
```

This object – `car.photo` in the example – is a `File` object, which means it has all the methods and attributes described below.

> **Note**
>
> The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

For example, you can change the file name by setting the file's *name* to a path relative to the file storage's location (*MEDIA_ROOT* if you are using the default *FileSystemStorage*):

```
>>> import os
>>> from django.conf import settings
>>> initial_path = car.photo.path
>>> car.photo.name = "cars/chevy_ii.jpg"
>>> new_path = settings.MEDIA_ROOT + car.photo.name
>>> # Move the file on the filesystem
>>> os.rename(initial_path, new_path)
>>> car.save()
>>> car.photo.path
'/media/cars/chevy_ii.jpg'
>>> car.photo.path == new_path
True
```

To save an existing file on disk to a *FileField*:

```
>>> from pathlib import Path
>>> from django.core.files import File
>>> path = Path("/some/external/specs.pdf")
>>> car = Car.objects.get(name="57 Chevy")
>>> with path.open(mode="rb") as f:
...     car.specs = File(f, name=path.name)
...     car.save()
...
```

Note

While *ImageField* non-image data attributes, such as `height`, `width`, and `size` are available on the instance, the underlying image data cannot be used without reopening the image. For example:

```
>>> from PIL import Image
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo.width
191
>>> car.photo.height
287
>>> image = Image.open(car.photo)
# Raises ValueError: seek of closed file.
>>> car.photo.open()
<ImageFieldFile: cars/chevy.jpg>
>>> image = Image.open(car.photo)
>>> image
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=191x287 at 0x7F99A94E9048>
```

3.8.2 The `File` object

Internally, Django uses a *django.core.files.File* instance any time it needs to represent a file.

Most of the time you'll use a `File` that Django's given you (i.e. a file attached to a model as above, or perhaps an uploaded file).

If you need to construct a `File` yourself, the easiest way is to create one using a Python built-in `file` object:

```
>>> from django.core.files import File

# Create a Python file object using open()
>>> f = open("/path/to/hello.world", "w")
>>> myfile = File(f)
```

Now you can use any of the documented attributes and methods of the *File* class.

Be aware that files created in this way are not automatically closed. The following approach may be used to close files automatically:

```
>>> from django.core.files import File

# Create a Python file object using open() and the with statement
>>> with open("/path/to/hello.world", "w") as f:
...     myfile = File(f)
...     myfile.write("Hello World")
...
>>> myfile.closed
True
>>> f.closed
True
```

Closing files is especially important when accessing file fields in a loop over a large number of objects. If files are not manually closed after accessing them, the risk of running out of file descriptors may arise. This may lead to the following error:

```
OSError: [Errno 24] Too many open files
```

3.8.3 File storage

Behind the scenes, Django delegates decisions about how and where to store files to a file storage system. This is the object that actually understands things like file systems, opening and reading files, etc.

Django's default file storage is '*django.core.files.storage.FileSystemStorage*'. If you don't explicitly provide a storage system in the `default` key of the *STORAGES* setting, this is the one that will be used.

See below for details of the built-in default file storage system, and see How to write a custom storage class for information on writing your own file storage system.

Storage objects

Though most of the time you'll want to use a `File` object (which delegates to the proper storage for that file), you can use file storage systems directly. You can create an instance of some custom file storage class, or – often more useful – you can use the global default storage system:

```
>>> from django.core.files.base import ContentFile
>>> from django.core.files.storage import default_storage

>>> path = default_storage.save("path/to/file", ContentFile(b"new content"))
```

(continues on next page)

(continued from previous page)

```
>>> path
'path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
b'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

See File storage API for the file storage API.

The built-in filesystem storage class

Django ships with a *django.core.files.storage.FileSystemStorage* class which implements basic local filesystem file storage.

For example, the following code will store uploaded files under `/media/photos` regardless of what your *MEDIA_ROOT* setting is:

```
from django.core.files.storage import FileSystemStorage
from django.db import models

fs = FileSystemStorage(location="/media/photos")


class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

Custom storage systems work the same way: you can pass them in as the `storage` argument to a *FileField*.

Using a callable

You can use a callable as the *storage* parameter for *FileField* or *ImageField*. This allows you to modify the used storage at runtime, selecting different storages for different environments, for example.

Your callable will be evaluated when your models classes are loaded, and must return an instance of *Storage*.

For example:

```
from django.conf import settings
from django.db import models
from .storages import MyLocalStorage, MyRemoteStorage


def select_storage():
    return MyLocalStorage() if settings.DEBUG else MyRemoteStorage()


class MyModel(models.Model):
    my_file = models.FileField(storage=select_storage)
```

In order to set a storage defined in the *STORAGES* setting you can use *storages*:

```
from django.core.files.storage import storages


def select_storage():
    return storages["mystorage"]


class MyModel(models.Model):
    upload = models.FileField(storage=select_storage)
```

Because the callable is evaluated when your models classes are loaded, if you need to override the *STORAGES* setting in tests, you should use a `LazyObject` subclass instead:

```
from django.core.files.storage import storages
from django.utils.functional import LazyObject


class OtherStorage(LazyObject):
    def _setup(self):
        self._wrapped = storages["mystorage"]


my_storage = OtherStorage()


class MyModel(models.Model):
    upload = models.FileField(storage=my_storage)
```

The `LazyObject` delays the evaluation of the storage until it's actually needed, allowing *`override_settings()`* to take effect:

```
@override_settings(
    STORAGES={
        "mystorage": {
            "BACKEND": "django.core.files.storage.InMemoryStorage",
        }
    }
)
def test_storage():
    model = MyModel()
    assert isinstance(model.upload.storage, InMemoryStorage)
```

3.9 Testing in Django

Automated testing is an extremely useful bug-killing tool for the modern web developer. You can use a collection of tests – a test suite – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code works as expected.
- When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly.

Testing a web application is a complex task, because a web application is made of several layers of logic – from HTTP-level request handling, to form validation and processing, to template rendering. With Django's test-execution framework and assorted utilities, you can simulate requests, insert test data, inspect your application's output and generally verify your code is doing what it should be doing.

The preferred way to write tests in Django is using the `unittest` module built-in to the Python standard library. This is covered in detail in the Writing and running tests document.

You can also use any other Python test framework; Django provides an API and tools for that kind of integration. They are described in the Using different testing frameworks section of Advanced testing topics.

3.9.1 Writing and running tests

> See also
>
> The testing tutorial, the testing tools reference, and the advanced testing topics.

This document is split into two primary sections. First, we explain how to write tests with Django. Then, we explain how to run them.

Writing tests

Django's unit tests use a Python standard library module: `unittest`. This module defines tests using a class-based approach.

Here is an example which subclasses from *`django.test.TestCase`*, which is a subclass of `unittest.TestCase` that runs each test inside a transaction to provide isolation:

```
from django.test import TestCase
from myapp.models import Animal


class AnimalTestCase(TestCase):
    def setUp(self):
        Animal.objects.create(name="lion", sound="roar")
        Animal.objects.create(name="cat", sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        lion = Animal.objects.get(name="lion")
        cat = Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(), 'The lion says "roar"')
        self.assertEqual(cat.speak(), 'The cat says "meow"')
```

When you run your tests, the default behavior of the test utility is to find all the test case classes (that is, subclasses of `unittest.TestCase`) in any file whose name begins with `test`, automatically build a test suite out of those test case classes, and run that suite.

For more details about `unittest`, see the Python documentation.

> **Where should the tests live?**
>
> The default *`startapp`* template creates a `tests.py` file in the new application. This might be fine if you only have a few tests, but as your test suite grows you'll likely want to restructure it into a tests package so you can split your tests into different submodules such as `test_models.py`, `test_views.py`, `test_forms.py`, etc. Feel free to pick whatever organizational scheme you like.
>
> See also Using the Django test runner to test reusable applications.

> **Warning**
>
> If your tests rely on database access such as creating or querying models, be sure to create your test classes as subclasses of *`django.test.TestCase`* rather than `unittest.TestCase`.

Using `unittest.TestCase` avoids the cost of running each test in a transaction and flushing the database, but if your tests interact with the database their behavior will vary based on the order that the test runner executes them. This can lead to unit tests that pass when run in isolation but fail when run in a suite.

Running tests

Once you've written tests, run them using the *`test`* command of your project's `manage.py` utility:

```
$ ./manage.py test
```

Test discovery is based on the unittest module's built-in test discovery. By default, this will discover tests in any file named `test*.py` under the current working directory.

You can specify particular tests to run by supplying any number of "test labels" to `./manage.py test`. Each test label can be a full Python dotted path to a package, module, `TestCase` subclass, or test method. For instance:

```
# Run all the tests in the animals.tests module
$ ./manage.py test animals.tests

# Run all the tests found within the 'animals' package
$ ./manage.py test animals

# Run just one test case class
$ ./manage.py test animals.tests.AnimalTestCase

# Run just one test method
$ ./manage.py test animals.tests.AnimalTestCase.test_animals_can_speak
```

You can also provide a path to a directory to discover tests below that directory:

```
$ ./manage.py test animals/
```

You can specify a custom filename pattern match using the `-p` (or `--pattern`) option, if your test files are named differently from the `test*.py` pattern:

```
$ ./manage.py test --pattern="tests_*.py"
```

If you press `Ctrl-C` while the tests are running, the test runner will wait for the currently running test to complete and then exit gracefully. During a graceful exit the test runner will output details of any test failures, report on how many tests were run and how many errors and failures were encountered, and destroy any test databases as usual. Thus pressing `Ctrl-C` can be very useful if you forget to pass the *`--failfast`* option, notice that some tests are unexpectedly failing and want to get details on the failures without waiting for the full test run to complete.

If you do not want to wait for the currently running test to finish, you can press `Ctrl-C` a second time and the test run will halt immediately, but not gracefully. No details of the tests run before the interruption will be reported, and any test databases created by the run will not be destroyed.

> **Test with warnings enabled**
>
> It's a good idea to run your tests with Python warnings enabled: `python -Wa manage.py test`. The `-Wa` flag tells Python to display deprecation warnings. Django, like many other Python libraries, uses these warnings to flag when features are going away. It also might flag areas in your code that aren't strictly wrong but could benefit from a better implementation.

The test database

Tests that require a database (namely, model tests) will not use your "real" (production) database. Separate, blank databases are created for the tests.

Regardless of whether the tests pass or fail, the test databases are destroyed when all the tests have been executed.

You can prevent the test databases from being destroyed by using the *`test --keepdb`* option. This will preserve the test database between runs. If the database does not exist, it will first be created. Any migrations will also be applied in order to keep it up to date.

As described in the previous section, if a test run is forcefully interrupted, the test database may not be destroyed. On the next run, you'll be asked whether you want to reuse or destroy the database. Use the *`test --noinput`* option to suppress that prompt and automatically destroy the database. This can be useful when running tests on a continuous integration server where tests may be interrupted by a timeout, for example.

The default test database names are created by prepending `test_` to the value of each *`NAME`* in *`DATABASES`*. When using SQLite, the tests will use an in-memory database by default (i.e., the database will be created in memory, bypassing the filesystem entirely!). The *`TEST`* dictionary in *`DATABASES`* offers a number of settings to configure your test database. For example, if you want to use a different database name, specify *`NAME`* in the *`TEST`* dictionary for any given database in *`DATABASES`*.

On PostgreSQL, *`USER`* will also need read access to the built-in `postgres` database.

Aside from using a separate database, the test runner will otherwise use all of the same database settings you have in your settings file: *`ENGINE`*, *`USER`*, *`HOST`*, etc. The test database is created by the user specified by *`USER`*, so you'll need to make sure that the given user account has sufficient privileges to create a new database on the system.

For fine-grained control over the character encoding of your test database, use the *`CHARSET`* TEST option. If you're using MySQL, you can also use the *`COLLATION`* option to control the particular collation used by the test database. See the settings documentation for details of these and other advanced settings.

If using an SQLite in-memory database with SQLite, shared cache is enabled, so you can write tests with ability to share the database between threads.

Finding data from your production database when running tests?

If your code attempts to access the database when its modules are compiled, this will occur before the test database is set up, with potentially unexpected results. For example, if you have a database query in module-level code and a real database exists, production data could pollute your tests. It is a bad idea to have such import-time database queries in your code anyway - rewrite your code so that it doesn't do this.

This also applies to customized implementations of *ready()*.

See also

The advanced multi-db testing topics.

Order in which tests are executed

In order to guarantee that all `TestCase` code starts with a clean database, the Django test runner reorders tests in the following way:

- All *TestCase* subclasses are run first.
- Then, all other Django-based tests (test case classes based on *SimpleTestCase*, including *TransactionTestCase*) are run with no particular ordering guaranteed nor enforced among them.
- Then any other `unittest.TestCase` tests (including doctests) that may alter the database without restoring it to its original state are run.

Note

The new ordering of tests may reveal unexpected dependencies on test case ordering. This is the case with doctests that relied on state left in the database by a given *TransactionTestCase* test, they must be updated to be able to run independently.

Note

Failures detected when loading tests are ordered before all of the above for quicker feedback. This includes things like test modules that couldn't be found or that couldn't be loaded due to syntax errors.

You may randomize and/or reverse the execution order inside groups using the *test --shuffle* and *--reverse* options. This can help with ensuring your tests are independent from each other.

Rollback emulation

Any initial data loaded in migrations will only be available in `TestCase` tests and not in `TransactionTestCase` tests, and additionally only on backends where transactions are supported (the most important exception being MyISAM). This is also true for tests which rely on `TransactionTestCase` such as *`LiveServerTestCase`* and *`StaticLiveServerTestCase`*.

Django can reload that data for you on a per-testcase basis by setting the `serialized_rollback` option to `True` in the body of the `TestCase` or `TransactionTestCase`, but note that this will slow down that test suite by approximately 3x.

Third-party apps or those developing against MyISAM will need to set this; in general, however, you should be developing your own projects against a transactional database and be using `TestCase` for most tests, and thus not need this setting.

The initial serialization is usually very quick, but if you wish to exclude some apps from this process (and speed up test runs slightly), you may add those apps to *`TEST_NON_SERIALIZED_APPS`*.

To prevent serialized data from being loaded twice, setting `serialized_rollback=True` disables the *`post_migrate`* signal when flushing the test database.

For *`TransactionTestCase`*, serialized migration data is made available during `setUpClass()`.

Other test conditions

Regardless of the value of the *`DEBUG`* setting in your configuration file, all Django tests run with *`DEBUG`*=False. This is to ensure that the observed output of your code matches what will be seen in a production setting.

Caches are not cleared after each test, and running `manage.py test fooapp` can insert data from the tests into the cache of a live system if you run your tests in production because, unlike databases, a separate "test cache" is not used. This behavior may change in the future.

Understanding the test output

When you run your tests, you'll see a number of messages as the test runner prepares itself. You can control the level of detail of these messages with the `verbosity` option on the command line:

```
Creating test database...
Creating table myapp_animal
Creating table myapp_mineral
```

This tells you that the test runner is creating a test database, as described in the previous section.

Once the test database has been created, Django will run your tests. If everything goes well, you'll see something like this:

```
----------------------------------------------------------------------
Ran 22 tests in 0.221s

OK
```

If there are test failures, however, you'll see full details about which tests failed:

```
======================================================================
FAIL: test_was_published_recently_with_future_poll (polls.tests.PollMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/dev/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_
↪poll
    self.assertIs(future_poll.was_published_recently(), False)
AssertionError: True is not False

----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (failures=1)
```

A full explanation of this error output is beyond the scope of this document, but it's pretty intuitive. You can consult the documentation of Python's `unittest` library for details.

Note that the return code for the test-runner script is 1 for any number of failed tests (whether the failure was caused by an error, a failed assertion, or an unexpected success). If all the tests pass, the return code is 0. This feature is useful if you're using the test-runner script in a shell script and need to test for success or failure at that level.

Speeding up the tests

Running tests in parallel

As long as your tests are properly isolated, you can run them in parallel to gain a speed up on multi-core hardware. See *`test --parallel`*.

Password hashing

The default password hasher is rather slow by design. If you're authenticating many users in your tests, you may want to use a custom settings file and set the *`PASSWORD_HASHERS`* setting to a faster hashing algorithm:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.MD5PasswordHasher",
```

(continues on next page)

(continued from previous page)

```
]
```

Don't forget to also include in *PASSWORD_HASHERS* any hashing algorithm used in fixtures, if any.

Preserving the test database

The *test --keepdb* option preserves the test database between test runs. It skips the create and destroy actions which can greatly decrease the time to run tests.

Avoiding disk access for media files

The *InMemoryStorage* is a convenient way to prevent disk access for media files. All data is kept in memory, then it gets discarded after tests run.

3.9.2 Testing tools

Django provides a small set of tools that come in handy when writing tests.

The test client

The test client is a Python class that acts as a dummy web browser, allowing you to test your views and interact with your Django-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate GET and POST requests on a URL and observe the response – everything from low-level HTTP (result headers and status codes) to page content.
- See the chain of redirects (if any) and check the URL and status code at each step.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

Note that the test client is not intended to be a replacement for Selenium or other "in-browser" frameworks. Django's test client has a different focus. In short:

- Use Django's test client to establish that the correct template is being rendered and that the template is passed the correct context data.
- Use *RequestFactory* to test view functions directly, bypassing the routing and middleware layers.
- Use in-browser frameworks like Selenium to test rendered HTML and the behavior of web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on *LiveServerTestCase* for more details.

A comprehensive test suite should use a combination of all of these test types.

Overview and a quick example

To use the test client, instantiate `django.test.Client` and retrieve web pages:

```
>>> from django.test import Client
>>> c = Client()
>>> response = c.post("/login/", {"username": "john", "password": "smith"})
>>> response.status_code
200
>>> response = c.get("/customer/details/")
>>> response.content
b'<!DOCTYPE html...'
```

As this example suggests, you can instantiate `Client` from within a session of the Python interactive interpreter.

Note a few important things about how the test client works:

- The test client does not require the web server to be running. In fact, it will run just fine with no web server running at all! That's because it avoids the overhead of HTTP and deals directly with the Django framework. This helps make the unit tests run quickly.
- When retrieving pages, remember to specify the path of the URL, not the whole domain. For example, this is correct:

  ```
  >>> c.get("/login/")
  ```

  This is incorrect:

  ```
  >>> c.get("https://www.example.com/login/")
  ```

  The test client is not capable of retrieving web pages that are not powered by your Django project. If you need to retrieve other web pages, use a Python standard library module such as `urllib`.
- To resolve URLs, the test client uses whatever URLconf is pointed-to by your *ROOT_URLCONF* setting.
- Although the above example would work in the Python interactive interpreter, some of the test client's functionality, notably the template-related functionality, is only available while tests are running.

  The reason for this is that Django's test runner performs a bit of black magic in order to determine which template was loaded by a given view. This black magic (essentially a patching of Django's template system in memory) only happens during test running.
- By default, the test client will disable any CSRF checks performed by your site.

  If, for some reason, you want the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks. To do this, pass in the `enforce_csrf_checks` argument when you construct your client:

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

Making requests

Use the `django.test.Client` class to make requests.

`class Client`(enforce_csrf_checks=False, raise_request_exception=True, json_encoder=DjangoJSONEncoder, * (Keyword-only parameters separator (PEP 3102)), headers=None, query_params=None, **defaults)

A testing HTTP client. Takes several arguments that can customize behavior.

`headers` allows you to specify default headers that will be sent with every request. For example, to set a `User-Agent` header:

```
client = Client(headers={"user-agent": "curl/7.79.1"})
```

`query_params` allows you to specify the default query string that will be set on every request.

Arbitrary keyword arguments in `**defaults` set WSGI environ variables. For example, to set the script name:

```
client = Client(SCRIPT_NAME="/app/")
```

> **Note**
>
> Keyword arguments starting with a `HTTP_` prefix are set as headers, but the `headers` parameter should be preferred for readability.

The values from the `headers`, `query_params`, and `extra` keyword arguments passed to *`get()`*, *`post()`*, etc. have precedence over the defaults passed to the class constructor.

The `enforce_csrf_checks` argument can be used to test CSRF protection (see above).

The `raise_request_exception` argument allows controlling whether or not exceptions raised during the request should also be raised in the test. Defaults to `True`.

The `json_encoder` argument allows setting a custom JSON encoder for the JSON serialization that's described in *`post()`*.

The `query_params` argument was added.

Once you have a `Client` instance, you can call any of the following methods:

`get`(path, data=None, follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a GET request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `query_params` dictionary are used to set query strings. For example:

```
>>> c = Client()
>>> c.get("/customers/details/", query_params={"name": "fred", "age": 7})
```

...will result in the evaluation of a GET request equivalent to:

```
/customers/details/?name=fred&age=7
```

It is also possible to pass these parameters into the `data` parameter. However, `query_params` is preferred as it works for any HTTP method.

The `headers` parameter can be used to specify headers to be sent in the request. For example:

```
>>> c = Client()
>>> c.get(
...     "/customers/details/",
...     query_params={"name": "fred", "age": 7},
...     headers={"accept": "application/json"},
... )
```

...will send the HTTP header `HTTP_ACCEPT` to the details view, which is a good way to test code paths that use the *`django.http.HttpRequest.accepts()`* method.

Arbitrary keyword arguments set WSGI environ variables. For example, headers to set the script name:

```
>>> c = Client()
>>> c.get("/", SCRIPT_NAME="/app/")
```

If you already have the GET arguments in URL-encoded form, you can use that encoding instead of using the data argument. For example, the previous GET request could also be posed as:

```
>>> c = Client()
>>> c.get("/customers/details/?name=fred&age=7")
```

If you provide a URL with both an encoded GET data and either a query_params or data argument these arguments will take precedence.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you had a URL `/redirect_me/` that redirected to `/next/`, that redirected to `/final/`, this is what you'd see:

```
>>> response = c.get("/redirect_me/", follow=True)
>>> response.redirect_chain
[('http://testserver/next/', 302), ('http://testserver/final/', 302)]
```

If you set `secure` to `True` the client will emulate an HTTPS request.

The `query_params` argument was added.

post(path, data=None, content_type=MULTIPART_CONTENT, follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a POST request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `data` dictionary are used to submit POST data. For example:

```
>>> c = Client()
>>> c.post("/login/", {"name": "fred", "passwd": "secret"})
```

...will result in the evaluation of a POST request to this URL:

```
/login/
```

...with this POST data:

```
name=fred&passwd=secret
```

If you provide `content_type` as *application/json*, the `data` is serialized using `json.dumps()` if it's a dict, list, or tuple. Serialization is performed with *DjangoJSONEncoder* by default, and can be overridden by providing a `json_encoder` argument to *Client*. This serialization also happens for *put()*, *patch()*, and *delete()* requests.

If you provide any other `content_type` (e.g. *text/xml* for an XML payload), the contents of `data` are sent as-is in the POST request, using `content_type` in the HTTP `Content-Type` header.

If you don't provide a value for `content_type`, the values in `data` will be transmitted with a content type of *multipart/form-data*. In this case, the key-value pairs in `data` will be encoded as a multipart message and used to create the POST data payload.

To submit multiple values for a given key – for example, to specify the selections for a `<select multiple>` – provide the values as a list or tuple for the required key. For example, this value of `data` would submit three selected values for the field named `choices`:

```
{"choices": ["a", "b", "d"]}
```

Submitting files is a special case. To POST a file, you need only provide the file field name as a key, and a file handle to the file you wish to upload as a value. For example, if your form has fields `name` and `attachment`, the latter a *`FileField`*:

```
>>> c = Client()
>>> with open("wishlist.doc", "rb") as fp:
...     c.post("/customers/wishes/", {"name": "fred", "attachment": fp})
...
```

You may also provide any file-like object (e.g., `StringIO` or `BytesIO`) as a file handle. If you're uploading to an *`ImageField`*, the object needs a `name` attribute that passes the *`validate_image_file_extension`* validator. For example:

```
>>> from io import BytesIO
>>> img = BytesIO(
...     b"GIF89a\x01\x00\x01\x00\x00\x00\x00!\xf9\x04\x01\x00\x00\x00"
...     b"\x00,\x00\x00\x00\x00\x01\x00\x01\x00\x00\x02\x01\x00\x00"
... )
>>> img.name = "myimage.gif"
```

Note that if you wish to use the same file handle for multiple `post()` calls then you will need to manually reset the file pointer between posts. The easiest way to do this is to manually close the file after it has been provided to `post()`, as demonstrated above.

You should also ensure that the file is opened in a way that allows the data to be read. If your file contains binary data such as an image, this means you will need to open the file in `rb` (read binary) mode.

The `headers`, `query_params`, and `extra` parameters acts the same as for *`Client.get()`*.

If the URL you request with a POST contains encoded parameters, these parameters will be made available in the request.GET data. For example, if you were to make the request:

```
>>> c.post(
...     "/login/", {"name": "fred", "passwd": "secret"}, query_params={"visitor
↪": "true"}
... )
```

... the view handling this request could interrogate request.POST to retrieve the username and password, and could interrogate request.GET to determine if the user was a visitor.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you set `secure` to `True` the client will emulate an HTTPS request.

The `query_params` argument was added.

head(path, data=None, follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a HEAD request on the provided `path` and returns a `Response` object. This method works just like *`Client.get()`*, including the `follow`, `secure`, `headers`, `query_params`, and `extra` parameters, except it does not return a message body.

The `query_params` argument was added.

options(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes an OPTIONS request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for *`Client.get()`*.

The `query_params` argument was added.

put(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a PUT request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for *`Client.get()`*.

The `query_params` argument was added.

patch(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a PATCH request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for *`Client.get()`*.

The `query_params` argument was added.

delete(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a DELETE request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for *`Client.get()`*.

The `query_params` argument was added.

`trace`(path, follow=False, secure=False, *, headers=None, query_params=None, **extra)

Makes a TRACE request on the provided `path` and returns a `Response` object. Useful for simulating diagnostic probes.

Unlike the other request methods, `data` is not provided as a keyword parameter in order to comply with RFC 9110 Section 9.3.8, which mandates that TRACE requests must not have a body.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for *`Client.get()`*.

The `query_params` argument was added.

`login`(**credentials)

`alogin`(**credentials)

Asynchronous version: `alogin()`

If your site uses Django's authentication system and you deal with logging in users, you can use the test client's `login()` method to simulate the effect of a user logging into the site.

After you call this method, the test client will have all the cookies and session data required to pass any login-based tests that may form part of a view.

The format of the `credentials` argument depends on which authentication backend you're using (which is configured by your *`AUTHENTICATION_BACKENDS`* setting). If you're using the standard authentication backend provided by Django (`ModelBackend`), `credentials` should be the user's username and password, provided as keyword arguments:

```
>>> c = Client()
>>> c.login(username="fred", password="secret")

# Now you can access a view that's only available to logged-in users.
```

If you're using a different authentication backend, this method may require different credentials. It requires whichever credentials are required by your backend's `authenticate()` method.

`login()` returns `True` if it the credentials were accepted and login was successful.

Finally, you'll need to remember to create user accounts before you can use this method. As we explained above, the test runner is executed using a test database, which contains no users by default. As a result, user accounts that are valid on your production site will not work under test

conditions. You'll need to create users as part of the test suite – either manually (using the Django model API) or with a test fixture. Remember that if you want your test user to have a password, you can't set the user's password by setting the password attribute directly – you must use the *set_password()* function to store a correctly hashed password. Alternatively, you can use the *create_user()* helper method to create a new user with a correctly hashed password.

force_login(user, backend=None)

aforce_login(user, backend=None)

Asynchronous version: `aforce_login()`

If your site uses Django's authentication system, you can use the `force_login()` method to simulate the effect of a user logging into the site. Use this method instead of *login()* when a test requires a user be logged in and the details of how a user logged in aren't important.

Unlike `login()`, this method skips the authentication and verification steps: inactive users (*is_active=False*) are permitted to login and the user's credentials don't need to be provided.

The user will have its `backend` attribute set to the value of the `backend` argument (which should be a dotted Python path string), or to `settings.AUTHENTICATION_BACKENDS[0]` if a value isn't provided. The *authenticate()* function called by *login()* normally annotates the user like this.

This method is faster than `login()` since the expensive password hashing algorithms are bypassed. Also, you can speed up `login()` by using a weaker hasher while testing.

logout()

alogout()

Asynchronous version: `alogout()`

If your site uses Django's authentication system, the `logout()` method can be used to simulate the effect of a user logging out of your site.

After you call this method, the test client will have all the cookies and session data cleared to defaults. Subsequent requests will appear to come from an *AnonymousUser*.

Testing responses

The `get()` and `post()` methods both return a `Response` object. This `Response` object is not the same as the `HttpResponse` object returned by Django views; the test response object has some additional data useful for test code to verify.

Specifically, a `Response` object has the following attributes:

class Response

client

The test client that was used to make the request that resulted in the response.

content

The body of the response, as a bytestring. This is the final page content as rendered by the view, or any error message.

context

The template `Context` instance that was used to render the template that produced the response content.

If the rendered page used multiple templates, then `context` will be a list of `Context` objects, in the order in which they were rendered.

Regardless of the number of templates used during rendering, you can retrieve context values using the `[]` operator. For example, the context variable `name` could be retrieved using:

```
>>> response = client.get("/foo/")
>>> response.context["name"]
'Arthur'
```

> **Not using Django templates?**
>
> This attribute is only populated when using the *DjangoTemplates* backend. If you're using another template engine, *context_data* may be a suitable alternative on responses with that attribute.

exc_info

A tuple of three values that provides information about the unhandled exception, if any, that occurred during the view.

The values are (type, value, traceback), the same as returned by Python's `sys.exc_info()`. Their meanings are:

- type: The type of the exception.
- value: The exception instance.
- traceback: A traceback object which encapsulates the call stack at the point where the exception originally occurred.

If no exception occurred, then `exc_info` will be `None`.

json(**kwargs)

The body of the response, parsed as JSON. Extra keyword arguments are passed to `json.loads()`. For example:

```
>>> response = client.get("/foo/")
>>> response.json()["name"]
'Arthur'
```

If the `Content-Type` header is not `"application/json"`, then a `ValueError` will be raised when trying to parse the response.

request

The request data that stimulated the response.

wsgi_request

The `WSGIRequest` instance generated by the test handler that generated the response.

status_code

The HTTP status of the response, as an integer. For a full list of defined codes, see the IANA status code registry.

templates

A list of `Template` instances used to render the final content, in the order they were rendered. For each template in the list, use `template.name` to get the template's file name, if the template was loaded from a file. (The name is a string such as `'admin/index.html'`.)

> **Not using Django templates?**
>
> This attribute is only populated when using the *DjangoTemplates* backend. If you're using another template engine, *template_name* may be a suitable alternative if you only need the name of the template used for rendering.

resolver_match

An instance of *ResolverMatch* for the response. You can use the *func* attribute, for example, to verify the view that served the response:

```
# my_view here is a function based view.
self.assertEqual(response.resolver_match.func, my_view)

# Class-based views need to compare the view_class, as the
# functions generated by as_view() won't be equal.
self.assertIs(response.resolver_match.func.view_class, MyView)
```

If the given URL is not found, accessing this attribute will raise a *Resolver404* exception.

As with a normal response, you can also access the headers through *HttpResponse.headers*. For example, you could determine the content type of a response using `response.headers['Content-Type']`.

Exceptions

If you point the test client at a view that raises an exception and `Client.raise_request_exception` is `True`, that exception will be visible in the test case. You can then use a standard `try ... except` block or `assertRaises()` to test for exceptions.

The only exceptions that are not visible to the test client are *Http404*, *PermissionDenied*, `SystemExit`, and *SuspiciousOperation*. Django catches these exceptions internally and converts them into the appropriate HTTP response codes. In these cases, you can check `response.status_code` in your test.

If `Client.raise_request_exception` is `False`, the test client will return a 500 response as would be returned to a browser. The response has the attribute *exc_info* to provide information about the unhandled exception.

Persistent state

The test client is stateful. If a response returns a cookie, then that cookie will be stored in the test client and sent with all subsequent `get()` and `post()` requests.

Expiration policies for these cookies are not followed. If you want a cookie to expire, either delete it manually or create a new `Client` instance (which will effectively delete all cookies).

A test client has attributes that store persistent state information. You can access these properties as part of a test condition.

`Client.cookies`

A Python `SimpleCookie` object, containing the current values of all the client cookies. See the documentation of the `http.cookies` module for more.

`Client.session`

A dictionary-like object containing session information. See the session documentation for full details.

To modify the session and then save it, it must be stored in a variable first (because a new `SessionStore` is created every time this property is accessed):

```
def test_something(self):
    session = self.client.session
    session["somekey"] = "test"
    session.save()
```

`Client.asession()`

This is similar to the *session* attribute but it works in async contexts.

Setting the language

When testing applications that support internationalization and localization, you might want to set the language for a test client request. The method for doing so depends on whether or not the *LocaleMiddleware* is enabled.

If the middleware is enabled, the language can be set by creating a cookie with a name of *LANGUAGE_COOKIE_NAME* and a value of the language code:

```
from django.conf import settings


def test_language_using_cookie(self):
    self.client.cookies.load({settings.LANGUAGE_COOKIE_NAME: "fr"})
    response = self.client.get("/")
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

or by including the `Accept-Language` HTTP header in the request:

```
def test_language_using_header(self):
    response = self.client.get("/", headers={"accept-language": "fr"})
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

> **Note**
>
> When using these methods, ensure to reset the active language at the end of each test:
>
> ```
> def tearDown(self):
>     translation.activate(settings.LANGUAGE_CODE)
> ```

More details are in How Django discovers language preference.

If the middleware isn't enabled, the active language may be set using *translation.override()*:

```
from django.utils import translation


def test_language_using_override(self):
    with translation.override("fr"):
        response = self.client.get("/")
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

More details are in Explicitly setting the active language.

Example

The following is a unit test using the test client:

```
import unittest
from django.test import Client


class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs a client.
        self.client = Client()

    def test_details(self):
        # Issue a GET request.
        response = self.client.get("/customer/details/")

        # Check that the response is 200 OK.
        self.assertEqual(response.status_code, 200)

        # Check that the rendered context contains 5 customers.
        self.assertEqual(len(response.context["customers"]), 5)
```

See also

django.test.RequestFactory

Provided test case classes

Normal Python unit test classes extend a base class of `unittest.TestCase`. Django provides a few extensions of this base class:

You can convert a normal `unittest.TestCase` to any of the subclasses: change the base class of your test from `unittest.TestCase` to the subclass. All of the standard Python unit test functionality will be available, and it will be augmented with some useful additions as described in each section below.

SimpleTestCase

class SimpleTestCase

A subclass of `unittest.TestCase` that adds this functionality:

- Some useful assertions like:
  - Checking that a callable *raises a certain exception*.

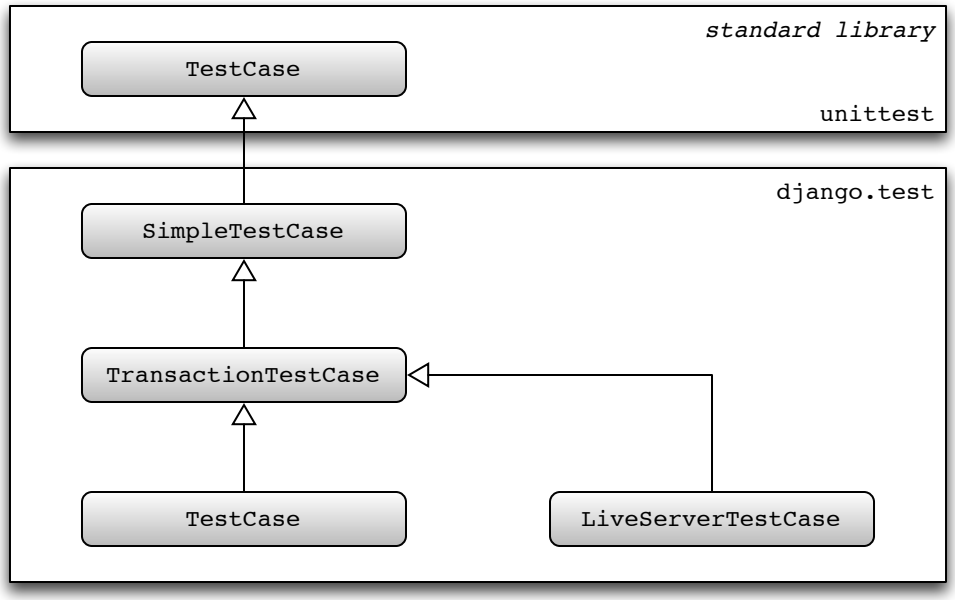

Fig. 1: Hierarchy of Django unit testing classes

  - Checking that a callable *triggers a certain warning*.
  - Testing form field *rendering and error treatment*.
  - Testing *HTML responses for the presence/lack of a given fragment*.
  - Verifying that a template *has/hasn't been used to generate a given response content*.
  - Verifying that two *URLs* are equal.
  - Verifying an HTTP *redirect* is performed by the app.
  - Robustly testing two *HTML fragments* for equality/inequality or *containment*.
  - Robustly testing two *XML fragments* for equality/inequality.
  - Robustly testing two *JSON fragments* for equality.
- The ability to run tests with modified settings.
- Using the *client Client*.

If your tests make any database queries, use subclasses *TransactionTestCase* or *TestCase*.

`SimpleTestCase.databases`

SimpleTestCase disallows database queries by default. This helps to avoid executing write queries which will affect other tests since each `SimpleTestCase` test isn't run in a transaction. If you aren't concerned about this problem, you can disable this behavior by setting the `databases` class attribute to `'__all__'` on your test class.

Warning

`SimpleTestCase` and its subclasses (e.g. `TestCase`, ...) rely on `setUpClass()` and `tearDownClass()` to perform some class-wide initialization (e.g. overriding settings). If you need to override those methods, don't forget to call the `super` implementation:

```
class MyTestCase(TestCase):
    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        ...

    @classmethod
    def tearDownClass(cls):
        ...
        super().tearDownClass()
```

Be sure to account for Python's behavior if an exception is raised during `setUpClass()`. If that happens, neither the tests in the class nor `tearDownClass()` are run. In the case of *`django.test.TestCase`*, this will leak the transaction created in `super()` which results in various symptoms including a segmentation fault on some platforms (reported on macOS). If you want to intentionally raise an exception such as `unittest.SkipTest` in `setUpClass()`, be sure to do it before calling `super()` to avoid this.

TransactionTestCase

class TransactionTestCase

`TransactionTestCase` inherits from *`SimpleTestCase`* to add some database-specific features:

- Resetting the database to a known state at the end of each test to ease testing and using the ORM.
- Database *`fixtures`*.
- Test skipping based on database backend features.
- The remaining specialized *`assert*`* methods.

Django's *`TestCase`* class is a more commonly used subclass of `TransactionTestCase` that makes use of database transaction facilities to speed up the process of resetting the database to a known state at the end of each test. A consequence of this, however, is that some database behaviors cannot be tested within a Django `TestCase` class. For instance, you cannot test that a block of code is executing within a transaction, as is required when using *`select_for_update()`*. In those cases, you should use `TransactionTestCase`.

`TransactionTestCase` and `TestCase` are identical except for the manner in which the database is reset to a known state and the ability for test code to test the effects of commit and rollback:

- A `TransactionTestCase` resets the database after the test runs by truncating all tables. A

`TransactionTestCase` may call commit and rollback and observe the effects of these calls on the database.

- A `TestCase`, on the other hand, does not truncate tables after a test. Instead, it encloses the test code in a database transaction that is rolled back at the end of the test. This guarantees that the rollback at the end of the test restores the database to its initial state.

> **Warning**
>
> `TestCase` running on a database that does not support rollback (e.g. MySQL with the MyISAM storage engine), and all instances of `TransactionTestCase`, will roll back at the end of the test by deleting all data from the test database.
>
> Apps will not see their data reloaded; if you need this functionality (for example, third-party apps should enable this) you can set `serialized_rollback = True` inside the `TestCase` body.

TestCase

class TestCase

This is the most common class to use for writing tests in Django. It inherits from *TransactionTestCase* (and by extension *SimpleTestCase*). If your Django application doesn't use a database, use *SimpleTestCase*.

The class:

- Wraps the tests within two nested *atomic()* blocks: one for the whole class and one for each test. Therefore, if you want to test some specific database transaction behavior, use *TransactionTestCase*.
- Checks deferrable database constraints at the end of each test.

It also provides an additional method:

classmethod TestCase.setUpTestData()

The class-level `atomic` block described above allows the creation of initial data at the class level, once for the whole `TestCase`. This technique allows for faster tests as compared to using `setUp()`.

For example:

```
from django.test import TestCase


class MyTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up data for the whole TestCase
        cls.foo = Foo.objects.create(bar="Test")
```

(continues on next page)

(continued from previous page)

```
        ...

    def test1(self):
        # Some test using self.foo
        ...

    def test2(self):
        # Some other test using self.foo
        ...
```

Note that if the tests are run on a database with no transaction support (for instance, MySQL with the MyISAM engine), `setUpTestData()` will be called before each test, negating the speed benefits.

Objects assigned to class attributes in `setUpTestData()` must support creating deep copies with `copy.deepcopy()` in order to isolate them from alterations performed by each test methods.

classmethod TestCase.**captureOnCommitCallbacks**(using=DEFAULT_DB_ALIAS, execute=False)

Returns a context manager that captures *`transaction.on_commit()`* callbacks for the given database connection. It returns a list that contains, on exit of the context, the captured callback functions. From this list you can make assertions on the callbacks or call them to invoke their side effects, emulating a commit.

`using` is the alias of the database connection to capture callbacks for.

If `execute` is `True`, all the callbacks will be called as the context manager exits, if no exception occurred. This emulates a commit after the wrapped block of code.

For example:

```
from django.core import mail
from django.test import TestCase


class ContactTests(TestCase):
    def test_post(self):
        with self.captureOnCommitCallbacks(execute=True) as callbacks:
            response = self.client.post(
                "/contact/",
                {"message": "I like your site"},
            )

        self.assertEqual(response.status_code, 200)
        self.assertEqual(len(callbacks), 1)
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(len(mail.outbox), 1)
        self.assertEqual(mail.outbox[0].subject, "Contact Form")
        self.assertEqual(mail.outbox[0].body, "I like your site")
```

LiveServerTestCase

class LiveServerTestCase

`LiveServerTestCase` does basically the same as *`TransactionTestCase`* with one extra feature: it launches a live Django server in the background on setup, and shuts it down on teardown. This allows the use of automated test clients other than the Django dummy client such as, for example, the Selenium client, to execute a series of functional tests inside a browser and simulate a real user's actions.

The live server listens on `localhost` and binds to port 0 which uses a free port assigned by the operating system. The server's URL can be accessed with `self.live_server_url` during the tests.

To demonstrate how to use `LiveServerTestCase`, let's write a Selenium test. First of all, you need to install the selenium package:

```
$ python -m pip install "selenium >= 4.8.0"
```

Then, add a `LiveServerTestCase`-based test to your app's tests module (for example: `myapp/tests.py`). For this example, we'll assume you're using the *`staticfiles`* app and want to have static files served during the execution of your tests similar to what we get at development time with `DEBUG=True`, i.e. without having to collect them using *`collectstatic`*. We'll use the *`StaticLiveServerTestCase`* subclass which provides that functionality. Replace it with `django.test.LiveServerTestCase` if you don't need that.

The code for this test may look as follows:

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium.webdriver.common.by import By
from selenium.webdriver.firefox.webdriver import WebDriver


class MySeleniumTests(StaticLiveServerTestCase):
    fixtures = ["user-data.json"]

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls.selenium = WebDriver()
        cls.selenium.implicitly_wait(10)
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super().tearDownClass()

    def test_login(self):
        self.selenium.get(f"{self.live_server_url}/login/")
        username_input = self.selenium.find_element(By.NAME, "username")
        username_input.send_keys("myuser")
        password_input = self.selenium.find_element(By.NAME, "password")
        password_input.send_keys("secret")
        self.selenium.find_element(By.XPATH, '//input[@value="Log in"]').click()
```

Finally, you may run the test as follows:

```
$ ./manage.py test myapp.tests.MySeleniumTests.test_login
```

This example will automatically open Firefox then go to the login page, enter the credentials and press the "Log in" button. Selenium offers other drivers in case you do not have Firefox installed or wish to use another browser. The example above is just a tiny fraction of what the Selenium client can do; check out the full reference for more details.

Note

When using an in-memory SQLite database to run the tests, the same database connection will be shared by two threads in parallel: the thread in which the live server is run and the thread in which the test case is run. It's important to prevent simultaneous database queries via this shared connection by the two threads, as that may sometimes randomly cause the tests to fail. So you need to ensure that the two threads don't access the database at the same time. In particular, this means that in some cases (for example, just after clicking a link or submitting a form), you might need to check that a response is received by Selenium and that the next page is loaded before proceeding with further test execution. Do this, for example, by making Selenium wait until the `<body>` HTML tag is found in the response (requires Selenium > 2.13):

```
def test_login(self):
    from selenium.webdriver.support.wait import WebDriverWait

    timeout = 2
    ...
    self.selenium.find_element(By.XPATH, '//input[@value="Log in"]').click()
    # Wait until the response is received
    WebDriverWait(self.selenium, timeout).until(
        lambda driver: driver.find_element(By.TAG_NAME, "body")
    )
```

The tricky thing here is that there's really no such thing as a "page load," especially in modern web apps that generate HTML dynamically after the server generates the initial document. So, checking for the presence of `<body>` in the response might not necessarily be appropriate for all use cases. Please refer to the Selenium FAQ and Selenium documentation for more information.

Test cases features

Default test client

`SimpleTestCase.client`

Every test case in a `django.test.*TestCase` instance has access to an instance of a Django test client. This client can be accessed as `self.client`. This client is recreated for each test, so you don't have to worry about state (such as cookies) carrying over from one test to another.

This means, instead of instantiating a `Client` in each test:

```
import unittest
from django.test import Client


class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get("/customer/details/")
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get("/customer/index/")
        self.assertEqual(response.status_code, 200)
```

...you can refer to `self.client`, like so:

```
from django.test import TestCase


class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get("/customer/details/")
        self.assertEqual(response.status_code, 200)

    def test_index(self):
```

(continues on next page)

(continued from previous page)

```
        response = self.client.get("/customer/index/")
        self.assertEqual(response.status_code, 200)
```

Customizing the test client

`SimpleTestCase.client_class`

If you want to use a different `Client` class (for example, a subclass with customized behavior), use the *`client_class`* class attribute:

```
from django.test import Client, TestCase


class MyTestClient(Client):
    # Specialized methods for your environment
    ...


class MyTest(TestCase):
    client_class = MyTestClient

    def test_my_stuff(self):
        # Here self.client is an instance of MyTestClient...
        call_some_test_code()
```

Fixture loading

`TransactionTestCase.fixtures`

A test case class for a database-backed website isn't much use if there isn't any data in the database. Tests are more readable and it's more maintainable to create objects using the ORM, for example in *`TestCase.setUpTestData()`*, however, you can also use fixtures.

A fixture is a collection of data that Django knows how to import into a database. For example, if your site has user accounts, you might set up a fixture of fake user accounts in order to populate your database during tests.

The most straightforward way of creating a fixture is to use the *`manage.py dumpdata`* command. This assumes you already have some data in your database. See the *`dumpdata documentation`* for more details.

Once you've created a fixture and placed it in a `fixtures` directory in one of your *`INSTALLED_APPS`*, you can use it in your unit tests by specifying a `fixtures` class attribute on your *`django.test.TestCase`* subclass:

```
from django.test import TestCase
from myapp.models import Animal


class AnimalTestCase(TestCase):
    fixtures = ["mammals.json", "birds"]

    def setUp(self):
        # Test definitions as before.
        call_setup_methods()

    def test_fluffy_animals(self):
        # A test that uses the fixtures.
        call_some_test_code()
```

Here's specifically what will happen:

- During `setUpClass()`, all the named fixtures are installed. In this example, Django will install any JSON fixture named `mammals`, followed by any fixture named `birds`. See the Fixtures topic for more details on defining and installing fixtures.

For most unit tests using *`TestCase`*, Django doesn't need to do anything else, because transactions are used to clean the database after each test for performance reasons. But for *`TransactionTestCase`*, the following actions will take place:

- At the end of each test Django will flush the database, returning the database to the state it was in directly after *`migrate`* was called.
- For each subsequent test, the fixtures will be reloaded before `setUp()` is run.

In any case, you can be certain that the outcome of a test will not be affected by another test or by the order of test execution.

By default, fixtures are only loaded into the `default` database. If you are using multiple databases and set *`TransactionTestCase.databases`*, fixtures will be loaded into all specified databases.

For *`TransactionTestCase`*, fixtures were made available during `setUpClass()`.

URLconf configuration

If your application provides views, you may want to include tests that use the test client to exercise those views. However, an end user is free to deploy the views in your application at any URL of their choosing. This means that your tests can't rely upon the fact that your views will be available at a particular URL. Decorate your test class or test method with `@override_settings(ROOT_URLCONF=...)` for URLconf configuration.

Multi-database support

`TransactionTestCase.databases`

Django sets up a test database corresponding to every database that is defined in the *DATABASES* definition in your settings and referred to by at least one test through `databases`.

However, a big part of the time taken to run a Django `TestCase` is consumed by the call to `flush` that ensures that you have a clean database at the end of each test run. If you have multiple databases, multiple flushes are required (one for each database), which can be a time consuming activity – especially if your tests don't need to test multi-database activity.

As an optimization, Django only flushes the `default` database at the end of each test run. If your setup contains multiple databases, and you have a test that requires every database to be clean, you can use the `databases` attribute on the test suite to request extra databases to be flushed.

For example:

```
class TestMyViews(TransactionTestCase):
    databases = {"default", "other"}

    def test_index_page_view(self):
        call_some_test_code()
```

This test case class will flush the `default` and `other` test databases after running `test_index_page_view`. You can also use `'__all__'` to specify that all of the test databases must be flushed.

The `databases` flag also controls which databases the *TransactionTestCase.fixtures* are loaded into. By default, fixtures are only loaded into the `default` database.

Queries against databases not in `databases` will give assertion errors to prevent state leaking between tests.

`TestCase.databases`

By default, only the `default` database will be wrapped in a transaction during a `TestCase`'s execution and attempts to query other databases will result in assertion errors to prevent state leaking between tests.

Use the `databases` class attribute on the test class to request transaction wrapping against non-`default` databases.

For example:

```
class OtherDBTests(TestCase):
    databases = {"other"}

    def test_other_db_query(self): ...
```

This test will only allow queries against the `other` database. Just like for *SimpleTestCase.databases* and

TransactionTestCase.databases, the `'__all__'` constant can be used to specify that the test should allow queries to all databases.

Overriding settings

> **Warning**
>
> Use the functions below to temporarily alter the value of settings in tests. Don't manipulate `django.conf.settings` directly as Django won't restore the original values after such manipulations.

SimpleTestCase.settings()

For testing purposes it's often useful to change a setting temporarily and revert to the original value after running the testing code. For this use case Django provides a standard Python context manager (see PEP 343) called *settings()*, which can be used like this:

```
from django.test import TestCase


class LoginTestCase(TestCase):
    def test_login(self):
        # First check for the default behavior
        response = self.client.get("/sekrit/")
        self.assertRedirects(response, "/accounts/login/?next=/sekrit/")

        # Then override the LOGIN_URL setting
        with self.settings(LOGIN_URL="/other/login/"):
            response = self.client.get("/sekrit/")
            self.assertRedirects(response, "/other/login/?next=/sekrit/")
```

This example will override the *LOGIN_URL* setting for the code in the `with` block and reset its value to the previous state afterward.

SimpleTestCase.modify_settings()

It can prove unwieldy to redefine settings that contain a list of values. In practice, adding or removing values is often sufficient. Django provides the *modify_settings()* context manager for easier settings changes:

```
from django.test import TestCase


class MiddlewareTestCase(TestCase):
    def test_cache_middleware(self):
```

(continues on next page)

(continued from previous page)

```
        with self.modify_settings(
            MIDDLEWARE={
                "append": "django.middleware.cache.FetchFromCacheMiddleware",
                "prepend": "django.middleware.cache.UpdateCacheMiddleware",
                "remove": [
                    "django.contrib.sessions.middleware.SessionMiddleware",
                    "django.contrib.auth.middleware.AuthenticationMiddleware",
                    "django.contrib.messages.middleware.MessageMiddleware",
                ],
            }
        ):
            response = self.client.get("/")
            # ...
```

For each action, you can supply either a list of values or a string. When the value already exists in the list, `append` and `prepend` have no effect; neither does `remove` when the value doesn't exist.

override_settings(**kwargs)

In case you want to override a setting for a test method, Django provides the *override_settings()* decorator (see PEP 318). It's used like this:

```
from django.test import TestCase, override_settings


class LoginTestCase(TestCase):
    @override_settings(LOGIN_URL="/other/login/")
    def test_login(self):
        response = self.client.get("/sekrit/")
        self.assertRedirects(response, "/other/login/?next=/sekrit/")
```

The decorator can also be applied to *TestCase* classes:

```
from django.test import TestCase, override_settings


@override_settings(LOGIN_URL="/other/login/")
class LoginTestCase(TestCase):
    def test_login(self):
        response = self.client.get("/sekrit/")
        self.assertRedirects(response, "/other/login/?next=/sekrit/")
```

modify_settings(*args, **kwargs)

Likewise, Django provides the *modify_settings()* decorator:

```
from django.test import TestCase, modify_settings


class MiddlewareTestCase(TestCase):
    @modify_settings(
        MIDDLEWARE={
            "append": "django.middleware.cache.FetchFromCacheMiddleware",
            "prepend": "django.middleware.cache.UpdateCacheMiddleware",
        }
    )
    def test_cache_middleware(self):
        response = self.client.get("/")
        # ...
```

The decorator can also be applied to test case classes:

```
from django.test import TestCase, modify_settings


@modify_settings(
    MIDDLEWARE={
        "append": "django.middleware.cache.FetchFromCacheMiddleware",
        "prepend": "django.middleware.cache.UpdateCacheMiddleware",
    }
)
class MiddlewareTestCase(TestCase):
    def test_cache_middleware(self):
        response = self.client.get("/")
        # ...
```

Note

When given a class, these decorators modify the class directly and return it; they don't create and return a modified copy of it. So if you try to tweak the above examples to assign the return value to a different name than `LoginTestCase` or `MiddlewareTestCase`, you may be surprised to find that the original test case classes are still equally affected by the decorator. For a given class, *modify_settings()* is always applied after *override_settings()*.

> **Warning**
>
> The settings file contains some settings that are only consulted during initialization of Django internals. If you change them with `override_settings`, the setting is changed if you access it via the `django.conf.settings` module, however, Django's internals access it differently. Effectively, using *`override_settings()`* or *`modify_settings()`* with these settings is probably not going to do what you expect it to do.
>
> We do not recommend altering the *`DATABASES`* setting. Altering the *`CACHES`* setting is possible, but a bit tricky if you are using internals that make using of caching, like *`django.contrib.sessions`*. For example, you will have to reinitialize the session backend in a test that uses cached sessions and overrides *`CACHES`*.
>
> Finally, avoid aliasing your settings as module-level constants as `override_settings()` won't work on such values since they are only evaluated the first time the module is imported.

You can also simulate the absence of a setting by deleting it after settings have been overridden, like this:

```
@override_settings()
def test_something(self):
    del settings.LOGIN_URL
    ...
```

When overriding settings, make sure to handle the cases in which your app's code uses a cache or similar feature that retains state even if the setting is changed. Django provides the *`django.test.signals.setting_changed`* signal that lets you register callbacks to clean up and otherwise reset state when settings are changed.

Django itself uses this signal to reset various data:

Overridden settings	Data reset
USE_TZ, TIME_ZONE	Databases timezone
TEMPLATES	Template engines
FORM_RENDERER	Default renderer
SERIALIZATION_MODULES	Serializers cache
LOCALE_PATHS, LANGUAGE_CODE	Default translation and loaded translations
STATIC_ROOT, STATIC_URL, STORAGES	Storages configuration

Resetting the default renderer when the `FORM_RENDERER` setting is changed was added.

Isolating apps

utils.isolate_apps(*app_labels, attr_name=None, kwarg_name=None)

Registers the models defined within a wrapped context into their own isolated *apps* registry. This functionality is useful when creating model classes for tests, as the classes will be cleanly deleted afterward, and there is no risk of name collisions.

The app labels which the isolated registry should contain must be passed as individual arguments. You can use `isolate_apps()` as a decorator or a context manager. For example:

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps


class MyModelTests(SimpleTestCase):
    @isolate_apps("app_label")
    def test_model_definition(self):
        class TestModel(models.Model):
            pass

        ...
```

... or:

```
with isolate_apps("app_label"):

    class TestModel(models.Model):
        pass

    ...
```

The decorator form can also be applied to classes.

Two optional keyword arguments can be specified:

- `attr_name`: attribute assigned the isolated registry if used as a class decorator.
- `kwarg_name`: keyword argument passing the isolated registry if used as a function decorator.

The temporary `Apps` instance used to isolate model registration can be retrieved as an attribute when used as a class decorator by using the `attr_name` parameter:

```
@isolate_apps("app_label", attr_name="apps")
class TestModelDefinition(SimpleTestCase):
```

(continues on next page)

(continued from previous page)

```
    def test_model_definition(self):
        class TestModel(models.Model):
            pass

        self.assertIs(self.apps.get_model("app_label", "TestModel"), TestModel)
```

... or alternatively as an argument on the test method when used as a method decorator by using the `kwarg_name` parameter:

```
class TestModelDefinition(SimpleTestCase):
    @isolate_apps("app_label", kwarg_name="apps")
    def test_model_definition(self, apps):
        class TestModel(models.Model):
            pass

        self.assertIs(apps.get_model("app_label", "TestModel"), TestModel)
```

Emptying the test outbox

If you use any of Django's custom `TestCase` classes, the test runner will clear the contents of the test email outbox at the start of each test case.

For more detail on email services during tests, see Email services below.

Assertions

As Python's normal `unittest.TestCase` class implements assertion methods such as `assertTrue()` and `assertEqual()`, Django's custom *`TestCase`* class provides a number of custom assertion methods that are useful for testing web applications:

The failure messages given by most of these assertion methods can be customized with the `msg_prefix` argument. This string will be prefixed to any failure message generated by the assertion. This allows you to provide additional details that may help you to identify the location and cause of a failure in your test suite.

`SimpleTestCase.assertRaisesMessage`(expected_exception, expected_message, callable, *args, **kwargs)

`SimpleTestCase.assertRaisesMessage`(expected_exception, expected_message)

Asserts that execution of `callable` raises `expected_exception` and that `expected_message` is found in the exception's message. Any other outcome is reported as a failure. It's a simpler version of `unittest.TestCase.assertRaisesRegex()` with the difference that `expected_message` isn't treated as a regular expression.

If only the `expected_exception` and `expected_message` parameters are given, returns a context manager so that the code being tested can be written inline rather than as a function:

```
with self.assertRaisesMessage(ValueError, "invalid literal for int()"):
    int("a")
```

SimpleTestCase.**assertWarnsMessage**(expected_warning, expected_message, callable, *args, **kwargs)

SimpleTestCase.**assertWarnsMessage**(expected_warning, expected_message)

Analogous to *SimpleTestCase.assertRaisesMessage()* but for `assertWarnsRegex()` instead of `assertRaisesRegex()`.

SimpleTestCase.**assertFieldOutput**(fieldclass, valid, invalid, field_args=None, field_kwargs=None, empty_value='')

Asserts that a form field behaves correctly with various inputs.

Parameters

- **fieldclass** – the class of the field to be tested.
- **valid** – a dictionary mapping valid inputs to their expected cleaned values.
- **invalid** – a dictionary mapping invalid inputs to one or more raised error messages.
- **field_args** – the args passed to instantiate the field.
- **field_kwargs** – the kwargs passed to instantiate the field.
- **empty_value** – the expected clean output for inputs in `empty_values`.

For example, the following code tests that an `EmailField` accepts `a@a.com` as a valid email address, but rejects `aaa` with a reasonable error message:

```
self.assertFieldOutput(
    EmailField, {"a@a.com": "a@a.com"}, {"aaa": ["Enter a valid email address."]}
)
```

SimpleTestCase.**assertFormError**(form, field, errors, msg_prefix='')

Asserts that a field on a form raises the provided list of errors.

`form` is a `Form` instance. The form must be bound but not necessarily validated (`assertFormError()` will automatically call `full_clean()` on the form).

`field` is the name of the field on the form to check. To check the form's *non-field errors*, use `field=None`.

`errors` is a list of all the error strings that the field is expected to have. You can also pass a single error string if you only expect one error which means that `errors='error message'` is the same as `errors=['error message']`.

SimpleTestCase.**assertFormSetError**(formset, form_index, field, errors, msg_prefix='')

Asserts that the `formset` raises the provided list of errors when rendered.

`formset` is a `FormSet` instance. The formset must be bound but not necessarily validated (`assertFormSetError()` will automatically call the `full_clean()` on the formset).

`form_index` is the number of the form within the `FormSet` (starting from 0). Use `form_index=None` to check the formset's non-form errors, i.e. the errors you get when calling `formset.non_form_errors()`. In that case you must also use `field=None`.

`field` and `errors` have the same meaning as the parameters to `assertFormError()`.

SimpleTestCase.**assertContains**(response, text, count=None, status_code=200, msg_prefix='', html=False)

Asserts that a *response* produced the given *status_code* and that `text` appears in its *content*. If `count` is provided, `text` must occur exactly `count` times in the response.

Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See *assertHTMLEqual()* for more details.

In older versions, error messages didn't contain the response content.

SimpleTestCase.**assertNotContains**(response, text, status_code=200, msg_prefix='', html=False)

Asserts that a *response* produced the given *status_code* and that `text` does not appear in its *content*.

Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See *assertHTMLEqual()* for more details.

In older versions, error messages didn't contain the response content.

SimpleTestCase.**assertTemplateUsed**(response, template_name, msg_prefix='', count=None)

Asserts that the template with the given name was used in rendering the response.

`response` must be a response instance returned by the *test client*.

`template_name` should be a string such as `'admin/index.html'`.

The `count` argument is an integer indicating the number of times the template should be rendered. Default is `None`, meaning that the template should be rendered one or more times.

You can use this as a context manager, like this:

```
with self.assertTemplateUsed("index.html"):
    render_to_string("index.html")
with self.assertTemplateUsed(template_name="index.html"):
    render_to_string("index.html")
```

`SimpleTestCase.assertTemplateNotUsed`(response, template_name, msg_prefix='')

Asserts that the template with the given name was not used in rendering the response.

You can use this as a context manager in the same way as *assertTemplateUsed()*.

`SimpleTestCase.assertURLEqual`(url1, url2, msg_prefix='')

Asserts that two URLs are the same, ignoring the order of query string parameters except for parameters with the same name. For example, `/path/?x=1&y=2` is equal to `/path/?y=2&x=1`, but `/path/?a=1&a=2` isn't equal to `/path/?a=2&a=1`.

`SimpleTestCase.assertRedirects`(response, expected_url, status_code=302, target_status_code=200, msg_prefix='', fetch_redirect_response=True)

Asserts that the *response* returned a *status_code* redirect status, redirected to `expected_url` (including any `GET` data), and that the final page was received with `target_status_code`.

If your request used the `follow` argument, the `expected_url` and `target_status_code` will be the url and status code for the final point of the redirect chain.

If `fetch_redirect_response` is `False`, the final page won't be loaded. Since the test client can't fetch external URLs, this is particularly useful if `expected_url` isn't part of your Django app.

Scheme is handled correctly when making comparisons between two URLs. If there isn't any scheme specified in the location where we are redirected to, the original request's scheme is used. If present, the scheme in `expected_url` is the one used to make the comparisons to.

`SimpleTestCase.assertHTMLEqual`(html1, html2, msg=None)

Asserts that the strings `html1` and `html2` are equal. The comparison is based on HTML semantics. The comparison takes following things into account:

- Whitespace before and after HTML tags is ignored.
- All types of whitespace are considered equivalent.
- All open tags are closed implicitly, e.g. when a surrounding tag is closed or the HTML document ends.
- Empty tags are equivalent to their self-closing version.
- The ordering of attributes of an HTML element is not significant.
- Boolean attributes (like `checked`) without an argument are equal to attributes that equal in name and value (see the examples).
- Text, character references, and entity references that refer to the same character are equivalent.

The following examples are valid tests and don't raise any `AssertionError`:

```
self.assertHTMLEqual(
    "<p>Hello <b>&#x27;world&#x27;!</p>",
    """<p>
```

(continues on next page)

(continued from previous page)

```
        Hello   <b>&#39;world&#39;! </b>
    </p>""",
)
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_terms" />',
    '<input id="id_accept_terms" type="checkbox" checked>',
)
```

`html1` and `html2` must contain HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertHTMLNotEqual(html1, html2, msg=None)`

Asserts that the strings `html1` and `html2` are not equal. The comparison is based on HTML semantics. See *assertHTMLEqual()* for details.

`html1` and `html2` must contain HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertXMLEqual(xml1, xml2, msg=None)`

Asserts that the strings `xml1` and `xml2` are equal. The comparison is based on XML semantics. Similarly to *assertHTMLEqual()*, the comparison is made on parsed content, hence only semantic differences are considered, not syntax differences. When invalid XML is passed in any parameter, an `AssertionError` is always raised, even if both strings are identical.

XML declaration, document type, processing instructions, and comments are ignored. Only the root element and its children are compared.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertXMLNotEqual(xml1, xml2, msg=None)`

Asserts that the strings `xml1` and `xml2` are not equal. The comparison is based on XML semantics. See *assertXMLEqual()* for details.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertInHTML(needle, haystack, count=None, msg_prefix='')`

Asserts that the HTML fragment `needle` is contained in the `haystack` once.

If the `count` integer argument is specified, then additionally the number of `needle` occurrences will be strictly verified.

Whitespace in most cases is ignored, and attribute ordering is not significant. See *assertHTMLEqual()* for more details.

In older versions, error messages didn't contain the `haystack`.

`SimpleTestCase.assertNotInHTML`(needle, haystack, msg_prefix='')

Asserts that the HTML fragment `needle` is not contained in the `haystack`.

Whitespace in most cases is ignored, and attribute ordering is not significant. See *assertHTMLEqual()* for more details.

`SimpleTestCase.assertJSONEqual`(raw, expected_data, msg=None)

Asserts that the JSON fragments `raw` and `expected_data` are equal. Usual JSON non-significant whitespace rules apply as the heavyweight is delegated to the `json` library.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertJSONNotEqual`(raw, expected_data, msg=None)

Asserts that the JSON fragments `raw` and `expected_data` are not equal. See *assertJSONEqual()* for further details.

Output in case of error can be customized with the `msg` argument.

`TransactionTestCase.assertQuerySetEqual`(qs, values, transform=None, ordered=True, msg=None)

Asserts that a queryset `qs` matches a particular iterable of values `values`.

If `transform` is provided, `values` is compared to a list produced by applying `transform` to each member of `qs`.

By default, the comparison is also ordering dependent. If `qs` doesn't provide an implicit ordering, you can set the `ordered` parameter to `False`, which turns the comparison into a `collections.Counter` comparison. If the order is undefined (if the given `qs` isn't ordered and the comparison is against more than one ordered value), a `ValueError` is raised.

Output in case of error can be customized with the `msg` argument.

`TransactionTestCase.assertNumQueries`(num, func, *args, **kwargs)

Asserts that when `func` is called with `*args` and `**kwargs` that `num` database queries are executed.

If a `"using"` key is present in `kwargs` it is used as the database alias for which to check the number of queries:

```
self.assertNumQueries(7, using="non_default_db")
```

If you wish to call a function with a `using` parameter you can do it by wrapping the call with a `lambda` to add an extra parameter:

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

You can also use this as a context manager:

```
with self.assertNumQueries(2):
    Person.objects.create(name="Aaron")
    Person.objects.create(name="Daniel")
```

Tagging tests

You can tag your tests so you can easily run a particular subset. For example, you might label fast or slow tests:

```
from django.test import tag


class SampleTestCase(TestCase):
    @tag("fast")
    def test_fast(self): ...

    @tag("slow")
    def test_slow(self): ...

    @tag("slow", "core")
    def test_slow_but_core(self): ...
```

You can also tag a test case class:

```
@tag("slow", "core")
class SampleTestCase(TestCase): ...
```

Subclasses inherit tags from superclasses, and methods inherit tags from their class. Given:

```
@tag("foo")
class SampleTestCaseChild(SampleTestCase):
    @tag("bar")
    def test(self): ...
```

`SampleTestCaseChild.test` will be labeled with `'slow'`, `'core'`, `'bar'`, and `'foo'`.

Then you can choose which tests to run. For example, to run only fast tests:

```
$ ./manage.py test --tag=fast
```

Or to run fast tests and the core one (even though it's slow):

```
$ ./manage.py test --tag=fast --tag=core
```

You can also exclude tests by tag. To run core tests if they are not slow:

```
$ ./manage.py test --tag=core --exclude-tag=slow
```

test --exclude-tag has precedence over *test --tag*, so if a test has two tags and you select one of them and exclude the other, the test won't be run.

Testing asynchronous code

If you merely want to test the output of your asynchronous views, the standard test client will run them inside their own asynchronous loop without any extra work needed on your part.

However, if you want to write fully-asynchronous tests for a Django project, you will need to take several things into account.

Firstly, your tests must be `async def` methods on the test class (in order to give them an asynchronous context). Django will automatically detect any `async def` tests and wrap them so they run in their own event loop.

If you are testing from an asynchronous function, you must also use the asynchronous test client. This is available as `django.test.AsyncClient`, or as `self.async_client` on any test.

class AsyncClient(enforce_csrf_checks=False, raise_request_exception=True, *, headers=None, query_params=None, **defaults)

`AsyncClient` has the same methods and signatures as the synchronous (normal) test client, with the following exceptions:

- In the initialization, arbitrary keyword arguments in `defaults` are added directly into the ASGI scope.
- Headers passed as `extra` keyword arguments should not have the `HTTP_` prefix required by the synchronous client (see *Client.get()*). For example, here is how to set an HTTP `Accept` header:

```
>>> c = AsyncClient()
>>> c.get("/customers/details/", {"name": "fred", "age": 7}, ACCEPT="application/
↪json")
```

The `query_params` argument was added.

Using `AsyncClient` any method that makes a request must be awaited:

```
async def test_my_thing(self):
    response = await self.async_client.get("/some-url/")
    self.assertEqual(response.status_code, 200)
```

The asynchronous client can also call synchronous views; it runs through Django's asynchronous request path, which supports both. Any view called through the `AsyncClient` will get an `ASGIRequest` object for its `request` rather than the `WSGIRequest` that the normal client creates.

> **Warning**
>
> If you are using test decorators, they must be async-compatible to ensure they work correctly. Django's built-in decorators will behave correctly, but third-party ones may appear to not execute (they will "wrap" the wrong part of the execution flow and not your test).
>
> If you need to use these decorators, then you should decorate your test methods with *async_to_sync()* inside of them instead:

```
from asgiref.sync import async_to_sync
from django.test import TestCase



class MyTests(TestCase):
    @mock.patch(...)
    @async_to_sync
    async def test_my_thing(self): ...
```

Email services

If any of your Django views send email using Django's email functionality, you probably don't want to send email each time you run a test using that view. For this reason, Django's test runner automatically redirects all Django-sent email to a dummy outbox. This lets you test every aspect of sending email – from the number of messages sent to the contents of each message – without actually sending the messages.

The test runner accomplishes this by transparently replacing the normal email backend with a testing backend. (Don't worry – this has no effect on any other email senders outside of Django, such as your machine's mail server, if you're running one.)

`django.core.mail.outbox`

During test running, each outgoing email is saved in `django.core.mail.outbox`. This is a list of all *EmailMessage* instances that have been sent. The `outbox` attribute is a special attribute that is created only when the `locmem` email backend is used. It doesn't normally exist as part of the *django.core.mail* module and you can't import it directly. The code below shows how to access this attribute correctly.

Here's an example test that examines `django.core.mail.outbox` for length and contents:

```
from django.core import mail
from django.test import TestCase

```

(continues on next page)

(continued from previous page)

```
class EmailTest(TestCase):
    def test_send_email(self):
        # Send message.
        mail.send_mail(
            "Subject here",
            "Here is the message.",
            "from@example.com",
            ["to@example.com"],
            fail_silently=False,
        )

        # Test that one message has been sent.
        self.assertEqual(len(mail.outbox), 1)

        # Verify that the subject of the first message is correct.
        self.assertEqual(mail.outbox[0].subject, "Subject here")
```

As noted previously, the test outbox is emptied at the start of every test in a Django `*TestCase`. To empty the outbox manually, assign the empty list to `mail.outbox`:

```
from django.core import mail

# Empty the test outbox
mail.outbox = []
```

Management Commands

Management commands can be tested with the *`call_command()`* function. The output can be redirected into a `StringIO` instance:

```
from io import StringIO
from django.core.management import call_command
from django.test import TestCase


class ClosepollTest(TestCase):
    def test_command_output(self):
        out = StringIO()
        call_command("closepoll", poll_ids=[1], stdout=out)
        self.assertIn('Successfully closed poll "1"', out.getvalue())
```

Skipping tests

The unittest library provides the `@skipIf` and `@skipUnless` decorators to allow you to skip tests if you know ahead of time that those tests are going to fail under certain conditions.

For example, if your test requires a particular optional library in order to succeed, you could decorate the test case with `@skipIf`. Then, the test runner will report that the test wasn't executed and why, instead of failing the test or omitting the test altogether.

To supplement these test skipping behaviors, Django provides two additional skip decorators. Instead of testing a generic boolean, these decorators check the capabilities of the database, and skip the test if the database doesn't support a specific named feature.

The decorators use a string identifier to describe database features. This string corresponds to attributes of the database connection features class. See django.db.backends.base.features.BaseDatabaseFeatures class for a full list of database features that can be used as a basis for skipping tests.

skipIfDBFeature(*feature_name_strings)

Skip the decorated test or `TestCase` if all of the named database features are supported.

For example, the following test will not be executed if the database supports transactions (e.g., it would not run under PostgreSQL, but it would under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipIfDBFeature("supports_transactions")
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

skipUnlessDBFeature(*feature_name_strings)

Skip the decorated test or `TestCase` if any of the named database features are not supported.

For example, the following test will only be executed if the database supports transactions (e.g., it would run under PostgreSQL, but not under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipUnlessDBFeature("supports_transactions")
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

3.9.3 Advanced testing topics

The request factory

class RequestFactory

The *RequestFactory* shares the same API as the test client. However, instead of behaving like a browser, the RequestFactory provides a way to generate a request instance that can be used as the first argument to any view. This means you can test a view function the same way as you would test any other function – as a black box, with exactly known inputs, testing for specific outputs.

The API for the *RequestFactory* is a slightly restricted subset of the test client API:

- It only has access to the HTTP methods *get()*, *post()*, *put()*, *delete()*, *head()*, *options()*, and *trace()*.
- These methods accept all the same arguments except for `follow`. Since this is just a factory for producing requests, it's up to you to handle the response.
- It does not support middleware. Session and authentication attributes must be supplied by the test itself if required for the view to function properly.

The `query_params` parameter was added.

Example

The following is a unit test using the request factory:

```
from django.contrib.auth.models import AnonymousUser, User
from django.test import RequestFactory, TestCase

from .views import MyView, my_view


class SimpleTest(TestCase):
    def setUp(self):
        # Every test needs access to the request factory.
        self.factory = RequestFactory()
        self.user = User.objects.create_user(
            username="jacob", email="jacob@...", password="top_secret"
        )

    def test_details(self):
        # Create an instance of a GET request.
        request = self.factory.get("/customer/details")
```

(continues on next page)

(continued from previous page)

```
        # Recall that middleware are not supported. You can simulate a
        # logged-in user by setting request.user manually.
        request.user = self.user

        # Or you can simulate an anonymous user by setting request.user to
        # an AnonymousUser instance.
        request.user = AnonymousUser()

        # Test my_view() as if it were deployed at /customer/details
        response = my_view(request)
        # Use this syntax for class-based views.
        response = MyView.as_view()(request)
        self.assertEqual(response.status_code, 200)
```

AsyncRequestFactory

class AsyncRequestFactory

`RequestFactory` creates WSGI-like requests. If you want to create ASGI-like requests, including having a correct ASGI `scope`, you can instead use `django.test.AsyncRequestFactory`.

This class is directly API-compatible with `RequestFactory`, with the only difference being that it returns `ASGIRequest` instances rather than `WSGIRequest` instances. All of its methods are still synchronous callables.

Arbitrary keyword arguments in `defaults` are added directly into the ASGI scope.

The `query_params` parameter was added.

Testing class-based views

In order to test class-based views outside of the request/response cycle you must ensure that they are configured correctly, by calling *`setup()`* after instantiation.

For example, assuming the following class-based view:

Listing 24: `views.py`

```
from django.views.generic import TemplateView


class HomeView(TemplateView):
    template_name = "myapp/home.html"

    def get_context_data(self, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        kwargs["environment"] = "Production"
        return super().get_context_data(**kwargs)
```

You may directly test the `get_context_data()` method by first instantiating the view, then passing a `request` to `setup()`, before proceeding with your test's code:

Listing 25: `tests.py`

```
from django.test import RequestFactory, TestCase
from .views import HomeView


class HomePageTest(TestCase):
    def test_environment_set_in_context(self):
        request = RequestFactory().get("/")
        view = HomeView()
        view.setup(request)

        context = view.get_context_data()
        self.assertIn("environment", context)
```

Tests and multiple host names

The *ALLOWED_HOSTS* setting is validated when running tests. This allows the test client to differentiate between internal and external URLs.

Projects that support multitenancy or otherwise alter business logic based on the request's host and use custom host names in tests must include those hosts in *ALLOWED_HOSTS*.

The first option to do so is to add the hosts to your settings file. For example, the test suite for docs.djangoproject.com includes the following:

```
from django.test import TestCase


class SearchFormTestCase(TestCase):
    def test_empty_get(self):
        response = self.client.get(
            "/en/dev/search/",
            headers={"host": "docs.djangoproject.dev:8000"},
        )
        self.assertEqual(response.status_code, 200)
```

and the settings file includes a list of the domains supported by the project:

```
ALLOWED_HOSTS = ["www.djangoproject.dev", "docs.djangoproject.dev", ...]
```

Another option is to add the required hosts to *ALLOWED_HOSTS* using *override_settings()* or *modify_settings()*. This option may be preferable in standalone apps that can't package their own settings file or for projects where the list of domains is not static (e.g., subdomains for multitenancy). For example, you could write a test for the domain `http://otherserver/` as follows:

```
from django.test import TestCase, override_settings


class MultiDomainTestCase(TestCase):
    @override_settings(ALLOWED_HOSTS=["otherserver"])
    def test_other_domain(self):
        response = self.client.get("http://otherserver/foo/bar/")
```

Disabling *ALLOWED_HOSTS* checking (`ALLOWED_HOSTS = ['*']`) when running tests prevents the test client from raising a helpful error message if you follow a redirect to an external URL.

Tests and multiple databases

Testing primary/replica configurations

If you're testing a multiple database configuration with primary/replica (referred to as master/slave by some databases) replication, this strategy of creating test databases poses a problem. When the test databases are created, there won't be any replication, and as a result, data created on the primary won't be seen on the replica.

To compensate for this, Django allows you to define that a database is a test mirror. Consider the following (simplified) example database configuration:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myproject",
        "HOST": "dbprimary",
        # ... plus some other settings
    },
    "replica": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myproject",
        "HOST": "dbreplica",
        "TEST": {
```

(continues on next page)

(continued from previous page)

```
            "MIRROR": "default",
        },
        # ... plus some other settings
    },
}
```

In this setup, we have two database servers: `dbprimary`, described by the database alias `default`, and `dbreplica` described by the alias `replica`. As you might expect, `dbreplica` has been configured by the database administrator as a read replica of `dbprimary`, so in normal activity, any write to `default` will appear on `replica`.

If Django created two independent test databases, this would break any tests that expected replication to occur. However, the `replica` database has been configured as a test mirror (using the *MIRROR* test setting), indicating that under testing, `replica` should be treated as a mirror of `default`.

When the test environment is configured, a test version of `replica` will not be created. Instead the connection to `replica` will be redirected to point at `default`. As a result, writes to `default` will appear on `replica` – but because they are actually the same database, not because there is data replication between the two databases. As this depends on transactions, the tests must use *TransactionTestCase* instead of *TestCase*.

Controlling creation order for test databases

By default, Django will assume all databases depend on the `default` database and therefore always create the `default` database first. However, no guarantees are made on the creation order of any other databases in your test setup.

If your database configuration requires a specific creation order, you can specify the dependencies that exist using the *DEPENDENCIES* test setting. Consider the following (simplified) example database configuration:

```
DATABASES = {
    "default": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds"],
        },
    },
    "diamonds": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": [],
        },
    },
    "clubs": {
```

(continues on next page)

(continued from previous page)

```
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds"],
        },
    },
    "spades": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds", "hearts"],
        },
    },
    "hearts": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds", "clubs"],
        },
    },
}
```

Under this configuration, the `diamonds` database will be created first, as it is the only database alias without dependencies. The `default` and `clubs` alias will be created next (although the order of creation of this pair is not guaranteed), then `hearts`, and finally `spades`.

If there are any circular dependencies in the *DEPENDENCIES* definition, an *ImproperlyConfigured* exception will be raised.

Advanced features of `TransactionTestCase`

`TransactionTestCase.available_apps`

> **Warning**
>
> This attribute is a private API. It may be changed or removed without a deprecation period in the future, for instance to accommodate changes in application loading.
>
> It's used to optimize Django's own test suite, which contains hundreds of models but no relations between models in different applications.

By default, `available_apps` is set to `None`. After each test, Django calls *flush* to reset the database state. This empties all tables and emits the *post_migrate* signal, which recreates one content type and four permissions for each model. This operation gets expensive proportionally to the number of models.

Setting `available_apps` to a list of applications instructs Django to behave as if only the models from these applications were available. The behavior of `TransactionTestCase` changes as follows:

- *post_migrate* is fired before each test to create the content types and permissions for each model in available apps, in case they're missing.
- After each test, Django empties only tables corresponding to models in available apps. However, at the database level, truncation may cascade to related models in unavailable apps. Furthermore *post_migrate* isn't fired; it will be fired by the next `TransactionTestCase`, after the correct set of applications is selected.

Since the database isn't fully flushed, if a test creates instances of models not included in `available_apps`, they will leak and they may cause unrelated tests to fail. Be careful with tests that use sessions; the default session engine stores them in the database.

Since *post_migrate* isn't emitted after flushing the database, its state after a `TransactionTestCase` isn't the same as after a `TestCase`: it's missing the rows created by listeners to *post_migrate*. Considering the order in which tests are executed, this isn't an issue, provided either all `TransactionTestCase` in a given test suite declare `available_apps`, or none of them.

`available_apps` is mandatory in Django's own test suite.

`TransactionTestCase.reset_sequences`

Setting `reset_sequences = True` on a `TransactionTestCase` will make sure sequences are always reset before the test run:

```
class TestsThatDependsOnPrimaryKeySequences(TransactionTestCase):
    reset_sequences = True

    def test_animal_pk(self):
        lion = Animal.objects.create(name="lion", sound="roar")
        # lion.pk is guaranteed to always be 1
        self.assertEqual(lion.pk, 1)
```

Unless you are explicitly testing primary keys sequence numbers, it is recommended that you do not hard code primary key values in tests.

Using `reset_sequences = True` will slow down the test, since the primary key reset is a relatively expensive database operation.

Enforce running test classes sequentially

If you have test classes that cannot be run in parallel (e.g. because they share a common resource), you can use `django.test.testcases.SerializeMixin` to run them sequentially. This mixin uses a filesystem `lockfile`.

For example, you can use `__file__` to determine that all test classes in the same file that inherit from `SerializeMixin` will run sequentially:

```
import os

from django.test import TestCase
from django.test.testcases import SerializeMixin


class ImageTestCaseMixin(SerializeMixin):
    lockfile = __file__

    def setUp(self):
        self.filename = os.path.join(temp_storage_dir, "my_file.png")
        self.file = create_file(self.filename)


class RemoveImageTests(ImageTestCaseMixin, TestCase):
    def test_remove_image(self):
        os.remove(self.filename)
        self.assertFalse(os.path.exists(self.filename))


class ResizeImageTests(ImageTestCaseMixin, TestCase):
    def test_resize_image(self):
        resize_image(self.file, (48, 48))
        self.assertEqual(get_image_size(self.file), (48, 48))
```

Using the Django test runner to test reusable applications

If you are writing a reusable application you may want to use the Django test runner to run your own test suite and thus benefit from the Django testing infrastructure.

A common practice is a tests directory next to the application code, with the following structure:

```
runtests.py
polls/
    __init__.py
    models.py
    ...
tests/
    __init__.py
```

(continues on next page)

(continued from previous page)

```
    models.py
    test_settings.py
    tests.py
```

Let's take a look inside a couple of those files:

Listing 26: `runtests.py`

```
#!/usr/bin/env python
import os
import sys

import django
from django.conf import settings
from django.test.utils import get_runner

if __name__ == "__main__":
    os.environ["DJANGO_SETTINGS_MODULE"] = "tests.test_settings"
    django.setup()
    TestRunner = get_runner(settings)
    test_runner = TestRunner()
    failures = test_runner.run_tests(["tests"])
    sys.exit(bool(failures))
```

This is the script that you invoke to run the test suite. It sets up the Django environment, creates the test database and runs the tests.

For the sake of clarity, this example contains only the bare minimum necessary to use the Django test runner. You may want to add command-line options for controlling verbosity, passing in specific test labels to run, etc.

Listing 27: `tests/test_settings.py`

```
SECRET_KEY = "fake-key"
INSTALLED_APPS = [
    "tests",
]
```

This file contains the Django settings required to run your app's tests.

Again, this is a minimal example; your tests may require additional settings to run.

Since the tests package is included in *INSTALLED_APPS* when running your tests, you can define test-only models in its `models.py` file.

Using different testing frameworks

Clearly, `unittest` is not the only Python testing framework. While Django doesn't provide explicit support for alternative frameworks, it does provide a way to invoke tests constructed for an alternative framework as if they were normal Django tests.

When you run `./manage.py test`, Django looks at the *TEST_RUNNER* setting to determine what to do. By default, *TEST_RUNNER* points to `'django.test.runner.DiscoverRunner'`. This class defines the default Django testing behavior. This behavior involves:

1. Performing global pre-test setup.
2. Looking for tests in any file below the current directory whose name matches the pattern `test*.py`.
3. Creating the test databases.
4. Running `migrate` to install models and initial data into the test databases.
5. Running the system checks.
6. Running the tests that were found.
7. Destroying the test databases.
8. Performing global post-test teardown.

If you define your own test runner class and point *TEST_RUNNER* at that class, Django will execute your test runner whenever you run `./manage.py test`. In this way, it is possible to use any test framework that can be executed from Python code, or to modify the Django test execution process to satisfy whatever testing requirements you may have.

Defining a test runner

A test runner is a class defining a `run_tests()` method. Django ships with a `DiscoverRunner` class that defines the default Django testing behavior. This class defines the `run_tests()` entry point, plus a selection of other methods that are used by `run_tests()` to set up, execute and tear down the test suite.

`class DiscoverRunner`(pattern='test*.py', top_level=None, verbosity=1, interactive=True, failfast=False, keepdb=False, reverse=False, debug_mode=False, debug_sql=False, parallel=0, tags=None, exclude_tags=None, test_name_patterns=None, pdb=False, buffer=False, enable_faulthandler=True, timing=True, shuffle=False, logger=None, durations=None, **kwargs)

`DiscoverRunner` will search for tests in any file matching `pattern`.

`top_level` can be used to specify the directory containing your top-level Python modules. Usually Django can figure this out automatically, so it's not necessary to specify this option. If specified, it should generally be the directory containing your `manage.py` file.

`verbosity` determines the amount of notification and debug information that will be printed to the console; `0` is no output, `1` is normal output, and `2` is verbose output.

If `interactive` is `True`, the test suite has permission to ask the user for instructions when the test suite is executed. An example of this behavior would be asking for permission to delete an existing test database. If `interactive` is `False`, the test suite must be able to run without any manual intervention.

If `failfast` is `True`, the test suite will stop running after the first test failure is detected.

If `keepdb` is `True`, the test suite will use the existing database, or create one if necessary. If `False`, a new database will be created, prompting the user to remove the existing one, if present.

If `reverse` is `True`, test cases will be executed in the opposite order. This could be useful to debug tests that aren't properly isolated and have side effects. Grouping by test class is preserved when using this option. This option can be used in conjunction with `--shuffle` to reverse the order for a particular random seed.

`debug_mode` specifies what the *DEBUG* setting should be set to prior to running tests.

`parallel` specifies the number of processes. If `parallel` is greater than `1`, the test suite will run in `parallel` processes. If there are fewer test case classes than configured processes, Django will reduce the number of processes accordingly. Each process gets its own database. This option requires the third-party `tblib` package to display tracebacks correctly.

`tags` can be used to specify a set of tags for filtering tests. May be combined with `exclude_tags`.

`exclude_tags` can be used to specify a set of tags for excluding tests. May be combined with `tags`.

If `debug_sql` is `True`, failing test cases will output SQL queries logged to the django.db.backends logger as well as the traceback. If `verbosity` is `2`, then queries in all tests are output.

`test_name_patterns` can be used to specify a set of patterns for filtering test methods and classes by their names.

If `pdb` is `True`, a debugger (`pdb` or `ipdb`) will be spawned at each test error or failure.

If `buffer` is `True`, outputs from passing tests will be discarded.

If `enable_faulthandler` is `True`, `faulthandler` will be enabled.

If `timing` is `True`, test timings, including database setup and total run time, will be shown.

If `shuffle` is an integer, test cases will be shuffled in a random order prior to execution, using the integer as a random seed. If `shuffle` is `None`, the seed will be generated randomly. In both cases, the seed will be logged and set to `self.shuffle_seed` prior to running tests. This option can be used to help detect tests that aren't properly isolated. Grouping by test class is preserved when using this option.

`logger` can be used to pass a Python Logger object. If provided, the logger will be used to log messages instead of printing to the console. The logger object will respect its logging level rather than the `verbosity`.

`durations` will show a list of the N slowest test cases. Setting this option to `0` will result in the duration for all tests being shown. Requires Python 3.12+.

Django may, from time to time, extend the capabilities of the test runner by adding new arguments. The `**kwargs` declaration allows for this expansion. If you subclass `DiscoverRunner` or write your own test runner, ensure it accepts `**kwargs`.

Your test runner may also define additional command-line options. Create or override an `add_arguments(cls, parser)` class method and add custom arguments by calling `parser.add_argument()` inside the method, so that the *`test`* command will be able to use those arguments.

Attributes

`DiscoverRunner.test_suite`

The class used to build the test suite. By default it is set to `unittest.TestSuite`. This can be overridden if you wish to implement different logic for collecting tests.

`DiscoverRunner.test_runner`

This is the class of the low-level test runner which is used to execute the individual tests and format the results. By default it is set to `unittest.TextTestRunner`. Despite the unfortunate similarity in naming conventions, this is not the same type of class as `DiscoverRunner`, which covers a broader set of responsibilities. You can override this attribute to modify the way tests are run and reported.

`DiscoverRunner.test_loader`

This is the class that loads tests, whether from TestCases or modules or otherwise and bundles them into test suites for the runner to execute. By default it is set to `unittest.defaultTestLoader`. You can override this attribute if your tests are going to be loaded in unusual ways.

Methods

`DiscoverRunner.run_tests`(test_labels, **kwargs)

Run the test suite.

`test_labels` allows you to specify which tests to run and supports several formats (see *`DiscoverRunner.build_suite()`* for a list of supported formats).

This method should return the number of tests that failed.

`classmethod DiscoverRunner.add_arguments`(parser)

Override this class method to add custom arguments accepted by the *`test`* management command. See `argparse.ArgumentParser.add_argument()` for details about adding arguments to a parser.

`DiscoverRunner.setup_test_environment`(**kwargs)

Sets up the test environment by calling *`setup_test_environment()`* and setting *`DEBUG`* to `self.debug_mode` (defaults to `False`).

`DiscoverRunner.build_suite`(test_labels=None, **kwargs)

Constructs a test suite that matches the test labels provided.

`test_labels` is a list of strings describing the tests to be run. A test label can take one of four forms:

- `path.to.test_module.TestCase.test_method` – Run a single test method in a test case class.
- `path.to.test_module.TestCase` – Run all the test methods in a test case.
- `path.to.module` – Search for and run all tests in the named Python package or module.
- `path/to/directory` – Search for and run all tests below the named directory.

If `test_labels` has a value of `None`, the test runner will search for tests in all files below the current directory whose names match its `pattern` (see above).

Returns a `TestSuite` instance ready to be run.

`DiscoverRunner.setup_databases`(**kwargs)

Creates the test databases by calling *`setup_databases()`*.

`DiscoverRunner.run_checks`(databases)

Runs the system checks on the test `databases`.

`DiscoverRunner.run_suite`(suite, **kwargs)

Runs the test suite.

Returns the result produced by the running the test suite.

`DiscoverRunner.get_test_runner_kwargs`()

Returns the keyword arguments to instantiate the `DiscoverRunner.test_runner` with.

`DiscoverRunner.teardown_databases`(old_config, **kwargs)

Destroys the test databases, restoring pre-test conditions by calling *`teardown_databases()`*.

`DiscoverRunner.teardown_test_environment`(**kwargs)

Restores the pre-test environment.

`DiscoverRunner.suite_result`(suite, result, **kwargs)

Computes and returns a return code based on a test suite, and the result from that test suite.

`DiscoverRunner.log`(msg, level=None)

If a `logger` is set, logs the message at the given integer logging level (e.g. `logging.DEBUG`, `logging.INFO`, or `logging.WARNING`). Otherwise, the message is printed to the console, respecting the current `verbosity`. For example, no message will be printed if the `verbosity` is 0, `INFO` and above will be printed if the `verbosity` is at least 1, and `DEBUG` will be printed if it is at least 2. The `level` defaults to `logging.INFO`.

Testing utilities

`django.test.utils`

To assist in the creation of your own test runner, Django provides a number of utility methods in the `django.test.utils` module.

setup_test_environment(debug=None)

Performs global pre-test setup, such as installing instrumentation for the template rendering system and setting up the dummy email outbox.

If `debug` isn't `None`, the *DEBUG* setting is updated to its value.

teardown_test_environment()

Performs global post-test teardown, such as removing instrumentation from the template system and restoring normal email services.

setup_databases(verbosity, interactive, *, time_keeper=None, keepdb=False, debug_sql=False, parallel=0, aliases=None, serialized_aliases=None, **kwargs)

Creates the test databases.

Returns a data structure that provides enough detail to undo the changes that have been made. This data will be provided to the *teardown_databases()* function at the conclusion of testing.

The `aliases` argument determines which *DATABASES* aliases test databases should be set up for. If it's not provided, it defaults to all of *DATABASES* aliases.

The `serialized_aliases` argument determines what subset of `aliases` test databases should have their state serialized to allow usage of the serialized_rollback feature. If it's not provided, it defaults to `aliases`.

teardown_databases(old_config, parallel=0, keepdb=False)

Destroys the test databases, restoring pre-test conditions.

`old_config` is a data structure defining the changes in the database configuration that need to be reversed. It's the return value of the *setup_databases()* method.

django.db.connection.creation

The creation module of the database backend also provides some utilities that can be useful during testing.

create_test_db(verbosity=1, autoclobber=False, serialize=True, keepdb=False)

Creates a new test database and runs `migrate` against it.

`verbosity` has the same behavior as in `run_tests()`.

`autoclobber` describes the behavior that will occur if a database with the same name as the test database is discovered:

- If `autoclobber` is `False`, the user will be asked to approve destroying the existing database. `sys.exit` is called if the user does not approve.
- If `autoclobber` is `True`, the database will be destroyed without consulting the user.

`serialize` determines if Django serializes the database into an in-memory JSON string before running tests (used to restore the database state between tests if you don't have transactions). You can set this to `False` to speed up creation time if you don't have any test classes with serialized_rollback=True.

`keepdb` determines if the test run should use an existing database, or create a new one. If `True`, the existing database will be used, or created if not present. If `False`, a new database will be created, prompting the user to remove the existing one, if present.

Returns the name of the test database that it created.

`create_test_db()` has the side effect of modifying the value of *NAME* in *DATABASES* to match the name of the test database.

`destroy_test_db`(old_database_name, verbosity=1, keepdb=False)

Destroys the database whose name is the value of *NAME* in *DATABASES*, and sets *NAME* to the value of `old_database_name`.

The `verbosity` argument has the same behavior as for *DiscoverRunner*.

If the `keepdb` argument is `True`, then the connection to the database will be closed, but the database will not be destroyed.

`serialize_db_to_string()`

Serializes the database into an in-memory JSON string that can be used to restore the database state between tests if the backend doesn't support transactions or if your suite contains test classes with serialized_rollback=True enabled.

This function should only be called once all test databases have been created as the serialization process could result in queries against non-test databases depending on your routing configuration.

Integration with `coverage.py`

Code coverage describes how much source code has been tested. It shows which parts of your code are being exercised by tests and which are not. It's an important part of testing applications, so it's strongly recommended to check the coverage of your tests.

Django can be easily integrated with coverage.py, a tool for measuring code coverage of Python programs. First, install coverage. Next, run the following from your project folder containing `manage.py`:

```
coverage run --source='.' manage.py test myapp
```

This runs your tests and collects coverage data of the executed files in your project. You can see a report of this data by typing following command:

```
coverage report
```

Note that some Django code was executed while running tests, but it is not listed here because of the `source` flag passed to the previous command.

For more options like annotated HTML listings detailing missed lines, see the coverage.py docs.

3.10 User authentication in Django

3.10.1 Using the Django authentication system

This document explains the usage of Django's authentication system in its default configuration. This configuration has evolved to serve the most common project needs, handling a reasonably wide range of tasks, and has a careful implementation of passwords and permissions. For projects where authentication needs differ from the default, Django supports extensive extension and customization of authentication.

Django authentication provides both authentication and authorization together and is generally referred to as the authentication system, as these features are somewhat coupled.

`User` objects

`User` objects are the core of the authentication system. They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc. Only one class of user exists in Django's authentication framework, i.e., *`'superusers'`* or admin *`'staff'`* users are just user objects with special attributes set, not different classes of user objects.

The primary attributes of the default user are:

- *`username`*
- *`password`*
- *`email`*
- *`first_name`*
- *`last_name`*

See the *`full API documentation`* for full reference, the documentation that follows is more task oriented.

Creating users

The most direct way to create users is to use the included *`create_user()`* helper function:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user("john", "lennon@thebeatles.com", "johnpassword")

# At this point, user is a User object that has already been saved
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.last_name = "Lennon"
>>> user.save()
```

If you have the Django admin installed, you can also create users interactively.

Creating superusers

Create superusers using the *createsuperuser* command:

```
$ python manage.py createsuperuser --username=joe --email=joe@example.com
```

You will be prompted for a password. After you enter one, the user will be created immediately. If you leave off the *--username* or *--email* options, it will prompt you for those values.

Changing passwords

Django does not store raw (clear text) passwords on the user model, but only a hash (see documentation of how passwords are managed for full details). Because of this, do not attempt to manipulate the password attribute of the user directly. This is why a helper function is used when creating a user.

To change a user's password, you have several options:

*manage.py changepassword *username** offers a method of changing a user's password from the command line. It prompts you to change the password of a given user which you must enter twice. If they both match, the new password will be changed immediately. If you do not supply a user, the command will attempt to change the password whose username matches the current system user.

You can also change a password programmatically, using *set_password()*:

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username="john")
>>> u.set_password("new password")
>>> u.save()
```

If you have the Django admin installed, you can also change user's passwords on the authentication system's admin pages.

Django also provides views and forms that may be used to allow users to change their own passwords.

Changing a user's password will log out all their sessions. See Session invalidation on password change for details.

Authenticating users

authenticate(request=None, **credentials)

aauthenticate(request=None, **credentials)

Asynchronous version: `aauthenticate()`

Use *authenticate()* to verify a set of credentials. It takes credentials as keyword arguments, `username` and `password` for the default case, checks them against each authentication backend, and returns a *User* object if the credentials are valid for a backend. If the credentials aren't valid for any backend or if a backend raises *PermissionDenied*, it returns `None`. For example:

```
from django.contrib.auth import authenticate

user = authenticate(username="john", password="secret")
if user is not None:
    # A backend authenticated the credentials
    ...
else:
    # No backend authenticated the credentials
    ...
```

`request` is an optional *HttpRequest* which is passed on the `authenticate()` method of the authentication backends.

> **Note**
>
> This is a low level way to authenticate a set of credentials; for example, it's used by the *RemoteUserMiddleware*. Unless you are writing your own authentication system, you probably won't use this. Rather if you're looking for a way to login a user, use the *LoginView*.

Permissions and Authorization

Django comes with a built-in permissions system. It provides a way to assign permissions to specific users and groups of users.

It's used by the Django admin site, but you're welcome to use it in your own code.

The Django admin site uses permissions as follows:

- Access to view objects is limited to users with the "view" or "change" permission for that type of object.
- Access to view the "add" form and add an object is limited to users with the "add" permission for that type of object.
- Access to view the change list, view the "change" form and change an object is limited to users with the "change" permission for that type of object.
- Access to delete an object is limited to users with the "delete" permission for that type of object.

Permissions can be set not only per type of object, but also per specific object instance. By using the *has_view_permission()*, *has_add_permission()*, *has_change_permission()* and *has_delete_permission()* methods provided by the *ModelAdmin* class, it is possible to customize permissions for different object instances of the same type.

User objects have two many-to-many fields: `groups` and `user_permissions`. *User* objects can access their related objects in the same way as any other Django model:

```
myuser.groups.set([group_list])
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions.set([permission_list])
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

Default permissions

When `django.contrib.auth` is listed in your *INSTALLED_APPS* setting, it will ensure that four default permissions – add, change, delete, and view – are created for each Django model defined in one of your installed applications.

These permissions will be created when you run *manage.py migrate*; the first time you run `migrate` after adding `django.contrib.auth` to *INSTALLED_APPS*, the default permissions will be created for all previously-installed models, as well as for any new models being installed at that time. Afterward, it will create default permissions for new models each time you run *manage.py migrate* (the function that creates permissions is connected to the *post_migrate* signal).

Assuming you have an application with an *app_label* `foo` and a model named `Bar`, to test for basic permissions you should use:

- add: `user.has_perm('foo.add_bar')`
- change: `user.has_perm('foo.change_bar')`
- delete: `user.has_perm('foo.delete_bar')`
- view: `user.has_perm('foo.view_bar')`

The *Permission* model is rarely accessed directly.

Groups

django.contrib.auth.models.Group models are a generic way of categorizing users so you can apply permissions, or some other label, to those users. A user can belong to any number of groups.

A user in a group automatically has the permissions granted to that group. For example, if the group `Site editors` has the permission `can_edit_home_page`, any user in that group will have that permission.

Beyond permissions, groups are a convenient way to categorize users to give them some label, or extended functionality. For example, you could create a group `'Special users'`, and you could write code that could, say, give them access to a members-only portion of your site, or send them members-only email messages.

Programmatically creating permissions

While custom permissions can be defined within a model's `Meta` class, you can also create permissions directly. For example, you can create the `can_publish` permission for a `BlogPost` model in `myapp`:

```
from myapp.models import BlogPost
from django.contrib.auth.models import Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(BlogPost)
permission = Permission.objects.create(
    codename="can_publish",
    name="Can Publish Posts",
    content_type=content_type,
)
```

The permission can then be assigned to a *User* via its `user_permissions` attribute or to a *Group* via its `permissions` attribute.

Proxy models need their own content type

If you want to create permissions for a proxy model, pass `for_concrete_model=False` to *ContentTypeManager.get_for_model()* to get the appropriate `ContentType`:

```
content_type = ContentType.objects.get_for_model(
    BlogPostProxy, for_concrete_model=False
)
```

Permission caching

The *ModelBackend* caches permissions on the user object after the first time they need to be fetched for a permissions check. This is typically fine for the request-response cycle since permissions aren't typically checked immediately after they are added (in the admin, for example). If you are adding permissions and checking them immediately afterward, in a test or view for example, the easiest solution is to re-fetch the user from the database. For example:

```
from django.contrib.auth.models import Permission, User
from django.contrib.contenttypes.models import ContentType
from django.shortcuts import get_object_or_404

from myapp.models import BlogPost
```

(continues on next page)

(continued from previous page)

```
def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    # any permission check will cache the current set of permissions
    user.has_perm("myapp.change_blogpost")

    content_type = ContentType.objects.get_for_model(BlogPost)
    permission = Permission.objects.get(
        codename="change_blogpost",
        content_type=content_type,
    )
    user.user_permissions.add(permission)

    # Checking the cached permission set
    user.has_perm("myapp.change_blogpost")  # False

    # Request new instance of User
    # Be aware that user.refresh_from_db() won't clear the cache.
    user = get_object_or_404(User, pk=user_id)

    # Permission cache is repopulated from the database
    user.has_perm("myapp.change_blogpost")  # True

    ...
```

Proxy models

Proxy models work exactly the same way as concrete models. Permissions are created using the own content type of the proxy model. Proxy models don't inherit the permissions of the concrete model they subclass:

```
class Person(models.Model):
    class Meta:
        permissions = [("can_eat_pizzas", "Can eat pizzas")]


class Student(Person):
    class Meta:
        proxy = True
        permissions = [("can_deliver_pizzas", "Can deliver pizzas")]
```

```
>>> # Fetch the content type for the proxy model.
>>> content_type = ContentType.objects.get_for_model(Student, for_concrete_model=False)
>>> student_permissions = Permission.objects.filter(content_type=content_type)
>>> [p.codename for p in student_permissions]
['add_student', 'change_student', 'delete_student', 'view_student',
'can_deliver_pizzas']
>>> for permission in student_permissions:
...     user.user_permissions.add(permission)
...
>>> user.has_perm("app.add_person")
False
>>> user.has_perm("app.can_eat_pizzas")
False
>>> user.has_perms(("app.add_student", "app.can_deliver_pizzas"))
True
```

Authentication in web requests

Django uses sessions and middleware to hook the authentication system into *request objects*.

These provide a *request.user* attribute and a *request.auser* async method on every request which represents the current user. If the current user has not logged in, this attribute will be set to an instance of *AnonymousUser*, otherwise it will be an instance of *User*.

You can tell them apart with *is_authenticated*, like so:

```
if request.user.is_authenticated:
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

Or in an asynchronous view:

```
user = await request.auser()
if user.is_authenticated:
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

How to log a user in

If you have an authenticated user you want to attach to the current session - this is done with a *login()* function.

login(request, user, backend=None)

alogin(request, user, backend=None)

Asynchronous version: `alogin()`

To log a user in, from a view, use *login()*. It takes an *HttpRequest* object and a *User* object. *login()* saves the user's ID in the session, using Django's session framework.

Note that any data set during the anonymous session is retained in the session after a user logs in.

This example shows how you might use both *authenticate()* and *login()*:

```
from django.contrib.auth import authenticate, login


def my_view(request):
    username = request.POST["username"]
    password = request.POST["password"]
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)
        # Redirect to a success page.
        ...
    else:
        # Return an 'invalid login' error message.
        ...
```

Selecting the authentication backend

When a user logs in, the user's ID and the backend that was used for authentication are saved in the user's session. This allows the same authentication backend to fetch the user's details on a future request. The authentication backend to save in the session is selected as follows:

1. Use the value of the optional `backend` argument, if provided.
2. Use the value of the `user.backend` attribute, if present. This allows pairing *authenticate()* and *login()*: *authenticate()* sets the `user.backend` attribute on the user object it returns.
3. Use the `backend` in *AUTHENTICATION_BACKENDS*, if there is only one.
4. Otherwise, raise an exception.

In cases 1 and 2, the value of the `backend` argument or the `user.backend` attribute should be a dotted import path string (like that found in *AUTHENTICATION_BACKENDS*), not the actual backend class.

How to log a user out

`logout`(request)

`alogout`(request)

Asynchronous version: `alogout()`

To log out a user who has been logged in via *django.contrib.auth.login()*, use *django.contrib.auth.logout()* within your view. It takes an *HttpRequest* object and has no return value. Example:

```
from django.contrib.auth import logout


def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

Note that *logout()* doesn't throw any errors if the user wasn't logged in.

When you call *logout()*, the session data for the current request is completely cleaned out. All existing data is removed. This is to prevent another person from using the same web browser to log in and have access to the previous user's session data. If you want to put anything into the session that will be available to the user immediately after logging out, do that after calling *django.contrib.auth.logout()*.

Limiting access to logged-in users

The raw way

The raw way to limit access to pages is to check *request.user.is_authenticated* and either redirect to a login page:

```
from django.conf import settings
from django.shortcuts import redirect


def my_view(request):
    if not request.user.is_authenticated:
        return redirect(f"{settings.LOGIN_URL}?next={request.path}")
    # ...
```

...or display an error message:

```
from django.shortcuts import render


def my_view(request):
    if not request.user.is_authenticated:
        return render(request, "myapp/login_error.html")
    # ...
```

The `login_required` decorator

login_required(redirect_field_name='next', login_url=None)

As a shortcut, you can use the convenient *login_required()* decorator:

```
from django.contrib.auth.decorators import login_required


@login_required
def my_view(request): ...
```

login_required() does the following:

- If the user isn't logged in, redirect to *settings.LOGIN_URL*, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.
- If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

By default, the path that the user should be redirected to upon successful authentication is stored in a query string parameter called `"next"`. If you would prefer to use a different name for this parameter, *login_required()* takes an optional `redirect_field_name` parameter:

```
from django.contrib.auth.decorators import login_required


@login_required(redirect_field_name="my_redirect_field")
def my_view(request): ...
```

Note that if you provide a value to `redirect_field_name`, you will most likely need to customize your login template as well, since the template context variable which stores the redirect path will use the value of `redirect_field_name` as its key rather than `"next"` (the default).

login_required() also takes an optional `login_url` parameter. Example:

```
from django.contrib.auth.decorators import login_required


@login_required(login_url="/accounts/login/")
def my_view(request): ...
```

Note that if you don't specify the `login_url` parameter, you'll need to ensure that the *settings.LOGIN_URL* and your login view are properly associated. For example, using the defaults, add the following lines to your URLconf:

```
from django.contrib.auth import views as auth_views

path("accounts/login/", auth_views.LoginView.as_view()),
```

The *settings.LOGIN_URL* also accepts view function names and named URL patterns. This allows you to freely remap your login view within your URLconf without having to update the setting.

> **Note**
>
> The `login_required` decorator does NOT check the `is_active` flag on a user, but the default *AUTHENTICATION_BACKENDS* reject inactive users.

> **See also**
>
> If you are writing custom views for Django's admin (or need the same authorization check that the built-in views use), you may find the *django.contrib.admin.views.decorators.staff_member_required()* decorator a useful alternative to `login_required()`.

Support for wrapping asynchronous view functions was added.

The `LoginRequiredMixin` mixin

When using class-based views, you can achieve the same behavior as with `login_required` by using the `LoginRequiredMixin`. This mixin should be at the leftmost position in the inheritance list.

class LoginRequiredMixin

If a view is using this mixin, all requests by non-authenticated users will be redirected to the login page or shown an HTTP 403 Forbidden error, depending on the *raise_exception* parameter.

You can set any of the parameters of *AccessMixin* to customize the handling of unauthorized users:

```
from django.contrib.auth.mixins import LoginRequiredMixin


class MyView(LoginRequiredMixin, View):
    login_url = "/login/"
    redirect_field_name = "redirect_to"
```

> **Note**
>
> Just as the `login_required` decorator, this mixin does NOT check the `is_active` flag on a user, but the default *`AUTHENTICATION_BACKENDS`* reject inactive users.

The `login_not_required` decorator

When *`LoginRequiredMiddleware`* is installed, all views require authentication by default. Some views, such as the login view, may need to disable this behavior.

`login_not_required`()

Allows unauthenticated requests to this view when *`LoginRequiredMiddleware`* is installed.

Limiting access to logged-in users that pass a test

To limit access based on certain permissions or some other test, you'd do essentially the same thing as described in the previous section.

You can run your test on *`request.user`* in the view directly. For example, this view checks to make sure the user has an email in the desired domain and if not, redirects to the login page:

```
from django.shortcuts import redirect


def my_view(request):
    if not request.user.email.endswith("@example.com"):
        return redirect("/login/?next=%s" % request.path)
    # ...
```

`user_passes_test`(test_func, login_url=None, redirect_field_name='next')

As a shortcut, you can use the convenient `user_passes_test` decorator which performs a redirect when the callable returns `False`:

```
from django.contrib.auth.decorators import user_passes_test
```

(continues on next page)

(continued from previous page)

```
def email_check(user):
    return user.email.endswith("@example.com")


@user_passes_test(email_check)
def my_view(request): ...
```

user_passes_test() takes a required argument: a callable that takes a *User* object and returns `True` if the user is allowed to view the page. Note that *user_passes_test()* does not automatically check that the *User* is not anonymous.

user_passes_test() takes two optional arguments:

login_url

Lets you specify the URL that users who don't pass the test will be redirected to. It may be a login page and defaults to *settings.LOGIN_URL* if you don't specify one.

redirect_field_name

Same as for *login_required()*. Setting it to `None` removes it from the URL, which you may want to do if you are redirecting users that don't pass the test to a non-login page where there's no "next page".

For example:

```
@user_passes_test(email_check, login_url="/login/")
def my_view(request): ...
```

Support for wrapping asynchronous view functions and using asynchronous test callables was added.

class UserPassesTestMixin

When using class-based views, you can use the `UserPassesTestMixin` to do this.

test_func()

You have to override the `test_func()` method of the class to provide the test that is performed. Furthermore, you can set any of the parameters of *AccessMixin* to customize the handling of unauthorized users:

```
from django.contrib.auth.mixins import UserPassesTestMixin


class MyView(UserPassesTestMixin, View):
    def test_func(self):
        return self.request.user.email.endswith("@example.com")
```

`get_test_func()`

You can also override the `get_test_func()` method to have the mixin use a differently named function for its checks (instead of *`test_func()`*).

Stacking `UserPassesTestMixin`

Due to the way `UserPassesTestMixin` is implemented, you cannot stack them in your inheritance list. The following does NOT work:

```
class TestMixin1(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.email.endswith("@example.com")


class TestMixin2(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.username.startswith("django")


class MyView(TestMixin1, TestMixin2, View): ...
```

If `TestMixin1` would call `super()` and take that result into account, `TestMixin1` wouldn't work standalone anymore.

The `permission_required` decorator

`permission_required`(perm, login_url=None, raise_exception=False)

It's a relatively common task to check whether a user has a particular permission. For that reason, Django provides a shortcut for that case: the *`permission_required()`* decorator:

```
from django.contrib.auth.decorators import permission_required


@permission_required("polls.add_choice")
def my_view(request): ...
```

Just like the *`has_perm()`* method, permission names take the form `"<app label>.<permission codename>"` (i.e. `polls.add_choice` for a permission on a model in the `polls` application).

The decorator may also take an iterable of permissions, in which case the user must have all of the permissions in order to access the view.

Note that *`permission_required()`* also takes an optional `login_url` parameter:

```
from django.contrib.auth.decorators import permission_required


@permission_required("polls.add_choice", login_url="/loginpage/")
def my_view(request): ...
```

As in the *login_required()* decorator, `login_url` defaults to *settings.LOGIN_URL*.

If the `raise_exception` parameter is given, the decorator will raise *PermissionDenied*, prompting the 403 (HTTP Forbidden) view instead of redirecting to the login page.

If you want to use `raise_exception` but also give your users a chance to login first, you can add the *login_required()* decorator:

```
from django.contrib.auth.decorators import login_required, permission_required


@login_required
@permission_required("polls.add_choice", raise_exception=True)
def my_view(request): ...
```

This also avoids a redirect loop when *LoginView*'s `redirect_authenticated_user=True` and the logged-in user doesn't have all of the required permissions.

Support for wrapping asynchronous view functions was added.

The `PermissionRequiredMixin` mixin

To apply permission checks to class-based views, you can use the `PermissionRequiredMixin`:

class PermissionRequiredMixin

This mixin, just like the `permission_required` decorator, checks whether the user accessing a view has all given permissions. You should specify the permission (or an iterable of permissions) using the `permission_required` parameter:

```
from django.contrib.auth.mixins import PermissionRequiredMixin


class MyView(PermissionRequiredMixin, View):
    permission_required = "polls.add_choice"
    # Or multiple of permissions:
    permission_required = ["polls.view_choice", "polls.change_choice"]
```

You can set any of the parameters of *AccessMixin* to customize the handling of unauthorized users.

You may also override these methods:

get_permission_required()

Returns an iterable of permission names used by the mixin. Defaults to the `permission_required` attribute, converted to a tuple if necessary.

has_permission()

Returns a boolean denoting whether the current user has permission to execute the decorated view. By default, this returns the result of calling *has_perms()* with the list of permissions returned by *get_permission_required()*.

Redirecting unauthorized requests in class-based views

To ease the handling of access restrictions in class-based views, the `AccessMixin` can be used to configure the behavior of a view when access is denied. Authenticated users are denied access with an HTTP 403 Forbidden response. Anonymous users are redirected to the login page or shown an HTTP 403 Forbidden response, depending on the *raise_exception* attribute.

class AccessMixin

login_url

Default return value for *get_login_url()*. Defaults to `None` in which case *get_login_url()* falls back to *settings.LOGIN_URL*.

permission_denied_message

Default return value for *get_permission_denied_message()*. Defaults to an empty string.

redirect_field_name

Default return value for *get_redirect_field_name()*. Defaults to `"next"`.

raise_exception

If this attribute is set to `True`, a *PermissionDenied* exception is raised when the conditions are not met. When `False` (the default), anonymous users are redirected to the login page.

get_login_url()

Returns the URL that users who don't pass the test will be redirected to. Returns *login_url* if set, or *settings.LOGIN_URL* otherwise.

get_permission_denied_message()

When *raise_exception* is `True`, this method can be used to control the error message passed to the error handler for display to the user. Returns the *permission_denied_message* attribute by default.

get_redirect_field_name()

Returns the name of the query parameter that will contain the URL the user should be redirected to after a successful login. If you set this to `None`, a query parameter won't be added. Returns the *redirect_field_name* attribute by default.

handle_no_permission()

Depending on the value of `raise_exception`, the method either raises a *PermissionDenied* exception or redirects the user to the `login_url`, optionally including the `redirect_field_name` if it is set.

Session invalidation on password change

If your *AUTH_USER_MODEL* inherits from *AbstractBaseUser* or implements its own *get_session_auth_hash()* method, authenticated sessions will include the hash returned by this function. In the *AbstractBaseUser* case, this is an HMAC of the password field. Django verifies that the hash in the session for each request matches the one that's computed during the request. This allows a user to log out all of their sessions by changing their password.

The default password change views included with Django, *PasswordChangeView* and the `user_change_password` view in the *django.contrib.auth* admin, update the session with the new password hash so that a user changing their own password won't log themselves out. If you have a custom password change view and wish to have similar behavior, use the *update_session_auth_hash()* function.

update_session_auth_hash(request, user)

aupdate_session_auth_hash(request, user)

Asynchronous version: `aupdate_session_auth_hash()`

This function takes the current request and the updated user object from which the new session hash will be derived and updates the session hash appropriately. It also rotates the session key so that a stolen session cookie will be invalidated.

Example usage:

```
from django.contrib.auth import update_session_auth_hash


def password_change(request):
    if request.method == "POST":
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            form.save()
            update_session_auth_hash(request, form.user)
    else:
        ...
```

Note

Since *get_session_auth_hash()* is based on *SECRET_KEY*, secret key values must be rotated to avoid

invalidating existing sessions when updating your site to use a new secret. See *SECRET_KEY_FALLBACKS* for details.

Authentication Views

Django provides several views that you can use for handling login, logout, and password management. These make use of the stock auth forms but you can pass in your own forms as well.

Django provides no default template for the authentication views. You should create your own templates for the views you want to use. The template context is documented in each view, see All authentication views.

Using the views

There are different methods to implement these views in your project. The easiest way is to include the provided URLconf in `django.contrib.auth.urls` in your own URLconf, for example:

```
urlpatterns = [
    path("accounts/", include("django.contrib.auth.urls")),
]
```

This will include the following URL patterns:

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

The views provide a URL name for easier reference. See the URL documentation for details on using named URL patterns.

If you want more control over your URLs, you can reference a specific view in your URLconf:

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    path("change-password/", auth_views.PasswordChangeView.as_view()),
]
```

The views have optional arguments you can use to alter the behavior of the view. For example, if you want to change the template name a view uses, you can provide the `template_name` argument. A way to do this

is to provide keyword arguments in the URLconf, these will be passed on to the view. For example:

```
urlpatterns = [
    path(
        "change-password/",
        auth_views.PasswordChangeView.as_view(template_name="change-password.html"),
    ),
]
```

All views are class-based, which allows you to easily customize them by subclassing.

All authentication views

This is a list with all the views `django.contrib.auth` provides. For implementation details see Using the views.

class LoginView

URL name: `login`

See the URL documentation for details on using named URL patterns.

Methods and Attributes

template_name

The name of a template to display for the view used to log the user in. Defaults to `registration/login.html`.

next_page

The URL to redirect to after login. Defaults to *LOGIN_REDIRECT_URL*.

redirect_field_name

The name of a `GET` field containing the URL to redirect to after login. Defaults to `next`. Overrides the *get_default_redirect_url()* URL if the given `GET` parameter is passed.

authentication_form

A callable (typically a form class) to use for authentication. Defaults to *AuthenticationForm*.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

redirect_authenticated_user

A boolean that controls whether or not authenticated users accessing the login page will be redirected as if they had just successfully logged in. Defaults to `False`.

Warning

If you enable `redirect_authenticated_user`, other websites will be able to determine if their visitors are authenticated on your site by requesting redirect URLs to image files on your website. To avoid this "social media fingerprinting" information leakage, host all images and your favicon on a separate domain.

Enabling `redirect_authenticated_user` can also result in a redirect loop when using the *permission_required()* decorator unless the `raise_exception` parameter is used.

success_url_allowed_hosts

A `set` of hosts, in addition to *request.get_host()*, that are safe for redirecting after login. Defaults to an empty `set`.

get_default_redirect_url()

Returns the URL to redirect to after login. The default implementation resolves and returns *next_page* if set, or *LOGIN_REDIRECT_URL* otherwise.

Here's what `LoginView` does:

- If called via `GET`, it displays a login form that POSTs to the same URL. More on this in a bit.
- If called via `POST` with user submitted credentials, it tries to log the user in. If login is successful, the view redirects to the URL specified in `next`. If `next` isn't provided, it redirects to *settings.LOGIN_REDIRECT_URL* (which defaults to `/accounts/profile/`). If login isn't successful, it redisplays the login form.

It's your responsibility to provide the html for the login template , called `registration/login.html` by default. This template gets passed four template context variables:

- `form`: A *Form* object representing the *AuthenticationForm*.
- `next`: The URL to redirect to after successful login. This may contain a query string, too.
- `site`: The current *Site*, according to the *SITE_ID* setting. If you don't have the site framework installed, this will be set to an instance of *RequestSite*, which derives the site name and domain from the current *HttpRequest*.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of *request.META['SERVER_NAME']*. For more on sites, see The "sites" framework.

If you'd prefer not to call the template `registration/login.html`, you can pass the `template_name` parameter via the extra arguments to the `as_view` method in your URLconf. For example, this URLconf line would use `myapp/login.html` instead:

```
path("accounts/login/", auth_views.LoginView.as_view(template_name="myapp/login.html
↪")),
```

You can also specify the name of the `GET` field which contains the URL to redirect to after login using `redirect_field_name`. By default, the field is called `next`.

Here's a sample `registration/login.html` template you can use as a starting point. It assumes you have a `base.html` template that defines a `content` block:

```
{% extends "base.html" %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

{% if next %}
    {% if user.is_authenticated %}
    <p>Your account doesn't have access to this page. To proceed,
    please login with an account that has access.</p>
    {% else %}
    <p>Please login to see this page.</p>
    {% endif %}
{% endif %}

<form method="post" action="{% url 'login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login">
<input type="hidden" name="next" value="{{ next }}">
</form>

{# Assumes you set up the password_reset view in your URLconf #}
<p><a href="{% url 'password_reset' %}">Lost password?</a></p>

{% endblock %}
```

If you have customized authentication (see Customizing Authentication) you can use a custom authentication form by setting the `authentication_form` attribute. This form must accept a `request` keyword argument in its `__init__()` method and provide a `get_user()` method which returns the authenticated user object (this method is only ever called after successful form validation).

`class LogoutView`

Logs a user out on `POST` requests.

URL name: `logout`

Attributes:

`next_page`

The URL to redirect to after logout. Defaults to *`LOGOUT_REDIRECT_URL`*.

`template_name`

The full name of a template to display after logging the user out. Defaults to `registration/logged_out.html`.

`redirect_field_name`

The name of a `GET` field containing the URL to redirect to after log out. Defaults to `'next'`. Overrides the *`next_page`* URL if the given `GET` parameter is passed.

`extra_context`

A dictionary of context data that will be added to the default context data passed to the template.

`success_url_allowed_hosts`

A `set` of hosts, in addition to *`request.get_host()`*, that are safe for redirecting after logout. Defaults to an empty `set`.

Template context:

- `title`: The string "Logged out", localized.
- `site`: The current *`Site`*, according to the *`SITE_ID`* setting. If you don't have the site framework installed, this will be set to an instance of *`RequestSite`*, which derives the site name and domain from the current *`HttpRequest`*.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of *`request.META['SERVER_NAME']`*. For more on sites, see The "sites" framework.

`logout_then_login`(request, login_url=None)

Logs a user out on `POST` requests, then redirects to the login page.

URL name: No default URL provided

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to *`settings.LOGIN_URL`* if not supplied.

class PasswordChangeView

URL name: `password_change`

Allows a user to change their password.

Attributes:

template_name

The full name of a template to use for displaying the password change form. Defaults to `registration/password_change_form.html` if not supplied.

success_url

The URL to redirect to after a successful password change. Defaults to `'password_change_done'`.

form_class

A custom "change password" form which must accept a `user` keyword argument. The form is responsible for actually changing the user's password. Defaults to *PasswordChangeForm*.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

Template context:

- `form`: The password change form (see `form_class` above).

class PasswordChangeDoneView

URL name: `password_change_done`

The page shown after a user has changed their password.

Attributes:

template_name

The full name of a template to use. Defaults to `registration/password_change_done.html` if not supplied.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

class PasswordResetView

URL name: `password_reset`

Allows a user to reset their password by generating a one-time use link that can be used to reset the password, and sending that link to the user's registered email address.

This view will send an email if the following conditions are met:

- The email address provided exists in the system.
- The requested user is active (`User.is_active` is `True`).

- The requested user has a usable password. Users flagged with an unusable password (see *set_unusable_password()*) aren't allowed to request a password reset to prevent misuse when using an external authentication source like LDAP.

If any of these conditions are not met, no email will be sent, but the user won't receive any error message either. This prevents information leaking to potential attackers. If you want to provide an error message in this case, you can subclass *PasswordResetForm* and use the `form_class` attribute.

> **Note**
>
> Be aware that sending an email costs extra time, hence you may be vulnerable to an email address enumeration timing attack due to a difference between the duration of a reset request for an existing email address and the duration of a reset request for a nonexistent email address. To reduce the overhead, you can use a 3rd party package that allows to send emails asynchronously, e.g. django-mailer.

Attributes:

`template_name`

The full name of a template to use for displaying the password reset form. Defaults to `registration/password_reset_form.html` if not supplied.

`form_class`

Form that will be used to get the email of the user to reset the password for. Defaults to *PasswordResetForm*.

`email_template_name`

The full name of a template to use for generating the email with the reset password link. Defaults to `registration/password_reset_email.html` if not supplied.

`subject_template_name`

The full name of a template to use for the subject of the email with the reset password link. Defaults to `registration/password_reset_subject.txt` if not supplied.

`token_generator`

Instance of the class to check the one time link. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.

`success_url`

The URL to redirect to after a successful password reset request. Defaults to `'password_reset_done'`.

`from_email`

A valid email address. By default Django uses the *DEFAULT_FROM_EMAIL*.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

html_email_template_name

The full name of a template to use for generating a *text/html* multipart email with the password reset link. By default, HTML email is not sent.

extra_email_context

A dictionary of context data that will be available in the email template. It can be used to override default template context values listed below e.g. `domain`.

Template context:

- `form`: The form (see `form_class` above) for resetting the user's password.

Email template context:

- `email`: An alias for `user.email`
- `user`: The current *User*, according to the `email` form field. Only active users are able to reset their passwords (`User.is_active is True`).
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of *request.META['SERVER_NAME']*. For more on sites, see The "sites" framework.
- `domain`: An alias for `site.domain`. If you don't have the site framework installed, this will be set to the value of `request.get_host()`.
- `protocol`: http or https
- `uid`: The user's primary key encoded in base 64.
- `token`: Token to check that the reset link is valid.

Sample `registration/password_reset_email.html` (email body template):

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token
↪%}
```

The same template context is used for subject template. Subject must be single line plain text string.

class PasswordResetDoneView

URL name: `password_reset_done`

The page shown after a user has been emailed a link to reset their password. This view is called by default if the *PasswordResetView* doesn't have an explicit `success_url` URL set.

> **Note**
>
> If the email address provided does not exist in the system, the user is inactive, or has an unusable password, the user will still be redirected to this view but no email will be sent.

Attributes:

template_name

The full name of a template to use. Defaults to `registration/password_reset_done.html` if not supplied.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

class PasswordResetConfirmView

URL name: `password_reset_confirm`

Presents a form for entering a new password.

Keyword arguments from the URL:

- `uidb64`: The user's id encoded in base 64.
- `token`: Token to check that the password is valid.

Attributes:

template_name

The full name of a template to display the confirm password view. Default value is `registration/password_reset_confirm.html`.

token_generator

Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.

post_reset_login

A boolean indicating if the user should be automatically authenticated after a successful password reset. Defaults to `False`.

post_reset_login_backend

A dotted path to the authentication backend to use when authenticating a user if `post_reset_login` is `True`. Required only if you have multiple *AUTHENTICATION_BACKENDS* configured. Defaults to `None`.

form_class

Form that will be used to set the password. Defaults to *SetPasswordForm*.

success_url

URL to redirect after the password reset done. Defaults to `'password_reset_complete'`.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

reset_url_token

Token parameter displayed as a component of password reset URLs. Defaults to `'set-password'`.

Template context:

- `form`: The form (see `form_class` above) for setting the new user's password.
- `validlink`: Boolean, True if the link (combination of `uidb64` and `token`) is valid or unused yet.

class PasswordResetCompleteView

URL name: `password_reset_complete`

Presents a view which informs the user that the password has been successfully changed.

Attributes:

template_name

The full name of a template to display the view. Defaults to `registration/password_reset_complete.html`.

extra_context

A dictionary of context data that will be added to the default context data passed to the template.

Helper functions

redirect_to_login(next, login_url=None, redirect_field_name='next')

Redirects to the login page, and then back to another URL after a successful login.

Required arguments:

- `next`: The URL to redirect to after a successful login.

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to *settings.LOGIN_URL* if not supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after login. Overrides `next` if the given `GET` parameter is passed.

Built-in forms

If you don't want to use the built-in views, but want the convenience of not having to write forms for this functionality, the authentication system provides several built-in forms located in *django.contrib.auth.forms*:

> **Note**
>
> The built-in authentication forms make certain assumptions about the user model that they are working with. If you're using a custom user model, it may be necessary to define your own forms for the authentication system. For more information, refer to the documentation about using the built-in authentication forms with custom user models.

class AdminPasswordChangeForm

A form used in the admin interface to change a user's password, including the ability to set an *unusable password*, which blocks the user from logging in with password-based authentication.

Takes the `user` as the first positional argument.

Option to disable (or reenable) password-based authentication was added.

class AdminUserCreationForm

A form used in the admin interface to create a new user. Inherits from *UserCreationForm*.

It includes an additional `usable_password` field, enabled by default. If `usable_password` is enabled, it verifies that `password1` and `password2` are non empty and match, validates the password using *validate_password()*, and sets the user's password using *set_password()*. If `usable_password` is disabled, no password validation is done, and password-based authentication is disabled for the user by calling *set_unusable_password()*.

class AuthenticationForm

A form for logging a user in.

Takes `request` as its first positional argument, which is stored on the form instance for use by subclasses.

confirm_login_allowed(user)

By default, `AuthenticationForm` rejects users whose `is_active` flag is set to `False`. You may override this behavior with a custom policy to determine which users can log in. Do this with a custom form that subclasses `AuthenticationForm` and overrides the `confirm_login_allowed()` method. This method should raise a *ValidationError* if the given user may not log in.

For example, to allow all users to log in regardless of "active" status:

```
from django.contrib.auth.forms import AuthenticationForm
```

(continues on next page)

(continued from previous page)

```
class AuthenticationFormWithInactiveUsersOkay(AuthenticationForm):
    def confirm_login_allowed(self, user):
        pass
```

(In this case, you'll also need to use an authentication backend that allows inactive users, such as *AllowAllUsersModelBackend*.)

Or to allow only some active users to log in:

```
class PickyAuthenticationForm(AuthenticationForm):
    def confirm_login_allowed(self, user):
        if not user.is_active:
            raise ValidationError(
                _("This account is inactive."),
                code="inactive",
            )
        if user.username.startswith("b"):
            raise ValidationError(
                _("Sorry, accounts starting with 'b' aren't welcome here."),
                code="no_b_users",
            )
```

class BaseUserCreationForm

A *ModelForm* for creating a new user. This is the recommended base class if you need to customize the user creation form.

It has three fields: `username` (from the user model), `password1`, and `password2`. It verifies that `password1` and `password2` match, validates the password using *validate_password()*, and sets the user's password using *set_password()*.

class PasswordChangeForm

A form for allowing a user to change their password.

class PasswordResetForm

A form for generating and emailing a one-time use link to reset a user's password.

send_mail(subject_template_name, email_template_name, context, from_email, to_email, html_email_template_name=None)

Uses the arguments to send an `EmailMultiAlternatives`. Can be overridden to customize how the email is sent to the user. If you choose to override this method, be mindful of handling potential exceptions raised due to email sending failures.

Parameters

- **subject_template_name** – the template for the subject.
- **email_template_name** – the template for the email body.
- **context** – context passed to the **subject_template**, **email_template**, and **html_email_template** (if it is not None).
- **from_email** – the sender's email.
- **to_email** – the email of the requester.
- **html_email_template_name** – the template for the HTML body; defaults to None, in which case a plain text email is sent.

By default, save() populates the **context** with the same variables that *PasswordResetView* passes to its email context.

class SetPasswordForm

A form that lets a user change their password without entering the old password.

class UserChangeForm

A form used in the admin interface to change a user's information and permissions.

class UserCreationForm

Inherits from *BaseUserCreationForm*. To help prevent confusion with similar usernames, the form doesn't allow usernames that differ only in case.

Authentication data in templates

The currently logged-in user and their permissions are made available in the template context when you use *RequestContext*.

> **Technicality**
>
> Technically, these variables are only made available in the template context if you use *RequestContext* and the 'django.contrib.auth.context_processors.auth' context processor is enabled. It is in the default generated settings file. For more, see the RequestContext docs.

Users

When rendering a template *RequestContext*, the currently logged-in user, either a *User* instance or an *AnonymousUser* instance, is stored in the template variable {{ user }}:

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
```

(continues on next page)

(continued from previous page)

```
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

This template context variable is not available if a `RequestContext` is not being used.

Permissions

The currently logged-in user's permissions are stored in the template variable `{{ perms }}`. This is an instance of `django.contrib.auth.context_processors.PermWrapper`, which is a template-friendly proxy of permissions.

Evaluating a single-attribute lookup of `{{ perms }}` as a boolean is a proxy to *`User.has_module_perms()`*. For example, to check if the logged-in user has any permissions in the `foo` app:

```
{% if perms.foo %}
```

Evaluating a two-level-attribute lookup as a boolean is a proxy to *`User.has_perm()`*. For example, to check if the logged-in user has the permission `foo.add_vote`:

```
{% if perms.foo.add_vote %}
```

Here's a more complete example of checking permissions in a template:

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.add_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.add_driving %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

It is possible to also look permissions up by `{% if in %}` statements. For example:

```
{% if 'foo' in perms %}
    {% if 'foo.add_vote' in perms %}
        <p>In lookup works, too.</p>
    {% endif %}
{% endif %}
```

Managing users in the admin

When you have both `django.contrib.admin` and `django.contrib.auth` installed, the admin provides a convenient way to view and manage users, groups, and permissions. Users can be created and deleted like any Django model. Groups can be created, and permissions can be assigned to users or groups. A log of user edits to models made within the admin is also stored and displayed.

Creating users

You should see a link to "Users" in the "Auth" section of the main admin index page. The "Add user" admin page is different than standard admin pages in that it requires you to choose a username and password before allowing you to edit the rest of the user's fields. Alternatively, on this page, you can choose a username and disable password-based authentication for the user.

Also note: if you want a user account to be able to create users using the Django admin site, you'll need to give them permission to add users and change users (i.e., the "Add user" and "Change user" permissions). If an account has permission to add users but not to change them, that account won't be able to add users. Why? Because if you have permission to add users, you have the power to create superusers, which can then, in turn, change other users. So Django requires add and change permissions as a slight security measure.

Be thoughtful about how you allow users to manage permissions. If you give a non-superuser the ability to edit users, this is ultimately the same as giving them superuser status because they will be able to elevate permissions of users including themselves!

Changing passwords

User passwords are not displayed in the admin (nor stored in the database), but the password storage details are displayed. Included in the display of this information is a link to a password change form that allows admins to change or unset user passwords.

3.10.2 Password management in Django

Password management is something that should generally not be reinvented unnecessarily, and Django endeavors to provide a secure and flexible set of tools for managing user passwords. This document describes how Django stores passwords, how the storage hashing can be configured, and some utilities to work with hashed passwords.

> See also
>
> Even though users may use strong passwords, attackers might be able to eavesdrop on their connections. Use HTTPS to avoid sending passwords (or any other sensitive data) over plain HTTP connections because they will be vulnerable to password sniffing.

How Django stores passwords

Django provides a flexible password storage system and uses PBKDF2 by default.

The *password* attribute of a *User* object is a string in this format:

```
<algorithm>$<iterations>$<salt>$<hash>
```

Those are the components used for storing a User's password, separated by the dollar-sign character and consist of: the hashing algorithm, the number of algorithm iterations (work factor), the random salt, and the resulting password hash. The algorithm is one of a number of one-way hashing or password storage algorithms Django can use; see below. Iterations describe the number of times the algorithm is run over the hash. Salt is the random seed used and the hash is the result of the one-way function.

By default, Django uses the PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST. This should be sufficient for most users: it's quite secure, requiring massive amounts of computing time to break.

However, depending on your requirements, you may choose a different algorithm, or even use a custom algorithm to match your specific security situation. Again, most users shouldn't need to do this – if you're not sure, you probably don't. If you do, please read on:

Django chooses the algorithm to use by consulting the *PASSWORD_HASHERS* setting. This is a list of hashing algorithm classes that this Django installation supports.

For storing passwords, Django will use the first hasher in *PASSWORD_HASHERS*. To store new passwords with a different algorithm, put your preferred algorithm first in *PASSWORD_HASHERS*.

For verifying passwords, Django will find the hasher in the list that matches the algorithm name in the stored password. If a stored password names an algorithm not found in *PASSWORD_HASHERS*, trying to verify it will raise `ValueError`.

The default for *PASSWORD_HASHERS* is:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
]
```

This means that Django will use PBKDF2 to store all passwords but will support checking passwords stored with PBKDF2SHA1, argon2, and bcrypt.

The next few sections describe a couple of common ways advanced users may want to modify this setting.

Using Argon2 with Django

Argon2 is the winner of the 2015 Password Hashing Competition, a community organized open competition to select a next generation hashing algorithm. It's designed not to be easier to compute on custom hardware than it is to compute on an ordinary CPU. The default variant for the Argon2 password hasher is Argon2id.

Argon2 is not the default for Django because it requires a third-party library. The Password Hashing Competition panel, however, recommends immediate use of Argon2 rather than the other algorithms supported by Django.

To use Argon2id as your default storage algorithm, do the following:

1. Install the argon2-cffi package. This can be done by running `python -m pip install django[argon2]`, which is equivalent to `python -m pip install argon2-cffi` (along with any version requirement from Django's `pyproject.toml`).

2. Modify *`PASSWORD_HASHERS`* to list `Argon2PasswordHasher` first. That is, in your settings file, you'd put:

   ```
   PASSWORD_HASHERS = [
       "django.contrib.auth.hashers.Argon2PasswordHasher",
       "django.contrib.auth.hashers.PBKDF2PasswordHasher",
       "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
       "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
       "django.contrib.auth.hashers.ScryptPasswordHasher",
   ]
   ```

   Keep and/or add any entries in this list if you need Django to upgrade passwords.

Using `bcrypt` with Django

Bcrypt is a popular password storage algorithm that's specifically designed for long-term password storage. It's not the default used by Django since it requires the use of third-party libraries, but since many people may want to use it Django supports bcrypt with minimal effort.

To use Bcrypt as your default storage algorithm, do the following:

1. Install the bcrypt package. This can be done by running `python -m pip install django[bcrypt]`, which is equivalent to `python -m pip install bcrypt` (along with any version requirement from Django's `pyproject.toml`).

2. Modify *`PASSWORD_HASHERS`* to list `BCryptSHA256PasswordHasher` first. That is, in your settings file, you'd put:

   ```
   PASSWORD_HASHERS = [
       "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
       "django.contrib.auth.hashers.PBKDF2PasswordHasher",
       "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
   ```

(continues on next page)

(continued from previous page)

```
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
]
```

Keep and/or add any entries in this list if you need Django to upgrade passwords.

That's it – now your Django install will use Bcrypt as the default storage algorithm.

Using `scrypt` with Django

scrypt is similar to PBKDF2 and bcrypt in utilizing a set number of iterations to slow down brute-force attacks. However, because PBKDF2 and bcrypt do not require a lot of memory, attackers with sufficient resources can launch large-scale parallel attacks in order to speed up the attacking process. scrypt is specifically designed to use more memory compared to other password-based key derivation functions in order to limit the amount of parallelism an attacker can use, see RFC 7914 for more details.

To use scrypt as your default storage algorithm, do the following:

1. Modify *`PASSWORD_HASHERS`* to list `ScryptPasswordHasher` first. That is, in your settings file:

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.ScryptPasswordHasher",
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
]
```

Keep and/or add any entries in this list if you need Django to upgrade passwords.

> **Note**
>
> `scrypt` requires OpenSSL 1.1+.

Increasing the salt entropy

Most password hashes include a salt along with their password hash in order to protect against rainbow table attacks. The salt itself is a random value which increases the size and thus the cost of the rainbow table and is currently set at 128 bits with the `salt_entropy` value in the `BasePasswordHasher`. As computing and storage costs decrease this value should be raised. When implementing your own password hasher you are free to override this value in order to use a desired entropy level for your password hashes. `salt_entropy` is measured in bits.

> **Implementation detail**
>
> Due to the method in which salt values are stored the `salt_entropy` value is effectively a minimum value. For instance a value of 128 would provide a salt which would actually contain 131 bits of entropy.

Increasing the work factor

PBKDF2 and bcrypt

The PBKDF2 and bcrypt algorithms use a number of iterations or rounds of hashing. This deliberately slows down attackers, making attacks against hashed passwords harder. However, as computing power increases, the number of iterations needs to be increased. We've chosen a reasonable default (and will increase it with each release of Django), but you may wish to tune it up or down, depending on your security needs and available processing power. To do so, you'll subclass the appropriate algorithm and override the `iterations` parameter (use the `rounds` parameter when subclassing a bcrypt hasher). For example, to increase the number of iterations used by the default PBKDF2 algorithm:

1. Create a subclass of `django.contrib.auth.hashers.PBKDF2PasswordHasher`

```
from django.contrib.auth.hashers import PBKDF2PasswordHasher


class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
    """
    A subclass of PBKDF2PasswordHasher that uses 100 times more iterations.
    """

    iterations = PBKDF2PasswordHasher.iterations * 100
```

   Save this somewhere in your project. For example, you might put this in a file like `myproject/hashers.py`.

2. Add your new hasher as the first entry in *PASSWORD_HASHERS*:

```
PASSWORD_HASHERS = [
    "myproject.hashers.MyPBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
]
```

That's it – now your Django install will use more iterations when it stores passwords using PBKDF2.

> **Note**
>
> bcrypt `rounds` is a logarithmic work factor, e.g. 12 rounds means `2 ** 12` iterations.

Argon2

Argon2 has the following attributes that can be customized:

1. `time_cost` controls the number of iterations within the hash.
2. `memory_cost` controls the size of memory that must be used during the computation of the hash.
3. `parallelism` controls how many CPUs the computation of the hash can be parallelized on.

The default values of these attributes are probably fine for you. If you determine that the password hash is too fast or too slow, you can tweak it as follows:

1. Choose `parallelism` to be the number of threads you can spare computing the hash.
2. Choose `memory_cost` to be the KiB of memory you can spare.
3. Adjust `time_cost` and measure the time hashing a password takes. Pick a `time_cost` that takes an acceptable time for you. If `time_cost` set to 1 is unacceptably slow, lower `memory_cost`.

> **`memory_cost` interpretation**
>
> The argon2 command-line utility and some other libraries interpret the `memory_cost` parameter differently from the value that Django uses. The conversion is given by `memory_cost == 2 ** memory_cost_commandline`.

`scrypt`

scrypt has the following attributes that can be customized:

1. `work_factor` controls the number of iterations within the hash.
2. `block_size`
3. `parallelism` controls how many threads will run in parallel.
4. `maxmem` limits the maximum size of memory that can be used during the computation of the hash. Defaults to `0`, which means the default limitation from the OpenSSL library.

We've chosen reasonable defaults, but you may wish to tune it up or down, depending on your security needs and available processing power.

> **Estimating memory usage**
>
> The minimum memory requirement of scrypt is:

```
work_factor * 2 * block_size * 64
```

so you may need to tweak `maxmem` when changing the `work_factor` or `block_size` values.

Password upgrading

When users log in, if their passwords are stored with anything other than the preferred algorithm, Django will automatically upgrade the algorithm to the preferred one. This means that old installs of Django will get automatically more secure as users log in, and it also means that you can switch to new (and better) storage algorithms as they get invented.

However, Django can only upgrade passwords that use algorithms mentioned in *PASSWORD_HASHERS*, so as you upgrade to new systems you should make sure never to remove entries from this list. If you do, users using unmentioned algorithms won't be able to upgrade. Hashed passwords will be updated when increasing (or decreasing) the number of PBKDF2 iterations, bcrypt rounds, or argon2 attributes.

Be aware that if all the passwords in your database aren't encoded in the default hasher's algorithm, you may be vulnerable to a user enumeration timing attack due to a difference between the duration of a login request for a user with a password encoded in a non-default algorithm and the duration of a login request for a nonexistent user (which runs the default hasher). You may be able to mitigate this by upgrading older password hashes.

Password upgrading without requiring a login

If you have an existing database with an older, weak hash such as MD5, you might want to upgrade those hashes yourself instead of waiting for the upgrade to happen when a user logs in (which may never happen if a user doesn't return to your site). In this case, you can use a "wrapped" password hasher.

For this example, we'll migrate a collection of MD5 hashes to use PBKDF2(MD5(password)) and add the corresponding password hasher for checking if a user entered the correct password on login. We assume we're using the built-in `User` model and that our project has an `accounts` app. You can modify the pattern to work with any algorithm or with a custom user model.

First, we'll add the custom hasher:

Listing 28: `accounts/hashers.py`

```
from django.contrib.auth.hashers import (
    PBKDF2PasswordHasher,
    MD5PasswordHasher,
)


class PBKDF2WrappedMD5PasswordHasher(PBKDF2PasswordHasher):
```

(continues on next page)

(continued from previous page)

```
    algorithm = "pbkdf2_wrapped_md5"

    def encode_md5_hash(self, md5_hash, salt, iterations=None):
        return super().encode(md5_hash, salt, iterations)

    def encode(self, password, salt, iterations=None):
        _, _, md5_hash = MD5PasswordHasher().encode(password, salt).split("$", 2)
        return self.encode_md5_hash(md5_hash, salt, iterations)
```

The data migration might look something like:

Listing 29: accounts/migrations/0002_migrate_md5_passwords.py

```
from django.db import migrations

from ..hashers import PBKDF2WrappedMD5PasswordHasher


def forwards_func(apps, schema_editor):
    User = apps.get_model("auth", "User")
    users = User.objects.filter(password__startswith="md5$")
    hasher = PBKDF2WrappedMD5PasswordHasher()
    for user in users:
        algorithm, salt, md5_hash = user.password.split("$", 2)
        user.password = hasher.encode_md5_hash(md5_hash, salt)
        user.save(update_fields=["password"])


class Migration(migrations.Migration):
    dependencies = [
        ("accounts", "0001_initial"),
        # replace this with the latest migration in contrib.auth
        ("auth", "####_migration_name"),
    ]

    operations = [
        migrations.RunPython(forwards_func),
    ]
```

Be aware that this migration will take on the order of several minutes for several thousand users, depending

on the speed of your hardware.

Finally, we'll add a *PASSWORD_HASHERS* setting:

Listing 30: `mysite/settings.py`

```
PASSWORD_HASHERS = [
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "accounts.hashers.PBKDF2WrappedMD5PasswordHasher",
]
```

Include any other hashers that your site uses in this list.

Included hashers

The full list of hashers included in Django is:

```
[
    "django.contrib.auth.hashers.PBKDF2PasswordHasher",
    "django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher",
    "django.contrib.auth.hashers.Argon2PasswordHasher",
    "django.contrib.auth.hashers.BCryptSHA256PasswordHasher",
    "django.contrib.auth.hashers.BCryptPasswordHasher",
    "django.contrib.auth.hashers.ScryptPasswordHasher",
    "django.contrib.auth.hashers.MD5PasswordHasher",
]
```

The corresponding algorithm names are:

- `pbkdf2_sha256`
- `pbkdf2_sha1`
- `argon2`
- `bcrypt_sha256`
- `bcrypt`
- `scrypt`
- `md5`

Writing your own hasher

If you write your own password hasher that contains a work factor such as a number of iterations, you should implement a `harden_runtime(self, password, encoded)` method to bridge the runtime gap between the work factor supplied in the `encoded` password and the default work factor of the hasher. This prevents a

user enumeration timing attack due to difference between a login request for a user with a password encoded in an older number of iterations and a nonexistent user (which runs the default hasher's default number of iterations).

Taking PBKDF2 as example, if `encoded` contains 20,000 iterations and the hasher's default `iterations` is 30,000, the method should run `password` through another 10,000 iterations of PBKDF2.

If your hasher doesn't have a work factor, implement the method as a no-op (`pass`).

Manually managing a user's password

The *`django.contrib.auth.hashers`* module provides a set of functions to create and validate hashed passwords. You can use them independently from the `User` model.

`check_password`(password, encoded, setter=None, preferred='default')

`acheck_password`(password, encoded, asetter=None, preferred='default')

Asynchronous version: `acheck_password()`

If you'd like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the convenience function *`check_password()`*. It takes two mandatory arguments: the plain-text password to check, and the full value of a user's `password` field in the database to check against. It returns `True` if they match, `False` otherwise. Optionally, you can pass a callable `setter` that takes the password and will be called when you need to regenerate it. You can also pass `preferred` to change a hashing algorithm if you don't want to use the default (first entry of `PASSWORD_HASHERS` setting). See Included hashers for the algorithm name of each hasher.

`make_password`(password, salt=None, hasher='default')

Creates a hashed password in the format used by this application. It takes one mandatory argument: the password in plain-text (string or bytes). Optionally, you can provide a salt and a hashing algorithm to use, if you don't want to use the defaults (first entry of `PASSWORD_HASHERS` setting). See Included hashers for the algorithm name of each hasher. If the password argument is `None`, an unusable password is returned (one that will never be accepted by *`check_password()`*).

`is_password_usable`(encoded_password)

Returns `False` if the password is a result of *`User.set_unusable_password()`*.

Password validation

Users often choose poor passwords. To help mitigate this problem, Django offers pluggable password validation. You can configure multiple password validators at the same time. A few validators are included in Django, but you can write your own as well.

Each password validator must provide a help text to explain the requirements to the user, validate a given password and return an error message if it does not meet the requirements, and optionally define a callback to be notified when the password for a user has been changed. Validators can also have optional settings to fine tune their behavior.

Validation is controlled by the *AUTH_PASSWORD_VALIDATORS* setting. The default for the setting is an empty list, which means no validators are applied. In new projects created with the default *startproject* template, a set of validators is enabled by default.

By default, validators are used in the forms to reset or change passwords and in the *createsuperuser* and *changepassword* management commands. Validators aren't applied at the model level, for example in `User.objects.create_user()` and `create_superuser()`, because we assume that developers, not users, interact with Django at that level and also because model validation doesn't automatically run as part of creating models.

> **Note**
>
> Password validation can prevent the use of many types of weak passwords. However, the fact that a password passes all the validators doesn't guarantee that it is a strong password. There are many factors that can weaken a password that are not detectable by even the most advanced password validators.

Enabling password validation

Password validation is configured in the *AUTH_PASSWORD_VALIDATORS* setting:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME": "django.contrib.auth.password_validation.UserAttributeSimilarityValidator
↪",
    },
    {
        "NAME": "django.contrib.auth.password_validation.MinimumLengthValidator",
        "OPTIONS": {
            "min_length": 9,
        },
    },
    {
        "NAME": "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.NumericPasswordValidator",
    },
]
```

This example enables all four included validators:

- `UserAttributeSimilarityValidator`, which checks the similarity between the password and a set of attributes of the user.

- `MinimumLengthValidator`, which checks whether the password meets a minimum length. This validator is configured with a custom option: it now requires the minimum length to be nine characters, instead of the default eight.
- `CommonPasswordValidator`, which checks whether the password occurs in a list of common passwords. By default, it compares to an included list of 20,000 common passwords.
- `NumericPasswordValidator`, which checks whether the password isn't entirely numeric.

For `UserAttributeSimilarityValidator` and `CommonPasswordValidator`, we're using the default settings in this example. `NumericPasswordValidator` has no settings.

The help texts and any errors from password validators are always returned in the order they are listed in *AUTH_PASSWORD_VALIDATORS*.

Included validators

Django includes four validators:

class `MinimumLengthValidator`(min_length=8)

Validates that the password is of a minimum length. The minimum length can be customized with the `min_length` parameter.

get_error_message()

A hook for customizing the `ValidationError` error message. Defaults to `"This password is too short. It must contain at least <min_length> characters."`.

get_help_text()

A hook for customizing the validator's help text. Defaults to `"Your password must contain at least <min_length> characters."`.

class `UserAttributeSimilarityValidator`(user_attributes=DEFAULT_USER_ATTRIBUTES, max_similarity=0.7)

Validates that the password is sufficiently different from certain attributes of the user.

The `user_attributes` parameter should be an iterable of names of user attributes to compare to. If this argument is not provided, the default is used: `'username', 'first_name', 'last_name', 'email'`. Attributes that don't exist are ignored.

The maximum allowed similarity of passwords can be set on a scale of 0.1 to 1.0 with the `max_similarity` parameter. This is compared to the result of `difflib.SequenceMatcher.quick_ratio()`. A value of 0.1 rejects passwords unless they are substantially different from the `user_attributes`, whereas a value of 1.0 rejects only passwords that are identical to an attribute's value.

get_error_message()

A hook for customizing the `ValidationError` error message. Defaults to `"The password is too similar to the <user_attribute>."`.

`get_help_text()`

A hook for customizing the validator's help text. Defaults to `"Your password can't be too similar to your other personal information."`.

`class CommonPasswordValidator`(password_list_path=DEFAULT_PASSWORD_LIST_PATH)

Validates that the password is not a common password. This converts the password to lowercase (to do a case-insensitive comparison) and checks it against a list of 20,000 common password created by Royce Williams.

The `password_list_path` can be set to the path of a custom file of common passwords. This file should contain one lowercase password per line and may be plain text or gzipped.

`get_error_message()`

A hook for customizing the `ValidationError` error message. Defaults to `"This password is too common."`.

`get_help_text()`

A hook for customizing the validator's help text. Defaults to `"Your password can't be a commonly used password."`.

`class NumericPasswordValidator`

Validate that the password is not entirely numeric.

`get_error_message()`

A hook for customizing the `ValidationError` error message. Defaults to `"This password is entirely numeric."`.

`get_help_text()`

A hook for customizing the validator's help text. Defaults to `"Your password can't be entirely numeric."`.

Integrating validation

There are a few functions in `django.contrib.auth.password_validation` that you can call from your own forms or other code to integrate password validation. This can be useful if you use custom forms for password setting, or if you have API calls that allow passwords to be set, for example.

`validate_password`(password, user=None, password_validators=None)

Validates a password. If all validators find the password valid, returns `None`. If one or more validators reject the password, raises a *`ValidationError`* with all the error messages from the validators.

The `user` object is optional: if it's not provided, some validators may not be able to perform any validation and will accept any password.

`password_changed`(password, user=None, password_validators=None)

Informs all validators that the password has been changed. This can be used by validators such as one that prevents password reuse. This should be called once the password has been successfully changed.

For subclasses of *AbstractBaseUser*, the password field will be marked as "dirty" when calling *set_password()* which triggers a call to `password_changed()` after the user is saved.

`password_validators_help_texts`(password_validators=None)

Returns a list of the help texts of all validators. These explain the password requirements to the user.

`password_validators_help_text_html`(password_validators=None)

Returns an HTML string with all help texts in an `<ul>`. This is helpful when adding password validation to forms, as you can pass the output directly to the `help_text` parameter of a form field.

`get_password_validators`(validator_config)

Returns a set of validator objects based on the `validator_config` parameter. By default, all functions use the validators defined in *AUTH_PASSWORD_VALIDATORS*, but by calling this function with an alternate set of validators and then passing the result into the `password_validators` parameter of the other functions, your custom set of validators will be used instead. This is useful when you have a typical set of validators to use for most scenarios, but also have a special situation that requires a custom set. If you always use the same set of validators, there is no need to use this function, as the configuration from *AUTH_PASSWORD_VALIDATORS* is used by default.

The structure of `validator_config` is identical to the structure of *AUTH_PASSWORD_VALIDATORS*. The return value of this function can be passed into the `password_validators` parameter of the functions listed above.

Note that where the password is passed to one of these functions, this should always be the clear text password - not a hashed password.

Writing your own validator

If Django's built-in validators are not sufficient, you can write your own password validators. Validators have a fairly small interface. They must implement two methods:

- `validate(self, password, user=None)`: validate a password. Return `None` if the password is valid, or raise a *ValidationError* with an error message if the password is not valid. You must be able to deal with `user` being `None` - if that means your validator can't run, return `None` for no error.
- `get_help_text()`: provide a help text to explain the requirements to the user.

Any items in the `OPTIONS` in *AUTH_PASSWORD_VALIDATORS* for your validator will be passed to the constructor. All constructor arguments should have a default value.

Here's a basic example of a validator, with one optional setting:

```
from django.core.exceptions import ValidationError
from django.utils.translation import gettext as _



class MinimumLengthValidator:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, min_length=8):
        self.min_length = min_length

    def validate(self, password, user=None):
        if len(password) < self.min_length:
            raise ValidationError(
                _("This password must contain at least %(min_length)d characters."),
                code="password_too_short",
                params={"min_length": self.min_length},
            )

    def get_help_text(self):
        return _(
            "Your password must contain at least %(min_length)d characters."
            % {"min_length": self.min_length}
        )
```

You can also implement `password_changed(password, user=None)`, which will be called after a successful password change. That can be used to prevent password reuse, for example. However, if you decide to store a user's previous passwords, you should never do so in clear text.

3.10.3 Customizing authentication in Django

The authentication that comes with Django is good enough for most common cases, but you may have needs not met by the out-of-the-box defaults. Customizing authentication in your projects requires understanding what points of the provided system are extensible or replaceable. This document provides details about how the auth system can be customized.

Authentication backends provide an extensible system for when a username and password stored with the user model need to be authenticated against a different service than Django's default.

You can give your models custom permissions that can be checked through Django's authorization system.

You can extend the default `User` model, or substitute a completely customized model.

Other authentication sources

There may be times you have the need to hook into another authentication source – that is, another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It'd be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

So, to handle situations like this, the Django authentication system lets you plug in other authentication

sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

See the authentication backend reference for information on the authentication backends included with Django.

Specifying authentication backends

Behind the scenes, Django maintains a list of "authentication backends" that it checks for authentication. When somebody calls *django.contrib.auth.authenticate()* – as described in How to log a user in – Django tries authenticating across all of its authentication backends. If the first authentication method fails, Django tries the second one, and so on, until all backends have been attempted.

The list of authentication backends to use is specified in the *AUTHENTICATION_BACKENDS* setting. This should be a list of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

By default, *AUTHENTICATION_BACKENDS* is set to:

```
["django.contrib.auth.backends.ModelBackend"]
```

That's the basic authentication backend that checks the Django users database and queries the built-in permissions. It does not provide protection against brute force attacks via any rate limiting mechanism. You may either implement your own rate limiting mechanism in a custom auth backend, or use the mechanisms provided by most web servers.

The order of *AUTHENTICATION_BACKENDS* matters, so if the same username and password is valid in multiple backends, Django will stop processing at the first positive match.

If a backend raises a *PermissionDenied* exception, authentication will immediately fail. Django won't check the backends that follow.

> **Note**
>
> Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and reuses the same backend for the duration of that session whenever access to the currently authenticated user is needed. This effectively means that authentication sources are cached on a per-session basis, so if you change *AUTHENTICATION_BACKENDS*, you'll need to clear out session data if you need to force users to re-authenticate using different methods. A simple way to do that is to execute `Session.objects.all().delete()`.

Writing an authentication backend

An authentication backend is a class that implements two required methods: `get_user(user_id)` and `authenticate(request, **credentials)`, as well as a set of optional permission related authorization methods.

The `get_user` method takes a `user_id` – which could be a username, database ID or whatever, but has to be the primary key of your user object – and returns a user object or `None`.

The `authenticate` method takes a `request` argument and credentials as keyword arguments. Most of the time, it'll look like this:

```
from django.contrib.auth.backends import BaseBackend


class MyBackend(BaseBackend):
    def authenticate(self, request, username=None, password=None):
        # Check the username/password and return a user.
        ...
```

But it could also authenticate a token, like so:

```
from django.contrib.auth.backends import BaseBackend


class MyBackend(BaseBackend):
    def authenticate(self, request, token=None):
        # Check the token and return a user.
        ...
```

Either way, `authenticate()` should check the credentials it gets and return a user object that matches those credentials if the credentials are valid. If they're not valid, it should return `None`.

`request` is an *`HttpRequest`* and may be `None` if it wasn't provided to *`authenticate()`* (which passes it on to the backend).

The Django admin is tightly coupled to the Django User object. For example, for a user to access the admin, *`User.is_staff`* and *`User.is_active`* must be `True` (see *`AdminSite.has_permission()`* for details).

The best way to deal with this is to create a Django `User` object for each user that exists for your backend (e.g., in your LDAP directory, your external SQL database, etc.). You can either write a script to do this in advance, or your `authenticate` method can do it the first time a user logs in.

Here's an example backend that authenticates against a username and password variable defined in your `settings.py` file and creates a Django `User` object the first time a user authenticates. In this example, the created Django `User` object is a superuser who will have full access to the admin:

```
from django.conf import settings
from django.contrib.auth.backends import BaseBackend
from django.contrib.auth.hashers import check_password
from django.contrib.auth.models import User


class SettingsBackend(BaseBackend):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'pbkdf2_sha256$30000$Vo0VlMnkR4Bk$qEvtdyZRWTcOsCnI/
↪oQ7fVOu1XAURIZYoOZ3iq8Dr4M='
    """

    def authenticate(self, request, username=None, password=None):
        login_valid = settings.ADMIN_LOGIN == username
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. There's no need to set a password
                # because only the password from settings.py is checked.
                user = User(username=username)  # is_active defaults to True.
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Handling authorization in custom backends

Custom auth backends can provide their own permissions.

The user model and its manager will delegate permission lookup functions (*get_user_permissions()*, *get_group_permissions()*, *get_all_permissions()*, *has_perm()*, *has_module_perms()*, and *with_perm()*) to any authentication backend that implements these functions.

The permissions given to the user will be the superset of all permissions returned by all backends. That is, Django grants a permission to a user that any one backend grants.

If a backend raises a *PermissionDenied* exception in *has_perm()* or *has_module_perms()*, the authorization will immediately fail and Django won't check the backends that follow.

A backend could implement permissions for the magic admin like this:

```
from django.contrib.auth.backends import BaseBackend


class MagicAdminBackend(BaseBackend):
    def has_perm(self, user_obj, perm, obj=None):
        return user_obj.username == settings.ADMIN_LOGIN
```

This gives full permissions to the user granted access in the above example. Notice that in addition to the same arguments given to the associated *django.contrib.auth.models.User* functions, the backend auth functions all take the user object, which may be an anonymous user, as an argument.

A full authorization implementation can be found in the `ModelBackend` class in django/contrib/auth/backends.py, which is the default backend and queries the `auth_permission` table most of the time.

Authorization for anonymous users

An anonymous user is one that is not authenticated i.e. they have provided no valid authentication details. However, that does not necessarily mean they are not authorized to do anything. At the most basic level, most websites authorize anonymous users to browse most of the site, and many allow anonymous posting of comments etc.

Django's permission framework does not have a place to store permissions for anonymous users. However, the user object passed to an authentication backend may be an *django.contrib.auth.models.AnonymousUser* object, allowing the backend to specify custom authorization behavior for anonymous users. This is especially useful for the authors of reusable apps, who can delegate all questions of authorization to the auth backend, rather than needing settings, for example, to control anonymous access.

Authorization for inactive users

An inactive user is one that has its *is_active* field set to `False`. The *ModelBackend* and *RemoteUserBackend* authentication backends prohibits these users from authenticating. If a custom user model doesn't have an *is_active* field, all users will be allowed to authenticate.

You can use *AllowAllUsersModelBackend* or *AllowAllUsersRemoteUserBackend* if you want to allow inactive users to authenticate.

The support for anonymous users in the permission system allows for a scenario where anonymous users have permissions to do something while inactive authenticated users do not.

Do not forget to test for the `is_active` attribute of the user in your own backend permission methods.

Handling object permissions

Django's permission framework has a foundation for object permissions, though there is no implementation for it in the core. That means that checking for object permissions will always return `False` or an empty list (depending on the check performed). An authentication backend will receive the keyword parameters `obj` and `user_obj` for each object related authorization method and can return the object level permission as appropriate.

Custom permissions

To create custom permissions for a given model object, use the `permissions` model Meta attribute.

This example `Task` model creates two custom permissions, i.e., actions users can or cannot do with `Task` instances, specific to your application:

```
class Task(models.Model):
    ...

    class Meta:
        permissions = [
            ("change_task_status", "Can change the status of tasks"),
            ("close_task", "Can remove a task by setting its status as closed"),
        ]
```

The only thing this does is create those extra permissions when you run *manage.py migrate* (the function that creates permissions is connected to the *post_migrate* signal). Your code is in charge of checking the value of these permissions when a user is trying to access the functionality provided by the application (changing the status of tasks or closing tasks.) Continuing the above example, the following checks if a user may close tasks:

```
user.has_perm("app.close_task")
```

Extending the existing `User` model

There are two ways to extend the default *`User`* model without substituting your own model. If the changes you need are purely behavioral, and don't require any change to what is stored in the database, you can create a proxy model based on *`User`*. This allows for any of the features offered by proxy models including default ordering, custom managers, or custom model methods.

If you wish to store information related to `User`, you can use a *`OneToOneField`* to a model containing the fields for additional information. This one-to-one model is often called a profile model, as it might store non-auth related information about a site user. For example you might create an Employee model:

```
from django.contrib.auth.models import User


class Employee(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    department = models.CharField(max_length=100)
```

Assuming an existing Employee Fred Smith who has both a User and Employee model, you can access the related information using Django's standard related model conventions:

```
>>> u = User.objects.get(username="fsmith")
>>> freds_department = u.employee.department
```

To add a profile model's fields to the user page in the admin, define an *`InlineModelAdmin`* (for this example, we'll use a *`StackedInline`*) in your app's `admin.py` and add it to a `UserAdmin` class which is registered with the *`User`* class:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import User

from my_user_profile_app.models import Employee


# Define an inline admin descriptor for Employee model
# which acts a bit like a singleton
class EmployeeInline(admin.StackedInline):
    model = Employee
    can_delete = False
    verbose_name_plural = "employee"
```

(continues on next page)

(continued from previous page)

```
# Define a new User admin
class UserAdmin(BaseUserAdmin):
    inlines = [EmployeeInline]


# Re-register UserAdmin
admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

These profile models are not special in any way - they are just Django models that happen to have a one-to-one link with a user model. As such, they aren't auto created when a user is created, but a *django.db.models.signals.post_save* could be used to create or update related models as appropriate.

Using related models results in additional queries or joins to retrieve the related data. Depending on your needs, a custom user model that includes the related fields may be your better option, however, existing relations to the default user model within your project's apps may justify the extra database load.

Substituting a custom `User` model

Some kinds of projects may have authentication requirements for which Django's built-in *User* model is not always appropriate. For instance, on some sites it makes more sense to use an email address as your identification token instead of a username.

Django allows you to override the default user model by providing a value for the *AUTH_USER_MODEL* setting that references a custom model:

```
AUTH_USER_MODEL = "myapp.MyUser"
```

This dotted pair describes the *label* of the Django app (which must be in your *INSTALLED_APPS*), and the name of the Django model that you wish to use as your user model.

Using a custom user model when starting a project

If you're starting a new project, you can set up a custom user model that behaves identically to the default user model by subclassing *AbstractUser*:

```
from django.contrib.auth.models import AbstractUser


class User(AbstractUser):
    pass
```

Don't forget to point *AUTH_USER_MODEL* to it. Do this before creating any migrations or running `manage.py migrate` for the first time.

Also, register the model in the app's `admin.py`:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from .models import User

admin.site.register(User, UserAdmin)
```

Changing to a custom user model mid-project

Changing *AUTH_USER_MODEL* after you've created database tables is possible, but can be complex, since it affects foreign keys and many-to-many relationships, for example.

This change can't be done automatically and requires manually fixing your schema, moving your data from the old user table, and possibly manually reapplying some migrations. See #25313 for an outline of the steps.

Due to limitations of Django's dynamic dependency feature for swappable models, the model referenced by *AUTH_USER_MODEL* must be created in the first migration of its app (usually called `0001_initial`); otherwise, you'll have dependency issues.

In addition, you may run into a `CircularDependencyError` when running your migrations as Django won't be able to automatically break the dependency loop due to the dynamic dependency. If you see this error, you should break the loop by moving the models depended on by your user model into a second migration. (You can try making two normal models that have a `ForeignKey` to each other and seeing how `makemigrations` resolves that circular dependency if you want to see how it's usually done.)

Reusable apps and `AUTH_USER_MODEL`

Reusable apps shouldn't implement a custom user model. A project may use many apps, and two reusable apps that implemented a custom user model couldn't be used together. If you need to store per user information in your app, use a *ForeignKey* or *OneToOneField* to `settings.AUTH_USER_MODEL` as described below.

Referencing the `User` model

If you reference *User* directly (for example, by referring to it in a foreign key), your code will not work in projects where the *AUTH_USER_MODEL* setting has been changed to a different user model.

`get_user_model()`

Instead of referring to *User* directly, you should reference the user model using `django.contrib.auth.get_user_model()`. This method will return the currently active user model – the custom user model if one is specified, or *User* otherwise.

When you define a foreign key or many-to-many relations to the user model, you should specify the custom model using the *AUTH_USER_MODEL* setting. For example:

```
from django.conf import settings
from django.db import models


class Article(models.Model):
    author = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
```

When connecting to signals sent by the user model, you should specify the custom model using the *AUTH_USER_MODEL* setting. For example:

```
from django.conf import settings
from django.db.models.signals import post_save


def post_save_receiver(sender, instance, created, **kwargs):
    pass


post_save.connect(post_save_receiver, sender=settings.AUTH_USER_MODEL)
```

Generally speaking, it's easiest to refer to the user model with the *AUTH_USER_MODEL* setting in code that's executed at import time, however, it's also possible to call `get_user_model()` while Django is importing models, so you could use `models.ForeignKey(get_user_model(), ...)`.

If your app is tested with multiple user models, using `@override_settings(AUTH_USER_MODEL=...)` for example, and you cache the result of `get_user_model()` in a module-level variable, you may need to listen to the *setting_changed* signal to clear the cache. For example:

```
from django.apps import apps
from django.contrib.auth import get_user_model
from django.core.signals import setting_changed
from django.dispatch import receiver


@receiver(setting_changed)
def user_model_swapped(*, setting, **kwargs):
    if setting == "AUTH_USER_MODEL":
        apps.clear_cache()
        from myapp import some_module
```

(continues on next page)

(continued from previous page)

```
some_module.UserModel = get_user_model()
```

Specifying a custom user model

When you start your project with a custom user model, stop to consider if this is the right choice for your project.

Keeping all user related information in one model removes the need for additional or more complex database queries to retrieve related models. On the other hand, it may be more suitable to store app-specific user information in a model that has a relation with your custom user model. That allows each app to specify its own user data requirements without potentially conflicting or breaking assumptions by other apps. It also means that you would keep your user model as simple as possible, focused on authentication, and following the minimum requirements Django expects custom user models to meet.

If you use the default authentication backend, then your model must have a single unique field that can be used for identification purposes. This can be a username, an email address, or any other unique attribute. A non-unique username field is allowed if you use a custom authentication backend that can support it.

The easiest way to construct a compliant custom user model is to inherit from *AbstractBaseUser*. *AbstractBaseUser* provides the core implementation of a user model, including hashed passwords and tokenized password resets. You must then provide some key implementation details:

class models.CustomUser

USERNAME_FIELD

A string describing the name of the field on the user model that is used as the unique identifier. This will usually be a username of some kind, but it can also be an email address, or any other unique identifier. The field must be unique (e.g. have `unique=True` set in its definition), unless you use a custom authentication backend that can support non-unique usernames.

In the following example, the field `identifier` is used as the identifying field:

```
class MyUser(AbstractBaseUser):
    identifier = models.CharField(max_length=40, unique=True)
    ...
    USERNAME_FIELD = "identifier"
```

EMAIL_FIELD

A string describing the name of the email field on the `User` model. This value is returned by *get_email_field_name()*.

REQUIRED_FIELDS

A list of the field names that will be prompted for when creating a user via the *createsuperuser*

management command. The user will be prompted to supply a value for each of these fields. It must include any field for which *blank* is `False` or undefined and may include additional fields you want prompted for when a user is created interactively. `REQUIRED_FIELDS` has no effect in other parts of Django, like creating a user in the admin.

For example, here is the partial definition for a user model that defines two required fields - a date of birth and height:

```
class MyUser(AbstractBaseUser):
    ...
    date_of_birth = models.DateField()
    height = models.FloatField()
    ...
    REQUIRED_FIELDS = ["date_of_birth", "height"]
```

> **Note**
>
> `REQUIRED_FIELDS` must contain all required fields on your user model, but should not contain the `USERNAME_FIELD` or `password` as these fields will always be prompted for.

`is_active`

A boolean attribute that indicates whether the user is considered "active". This attribute is provided as an attribute on `AbstractBaseUser` defaulting to `True`. How you choose to implement it will depend on the details of your chosen auth backends. See the documentation of the *is_active attribute on the built-in user model* for details.

`get_full_name()`

Optional. A longer formal identifier for the user such as their full name. If implemented, this appears alongside the username in an object's history in *django.contrib.admin*.

`get_short_name()`

Optional. A short, informal identifier for the user such as their first name. If implemented, this replaces the username in the greeting to the user in the header of *django.contrib.admin*.

> **Importing `AbstractBaseUser`**
>
> `AbstractBaseUser` and `BaseUserManager` are importable from `django.contrib.auth.base_user` so that they can be imported without including `django.contrib.auth` in *INSTALLED_APPS*.

The following attributes and methods are available on any subclass of *AbstractBaseUser*:

`class models.AbstractBaseUser`

get_username()

Returns the value of the field nominated by `USERNAME_FIELD`.

clean()

Normalizes the username by calling *normalize_username()*. If you override this method, be sure to call `super()` to retain the normalization.

classmethod get_email_field_name()

Returns the name of the email field specified by the *EMAIL_FIELD* attribute. Defaults to `'email'` if `EMAIL_FIELD` isn't specified.

classmethod normalize_username(username)

Applies NFKC Unicode normalization to usernames so that visually identical characters with different Unicode code points are considered identical.

is_authenticated

Read-only attribute which is always `True` (as opposed to `AnonymousUser.is_authenticated` which is always `False`). This is a way to tell if the user has been authenticated. This does not imply any permissions and doesn't check if the user is active or has a valid session. Even though normally you will check this attribute on `request.user` to find out whether it has been populated by the *AuthenticationMiddleware* (representing the currently logged-in user), you should know this attribute is `True` for any *User* instance.

is_anonymous

Read-only attribute which is always `False`. This is a way of differentiating *User* and *AnonymousUser* objects. Generally, you should prefer using *is_authenticated* to this attribute.

set_password(raw_password)

Sets the user's password to the given raw string, taking care of the password hashing. Doesn't save the *AbstractBaseUser* object.

When the raw_password is `None`, the password will be set to an unusable password, as if *set_unusable_password()* were used.

check_password(raw_password)

acheck_password(raw_password)

Asynchronous version: `acheck_password()`

Returns `True` if the given raw string is the correct password for the user. (This takes care of the password hashing in making the comparison.)

set_unusable_password()

Marks the user as having no password set. This isn't the same as having a blank string for a password. *check_password()* for this user will never return `True`. Doesn't save the *AbstractBaseUser* object.

You may need this if authentication for your application takes place against an existing external source such as an LDAP directory.

has_usable_password()

Returns `False` if *set_unusable_password()* has been called for this user.

get_session_auth_hash()

Returns an HMAC of the password field. Used for Session invalidation on password change.

get_session_auth_fallback_hash()

Yields the HMAC of the password field using *SECRET_KEY_FALLBACKS*. Used by `get_user()`.

AbstractUser subclasses *AbstractBaseUser*:

class models.**AbstractUser**

clean()

Normalizes the email by calling *BaseUserManager.normalize_email()*. If you override this method, be sure to call `super()` to retain the normalization.

Writing a manager for a custom user model

You should also define a custom manager for your user model. If your user model defines `username`, `email`, `is_staff`, `is_active`, `is_superuser`, `last_login`, and `date_joined` fields the same as Django's default user, you can install Django's *UserManager*; however, if your user model defines different fields, you'll need to define a custom manager that extends *BaseUserManager* providing two additional methods:

class models.**CustomUserManager**

create_user(username_field, password=None, **other_fields)

The prototype of `create_user()` should accept the username field, plus all required fields as arguments. For example, if your user model uses `email` as the username field, and has `date_of_birth` as a required field, then `create_user` should be defined as:

```
def create_user(self, email, date_of_birth, password=None):
    # create user here
    ...
```

create_superuser(username_field, password=None, **other_fields)

The prototype of `create_superuser()` should accept the username field, plus all required fields as arguments. For example, if your user model uses `email` as the username field, and has `date_of_birth` as a required field, then `create_superuser` should be defined as:

```
def create_superuser(self, email, date_of_birth, password=None):
    # create superuser here
    ...
```

For a *ForeignKey* in *USERNAME_FIELD* or *REQUIRED_FIELDS*, these methods receive the value of the *to_field* (the *primary_key* by default) of an existing instance.

BaseUserManager provides the following utility methods:

class models.**BaseUserManager**

classmethod normalize_email(email)

Normalizes email addresses by lowercasing the domain portion of the email address.

get_by_natural_key(username)

aget_by_natural_key(username)

Asynchronous version: `aget_by_natural_key()`

Retrieves a user instance using the contents of the field nominated by `USERNAME_FIELD`.

`aget_by_natural_key()` method was added.

Extending Django's default `User`

If you're entirely happy with Django's *User* model, but you want to add some additional profile information, you could subclass *django.contrib.auth.models.AbstractUser* and add your custom profile fields, although we'd recommend a separate model as described in Specifying a custom user model. `AbstractUser` provides the full implementation of the default *User* as an abstract model.

Custom users and the built-in auth forms

Django's built-in forms and views make certain assumptions about the user model that they are working with.

The following forms are compatible with any subclass of *AbstractBaseUser*:

- *AuthenticationForm*: Uses the username field specified by *USERNAME_FIELD*.
- *SetPasswordForm*
- *PasswordChangeForm*
- *AdminPasswordChangeForm*

The following forms make assumptions about the user model and can be used as-is if those assumptions are met:

- *PasswordResetForm*: Assumes that the user model has a field that stores the user's email address with the name returned by *get_email_field_name()* (`email` by default) that can be used to identify the user and a boolean field named `is_active` to prevent password resets for inactive users.

Finally, the following forms are tied to *User* and need to be rewritten or extended to work with a custom user model:

- *UserCreationForm*

- *UserChangeForm*

If your custom user model is a subclass of `AbstractUser`, then you can extend these forms in this manner:

```
from django.contrib.auth.forms import UserCreationForm
from myapp.models import CustomUser


class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ("custom_field",)
```

Custom users and `django.contrib.admin`

If you want your custom user model to also work with the admin, your user model must define some additional attributes and methods. These methods allow the admin to control access of the user to admin content:

class models.**CustomUser**

is_staff

Returns `True` if the user is allowed to have access to the admin site.

is_active

Returns `True` if the user account is currently active.

has_perm(perm, obj=None):

Returns `True` if the user has the named permission. If `obj` is provided, the permission needs to be checked against a specific object instance.

has_module_perms(app_label):

Returns `True` if the user has permission to access models in the given app.

You will also need to register your custom user model with the admin. If your custom user model extends `django.contrib.auth.models.AbstractUser`, you can use Django's existing `django.contrib.auth.admin.UserAdmin` class. However, if your user model extends *AbstractBaseUser*, you'll need to define a custom `ModelAdmin` class. It may be possible to subclass the default `django.contrib.auth.admin.UserAdmin`; however, you'll need to override any of the definitions that refer to fields on `django.contrib.auth.models.AbstractUser` that aren't on your custom user class.

> **Note**
>
> If you are using a custom `ModelAdmin` which is a subclass of `django.contrib.auth.admin.UserAdmin`, then you need to add your custom fields to `fieldsets` (for fields to be used in editing users) and to `add_fieldsets` (for fields to be used when creating a user). For example:

```
from django.contrib.auth.admin import UserAdmin


class CustomUserAdmin(UserAdmin):
    ...
    fieldsets = UserAdmin.fieldsets + ((None, {"fields": ["custom_field"]}),)
    add_fieldsets = UserAdmin.add_fieldsets + ((None, {"fields": ["custom_field"]}),)
```

See a full example for more details.

Custom users and permissions

To make it easy to include Django's permission framework into your own user class, Django provides *PermissionsMixin*. This is an abstract model you can include in the class hierarchy for your user model, giving you all the methods and database fields necessary to support Django's permission model.

PermissionsMixin provides the following methods and attributes:

class models.**PermissionsMixin**

is_superuser

Boolean. Designates that this user has all permissions without explicitly assigning them.

get_user_permissions(obj=None)

Returns a set of permission strings that the user has directly.

If `obj` is passed in, only returns the user permissions for this specific object.

get_group_permissions(obj=None)

Returns a set of permission strings that the user has, through their groups.

If `obj` is passed in, only returns the group permissions for this specific object.

get_all_permissions(obj=None)

Returns a set of permission strings that the user has, both through group and user permissions.

If `obj` is passed in, only returns the permissions for this specific object.

has_perm(perm, obj=None)

Returns `True` if the user has the specified permission, where `perm` is in the format `"<app label>.<permission codename>"` (see permissions). If *User.is_active* and *is_superuser* are both `True`, this method always returns `True`.

If `obj` is passed in, this method won't check for a permission for the model, but for this specific object.

has_perms(perm_list, obj=None)

Returns `True` if the user has each of the specified permissions, where each perm is in the format `"<app label>.<permission codename>"`. If *User.is_active* and *is_superuser* are both `True`, this method always returns `True`.

If `obj` is passed in, this method won't check for permissions for the model, but for the specific object.

has_module_perms(package_name)

Returns `True` if the user has any permissions in the given package (the Django app label). If *User.is_active* and *is_superuser* are both `True`, this method always returns `True`.

PermissionsMixin and **ModelBackend**

If you don't include the *PermissionsMixin*, you must ensure you don't invoke the permissions methods on `ModelBackend`. `ModelBackend` assumes that certain fields are available on your user model. If your user model doesn't provide those fields, you'll receive database errors when you check permissions.

Custom users and proxy models

One limitation of custom user models is that installing a custom user model will break any proxy model extending *User*. Proxy models must be based on a concrete base class; by defining a custom user model, you remove the ability of Django to reliably identify the base class.

If your project uses proxy models, you must either modify the proxy to extend the user model that's in use in your project, or merge your proxy's behavior into your *User* subclass.

A full example

Here is an example of an admin-compliant custom user app. This user model uses an email address as the username, and has a required date of birth; it provides no permission checking beyond an `admin` flag on the user account. This model would be compatible with all the built-in auth forms and views, except for the user creation forms. This example illustrates how most of the components work together, but is not intended to be copied directly into projects for production use.

This code would all live in a `models.py` file for a custom authentication app:

```
from django.db import models
from django.contrib.auth.models import BaseUserManager, AbstractBaseUser


class MyUserManager(BaseUserManager):
    def create_user(self, email, date_of_birth, password=None):
        """
```

(continues on next page)

(continued from previous page)

```
        Creates and saves a User with the given email, date of
        birth and password.
        """
        if not email:
            raise ValueError("Users must have an email address")

        user = self.model(
            email=self.normalize_email(email),
            date_of_birth=date_of_birth,
        )

        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, date_of_birth, password=None):
        """
        Creates and saves a superuser with the given email, date of
        birth and password.
        """
        user = self.create_user(
            email,
            password=password,
            date_of_birth=date_of_birth,
        )
        user.is_admin = True
        user.save(using=self._db)
        return user


class MyUser(AbstractBaseUser):
    email = models.EmailField(
        verbose_name="email address",
        max_length=255,
        unique=True,
    )
    date_of_birth = models.DateField()
    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)
```

(continues on next page)

(continued from previous page)

```
    objects = MyUserManager()

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["date_of_birth"]

    def __str__(self):
        return self.email

    def has_perm(self, perm, obj=None):
        "Does the user have a specific permission?"
        # Simplest possible answer: Yes, always
        return True

    def has_module_perms(self, app_label):
        "Does the user have permissions to view the app `app_label`?"
        # Simplest possible answer: Yes, always
        return True

    @property
    def is_staff(self):
        "Is the user a member of staff?"
        # Simplest possible answer: All admins are staff
        return self.is_admin
```

Then, to register this custom user model with Django's admin, the following code would be required in the app's `admin.py` file:

```
from django import forms
from django.contrib import admin
from django.contrib.auth.models import Group
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.forms import ReadOnlyPasswordHashField
from django.core.exceptions import ValidationError

from customauth.models import MyUser


class UserCreationForm(forms.ModelForm):
    """A form for creating new users. Includes all the required
    fields, plus a repeated password."""
```

(continues on next page)

(continued from previous page)

```
    password1 = forms.CharField(label="Password", widget=forms.PasswordInput)
    password2 = forms.CharField(
        label="Password confirmation", widget=forms.PasswordInput
    )

    class Meta:
        model = MyUser
        fields = ["email", "date_of_birth"]

    def clean_password2(self):
        # Check that the two password entries match
        password1 = self.cleaned_data.get("password1")
        password2 = self.cleaned_data.get("password2")
        if password1 and password2 and password1 != password2:
            raise ValidationError("Passwords don't match")
        return password2

    def save(self, commit=True):
        # Save the provided password in hashed format
        user = super().save(commit=False)
        user.set_password(self.cleaned_data["password1"])
        if commit:
            user.save()
        return user


class UserChangeForm(forms.ModelForm):
    """A form for updating users. Includes all the fields on
    the user, but replaces the password field with admin's
    disabled password hash display field.
    """

    password = ReadOnlyPasswordHashField()

    class Meta:
        model = MyUser
        fields = ["email", "password", "date_of_birth", "is_active", "is_admin"]
```

(continues on next page)

(continued from previous page)

```
class UserAdmin(BaseUserAdmin):
    # The forms to add and change user instances
    form = UserChangeForm
    add_form = UserCreationForm

    # The fields to be used in displaying the User model.
    # These override the definitions on the base UserAdmin
    # that reference specific fields on auth.User.
    list_display = ["email", "date_of_birth", "is_admin"]
    list_filter = ["is_admin"]
    fieldsets = [
        (None, {"fields": ["email", "password"]}),
        ("Personal info", {"fields": ["date_of_birth"]}),
        ("Permissions", {"fields": ["is_admin"]}),
    ]
    # add_fieldsets is not a standard ModelAdmin attribute. UserAdmin
    # overrides get_fieldsets to use this attribute when creating a user.
    add_fieldsets = [
        (
            None,
            {
                "classes": ["wide"],
                "fields": ["email", "date_of_birth", "password1", "password2"],
            },
        ),
    ]
    search_fields = ["email"]
    ordering = ["email"]
    filter_horizontal = []


# Now register the new UserAdmin...
admin.site.register(MyUser, UserAdmin)
# ... and, since we're not using Django's built-in permissions,
# unregister the Group model from admin.
admin.site.unregister(Group)
```

Finally, specify the custom model as the default user model for your project using the *AUTH_USER_MODEL* setting in your `settings.py`:

```
AUTH_USER_MODEL = "customauth.MyUser"
```

Adding an async interface

To optimize performance when called from an async context authentication, backends can implement async versions of each function - `aget_user(user_id)` and `aauthenticate(request, **credentials)`. When an authentication backend extends `BaseBackend` and async versions of these functions are not provided, they will be automatically synthesized with `sync_to_async`. This has performance penalties.

While an async interface is optional, a synchronous interface is always required. There is no automatic synthesis for a synchronous interface if an async interface is implemented.

Django's out-of-the-box authentication backends have native async support. If these native backends are extended take special care to make sure the async versions of modified functions are modified as well. Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions. This section of the documentation explains how the default implementation works out of the box, as well as how to extend and customize it to suit your project's needs.

3.10.4 Overview

The Django authentication system handles both authentication and authorization. Briefly, authentication verifies a user is who they claim to be, and authorization determines what an authenticated user is allowed to do. Here the term authentication is used to refer to both tasks.

The auth system consists of:

- Users
- Permissions: Binary (yes/no) flags designating whether a user may perform a certain task.
- Groups: A generic way of applying labels and permissions to more than one user.
- A configurable password hashing system
- Forms and view tools for logging in users, or restricting content
- A pluggable backend system

The authentication system in Django aims to be very generic and doesn't provide some features commonly found in web authentication systems. Solutions for some of these common problems have been implemented in third-party packages:

- Password strength checking
- Throttling of login attempts
- Authentication against third-parties (OAuth, for example)
- Object-level permissions

3.10.5 Installation

Authentication support is bundled as a Django contrib module in `django.contrib.auth`. By default, the required configuration is already included in the `settings.py` generated by *`django-admin startproject`*, these consist of two items listed in your *`INSTALLED_APPS`* setting:

1. `'django.contrib.auth'` contains the core of the authentication framework, and its default models.
2. `'django.contrib.contenttypes'` is the Django content type system, which allows permissions to be associated with models you create.

and these items in your *`MIDDLEWARE`* setting:

1. *`SessionMiddleware`* manages sessions across requests.
2. *`AuthenticationMiddleware`* associates users with requests using sessions.

With these settings in place, running the command `manage.py migrate` creates the necessary database tables for auth related models and permissions for any models defined in your installed apps.

3.10.6 Usage

Using Django's default implementation

- Working with User objects
- Permissions and authorization
- Authentication in web requests
- Managing users in the admin

API reference for the default implementation

Customizing Users and authentication

Password management in Django

3.11 Django's cache framework

A fundamental trade-off in dynamic websites is, well, they're dynamic. Each time a user requests a page, the web server makes all sorts of calculations – from database queries to template rendering to business logic – to create the page that your site's visitor sees. This is a lot more expensive, from a processing-overhead perspective, than your standard read-a-file-off-the-filesystem server arrangement.

For most web applications, this overhead isn't a big deal. Most web applications aren't `washingtonpost.com` or `slashdot.org`; they're small- to medium-sized sites with so-so traffic. But for medium- to high-traffic sites, it's essential to cut as much overhead as possible.

That's where caching comes in.

To cache something is to save the result of an expensive calculation so that you don't have to perform the calculation next time. Here's some pseudocode explaining how this would work for a dynamically generated web page:

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django comes with a robust cache system that lets you save dynamic pages so they don't have to be calculated for each request. For convenience, Django offers different levels of cache granularity: You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Django also works well with "downstream" caches, such as Squid and browser-based caches. These are the types of caches that you don't directly control but to which you can provide hints (via HTTP headers) about which parts of your site should be cached, and how.

See also

The Cache Framework design philosophy explains a few of the design decisions of the framework.

3.11.1 Setting up the cache

The cache system requires a small amount of setup. Namely, you have to tell it where your cached data should live – whether in a database, on the filesystem or directly in memory. This is an important decision that affects your cache's performance; yes, some cache types are faster than others.

Your cache preference goes in the *CACHES* setting in your settings file. Here's an explanation of all available values for *CACHES*.

Memcached

Memcached is an entirely memory-based cache server, originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It is used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached runs as a daemon and is allotted a specified amount of RAM. All it does is provide a fast interface for adding, retrieving and deleting data in the cache. All data is stored directly in memory, so there's no overhead of database or filesystem usage.

After installing Memcached itself, you'll need to install a Memcached binding. There are several Python Memcached bindings available; the two supported by Django are pylibmc and pymemcache.

To use Memcached with Django:

- Set *BACKEND* to `django.core.cache.backends.memcached.PyMemcacheCache` or `django.core.cache.backends.memcached.PyLibMCCache` (depending on your chosen memcached binding)
- Set *LOCATION* to `ip:port` values, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running, or to a `unix:path` value, where `path` is the path to a Memcached Unix socket file.

In this example, Memcached is running on localhost (127.0.0.1) port 11211, using the `pymemcache` binding:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": "127.0.0.1:11211",
    }
}
```

In this example, Memcached is available through a local Unix socket file `/tmp/memcached.sock` using the `pymemcache` binding:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": "unix:/tmp/memcached.sock",
    }
}
```

One excellent feature of Memcached is its ability to share a cache over multiple servers. This means you can run Memcached daemons on multiple machines, and the program will treat the group of machines as a single cache, without the need to duplicate cache values on each machine. To take advantage of this feature, include all server addresses in *LOCATION*, either as a semicolon or comma delimited string, or as a list.

In this example, the cache is shared over Memcached instances running on IP address 172.19.26.240 and 172.19.26.242, both on port 11211:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": [
            "172.19.26.240:11211",
            "172.19.26.242:11211",
```

(continues on next page)

(continued from previous page)

```
        ],
    }
}
```

In the following example, the cache is shared over Memcached instances running on the IP addresses 172.19.26.240 (port 11211), 172.19.26.242 (port 11212), and 172.19.26.244 (port 11213):

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": [
            "172.19.26.240:11211",
            "172.19.26.242:11212",
            "172.19.26.244:11213",
        ],
    }
}
```

By default, the `PyMemcacheCache` backend sets the following options (you can override them in your *`OPTIONS`*):

```
"OPTIONS": {
    "allow_unicode_keys": True,
    "default_noreply": False,
    "serde": pymemcache.serde.pickle_serde,
}
```

A final point about Memcached is that memory-based caching has a disadvantage: because the cached data is stored in memory, the data will be lost if your server crashes. Clearly, memory isn't intended for permanent data storage, so don't rely on memory-based caching as your only data storage. Without a doubt, none of the Django caching backends should be used for permanent storage – they're all intended to be solutions for caching, not storage – but we point this out here because memory-based caching is particularly temporary.

Redis

Redis is an in-memory database that can be used for caching. To begin you'll need a Redis server running either locally or on a remote machine.

After setting up the Redis server, you'll need to install Python bindings for Redis. redis-py is the binding supported natively by Django. Installing the hiredis-py package is also recommended.

To use Redis as your cache backend with Django:

- Set *`BACKEND`* to `django.core.cache.backends.redis.RedisCache`.

- Set *LOCATION* to the URL pointing to your Redis instance, using the appropriate scheme. See the `redis-py` docs for details on the available schemes.

For example, if Redis is running on localhost (127.0.0.1) port 6379:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379",
    }
}
```

Often Redis servers are protected with authentication. In order to supply a username and password, add them in the `LOCATION` along with the URL:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": "redis://username:password@127.0.0.1:6379",
    }
}
```

If you have multiple Redis servers set up in the replication mode, you can specify the servers either as a semicolon or comma delimited string, or as a list. While using multiple servers, write operations are performed on the first server (leader). Read operations are performed on the other servers (replicas) chosen at random:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
        "LOCATION": [
            "redis://127.0.0.1:6379",  # leader
            "redis://127.0.0.1:6378",  # read-replica 1
            "redis://127.0.0.1:6377",  # read-replica 2
        ],
    }
}
```

Database caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.

To use a database table as your cache backend:

- Set *BACKEND* to `django.core.cache.backends.db.DatabaseCache`

- Set *LOCATION* to `tablename`, the name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

In this example, the cache table's name is `my_cache_table`:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.db.DatabaseCache",
        "LOCATION": "my_cache_table",
    }
}
```

Unlike other cache backends, the database cache does not support automatic culling of expired entries at the database level. Instead, expired cache entries are culled each time `add()`, `set()`, or `touch()` is called.

Creating the cache table

Before using the database cache, you must create the cache table with this command:

```
python manage.py createcachetable
```

This creates a table in your database that is in the proper format that Django's database-cache system expects. The name of the table is taken from *LOCATION*.

If you are using multiple database caches, *createcachetable* creates one table for each cache.

If you are using multiple databases, *createcachetable* observes the `allow_migrate()` method of your database routers (see below).

Like *migrate*, *createcachetable* won't touch an existing table. It will only create missing tables.

To print the SQL that would be run, rather than run it, use the *createcachetable --dry-run* option.

Multiple databases

If you use database caching with multiple databases, you'll also need to set up routing instructions for your database cache table. For the purposes of routing, the database cache table appears as a model named `CacheEntry`, in an application named `django_cache`. This model won't appear in the models cache, but the model details can be used for routing purposes.

For example, the following router would direct all cache read operations to `cache_replica`, and all write operations to `cache_primary`. The cache table will only be synchronized onto `cache_primary`:

```
class CacheRouter:
    """A router to control all database cache operations"""

    def db_for_read(self, model, **hints):
```

(continues on next page)

(continued from previous page)

```
        "All cache read operations go to the replica"
        if model._meta.app_label == "django_cache":
            return "cache_replica"
        return None

    def db_for_write(self, model, **hints):
        "All cache write operations go to primary"
        if model._meta.app_label == "django_cache":
            return "cache_primary"
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        "Only install the cache model on primary"
        if app_label == "django_cache":
            return db == "cache_primary"
        return None
```

If you don't specify routing directions for the database cache model, the cache backend will use the `default` database.

And if you don't use the database cache backend, you don't need to worry about providing routing instructions for the database cache model.

Filesystem caching

The file-based backend serializes and stores each cache value as a separate file. To use this backend set *BACKEND* to `"django.core.cache.backends.filebased.FileBasedCache"` and *LOCATION* to a suitable directory. For example, to store cached data in `/var/tmp/django_cache`, use this setting:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "/var/tmp/django_cache",
    }
}
```

If you're on Windows, put the drive letter at the beginning of the path, like this:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "c:/foo/bar",
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The directory path should be absolute – that is, it should start at the root of your filesystem. It doesn't matter whether you put a slash at the end of the setting.

Make sure the directory pointed-to by this setting either exists and is readable and writable, or that it can be created by the system user under which your web server runs. Continuing the above example, if your server runs as the user `apache`, make sure the directory `/var/tmp/django_cache` exists and is readable and writable by the user `apache`, or that it can be created by the user `apache`.

> **Warning**
>
> When the cache *LOCATION* is contained within *MEDIA_ROOT*, *STATIC_ROOT*, or *STATICFILES_FINDERS*, sensitive data may be exposed.
>
> An attacker who gains access to the cache file can not only falsify HTML content, which your site will trust, but also remotely execute arbitrary code, as the data is serialized using `pickle`.

> **Warning**
>
> Filesystem caching may become slow when storing a large number of files. If you run into this problem, consider using a different caching mechanism. You can also subclass FileBasedCache and improve the culling strategy.

Local-memory caching

This is the default cache if another is not specified in your settings file. If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. This cache is per-process (see below) and thread-safe. To use it, set *BACKEND* to `"django.core.cache.backends.locmem.LocMemCache"`. For example:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.locmem.LocMemCache",
        "LOCATION": "unique-snowflake",
    }
}
```

The cache *LOCATION* is used to identify individual memory stores. If you only have one `locmem` cache, you can omit the *LOCATION*; however, if you have more than one local memory cache, you will need to assign a name to at least one of them in order to keep them separate.

The cache uses a least-recently-used (LRU) culling strategy.

Note that each process will have its own private cache instance, which means no cross-process caching is possible. This also means the local memory cache isn't particularly memory-efficient, so it's probably not a good choice for production environments. It's nice for development.

Dummy caching (for development)

Finally, Django comes with a "dummy" cache that doesn't actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don't want to cache and don't want to have to change your code to special-case the latter. To activate dummy caching, set *BACKEND* like so:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.dummy.DummyCache",
    }
}
```

Using a custom cache backend

While Django includes support for a number of cache backends out-of-the-box, sometimes you might want to use a customized cache backend. To use an external cache backend with Django, use the Python import path as the *BACKEND* of the *CACHES* setting, like so:

```
CACHES = {
    "default": {
        "BACKEND": "path.to.backend",
    }
}
```

If you're building your own backend, you can use the standard cache backends as reference implementations. You'll find the code in the django/core/cache/backends/ directory of the Django source.

Note: Without a really compelling reason, such as a host that doesn't support them, you should stick to the cache backends included with Django. They've been well-tested and are well-documented.

Cache arguments

Each cache backend can be given additional arguments to control caching behavior. These arguments are provided as additional keys in the *CACHES* setting. Valid arguments are as follows:

- *TIMEOUT*: The default timeout, in seconds, to use for the cache. This argument defaults to `300` seconds (5 minutes). You can set `TIMEOUT` to `None` so that, by default, cache keys never expire. A value of `0`

causes keys to immediately expire (effectively "don't cache").

- *OPTIONS*: Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.

  Cache backends that implement their own culling strategy (i.e., the `locmem`, `filesystem` and `database` backends) will honor the following options:

  - `MAX_ENTRIES`: The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to `300`.
  - `CULL_FREQUENCY`: The fraction of entries that are culled when `MAX_ENTRIES` is reached. The actual ratio is `1 / CULL_FREQUENCY`, so set `CULL_FREQUENCY` to `2` to cull half the entries when `MAX_ENTRIES` is reached. This argument should be an integer and defaults to `3`.

    A value of `0` for `CULL_FREQUENCY` means that the entire cache will be dumped when `MAX_ENTRIES` is reached. On some backends (`database` in particular) this makes culling much faster at the expense of more cache misses.

  The Memcached and Redis backends pass the contents of *OPTIONS* as keyword arguments to the client constructors, allowing for more advanced control of client behavior. For example usage, see below.

- *KEY_PREFIX*: A string that will be automatically included (prepended by default) to all cache keys used by the Django server.

  See the cache documentation for more information.

- *VERSION*: The default version number for cache keys generated by the Django server.

  See the cache documentation for more information.

- *KEY_FUNCTION* A string containing a dotted path to a function that defines how to compose a prefix, version and key into a final cache key.

  See the cache documentation for more information.

In this example, a filesystem backend is being configured with a timeout of 60 seconds, and a maximum capacity of 1000 items:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "/var/tmp/django_cache",
        "TIMEOUT": 60,
        "OPTIONS": {"MAX_ENTRIES": 1000},
    }
}
```

Here's an example configuration for a `pylibmc` based backend that enables the binary protocol, SASL authentication, and the `ketama` behavior mode:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyLibMCCache",
        "LOCATION": "127.0.0.1:11211",
        "OPTIONS": {
            "binary": True,
            "username": "user",
            "password": "pass",
            "behaviors": {
                "ketama": True,
            },
        },
    }
}
```

Here's an example configuration for a `pymemcache` based backend that enables client pooling (which may improve performance by keeping clients connected), treats memcache/network errors as cache misses, and sets the `TCP_NODELAY` flag on the connection's socket:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION": "127.0.0.1:11211",
        "OPTIONS": {
            "no_delay": True,
            "ignore_exc": True,
            "max_pool_size": 4,
            "use_pooling": True,
        },
    }
}
```

Here's an example configuration for a `redis` based backend that selects database `10` (by default Redis ships with 16 logical databases), and sets a custom connection pool class (`redis.ConnectionPool` is used by default):

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache",
```

(continues on next page)

(continued from previous page)

```
        "LOCATION": "redis://127.0.0.1:6379",
        "OPTIONS": {
            "db": "10",
            "pool_class": "redis.BlockingConnectionPool",
        },
    }
}
```

3.11.2 The per-site cache

Once the cache is set up, the simplest way to use caching is to cache your entire site. You'll need to add `'django.middleware.cache.UpdateCacheMiddleware'` and `'django.middleware.cache.FetchFromCacheMiddleware'` to your *MIDDLEWARE* setting, as in this example:

```
MIDDLEWARE = [
    "django.middleware.cache.UpdateCacheMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.cache.FetchFromCacheMiddleware",
]
```

> **Note**
>
> No, that's not a typo: the "update" middleware must be first in the list, and the "fetch" middleware must be last. The details are a bit obscure, but see Order of MIDDLEWARE below if you'd like the full story.

Then, add the following required settings to your Django settings file:

- *CACHE_MIDDLEWARE_ALIAS* – The cache alias to use for storage.
- *CACHE_MIDDLEWARE_SECONDS* – The integer number of seconds each page should be cached.
- *CACHE_MIDDLEWARE_KEY_PREFIX* – If the cache is shared across multiple sites using the same Django installation, set this to the name of the site, or some other string that is unique to this Django instance, to prevent key collisions. Use an empty string if you don't care.

`FetchFromCacheMiddleware` caches GET and HEAD responses with status 200, where the request and response headers allow. Responses to requests for the same URL with different query parameters are considered to be unique pages and are cached separately. This middleware expects that a HEAD request is answered with the same response headers as the corresponding GET request; in which case it can return a cached GET response for HEAD request.

Additionally, `UpdateCacheMiddleware` automatically sets a few headers in each *HttpResponse* which affect downstream caches:

- Sets the `Expires` header to the current date/time plus the defined *CACHE_MIDDLEWARE_SECONDS*.
- Sets the `Cache-Control` header to give a max age for the page – again, from the *CACHE_MIDDLEWARE_SECONDS* setting.

See Middleware for more on middleware.

If a view sets its own cache expiry time (i.e. it has a `max-age` section in its `Cache-Control` header) then the page will be cached until the expiry time, rather than *CACHE_MIDDLEWARE_SECONDS*. Using the decorators in `django.views.decorators.cache` you can easily set a view's expiry time (using the *cache_control()* decorator) or disable caching for a view (using the *never_cache()* decorator). See the using other headers section for more on these decorators.

If *USE_I18N* is set to `True` then the generated cache key will include the name of the active language – see also How Django discovers language preference). This allows you to easily cache multilingual sites without having to create the cache key yourself.

Cache keys also include the current time zone when *USE_TZ* is set to `True`.

3.11.3 The per-view cache

`django.views.decorators.cache.cache_page`(timeout, *, cache=None, key_prefix=None)

A more granular way to use the caching framework is by caching the output of individual views. `django.views.decorators.cache` defines a `cache_page` decorator that will automatically cache the view's response for you:

```
from django.views.decorators.cache import cache_page


@cache_page(60 * 15)
def my_view(request): ...
```

`cache_page` takes a single argument: the cache timeout, in seconds. In the above example, the result of the `my_view()` view will be cached for 15 minutes. (Note that we've written it as `60 * 15` for the purpose of readability. `60 * 15` will be evaluated to `900` – that is, 15 minutes multiplied by 60 seconds per minute.)

The cache timeout set by `cache_page` takes precedence over the `max-age` directive from the `Cache-Control` header.

The per-view cache, like the per-site cache, is keyed off of the URL. If multiple URLs point at the same view, each URL will be cached separately. Continuing the `my_view` example, if your URLconf looks like this:

```
urlpatterns = [
    path("foo/<int:code>/", my_view),
]
```

then requests to `/foo/1/` and `/foo/23/` will be cached separately, as you may expect. But once a particular URL (e.g., `/foo/23/`) has been requested, subsequent requests to that URL will use the cache.

`cache_page` can also take an optional keyword argument, `cache`, which directs the decorator to use a specific cache (from your *CACHES* setting) when caching view results. By default, the `default` cache will be used, but you can specify any cache you want:

```
@cache_page(60 * 15, cache="special_cache")
def my_view(request): ...
```

You can also override the cache prefix on a per-view basis. `cache_page` takes an optional keyword argument, `key_prefix`, which works in the same way as the *CACHE_MIDDLEWARE_KEY_PREFIX* setting for the middleware. It can be used like this:

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request): ...
```

The `key_prefix` and `cache` arguments may be specified together. The `key_prefix` argument and the *KEY_PREFIX* specified under *CACHES* will be concatenated.

Additionally, `cache_page` automatically sets `Cache-Control` and `Expires` headers in the response which affect downstream caches.

Specifying per-view cache in the URLconf

The examples in the previous section have hard-coded the fact that the view is cached, because `cache_page` alters the `my_view` function in place. This approach couples your view to the cache system, which is not ideal for several reasons. For instance, you might want to reuse the view functions on another, cache-less site, or you might want to distribute the views to people who might want to use them without being cached. The solution to these problems is to specify the per-view cache in the URLconf rather than next to the view functions themselves.

You can do so by wrapping the view function with `cache_page` when you refer to it in the URLconf. Here's the old URLconf from earlier:

```
urlpatterns = [
    path("foo/<int:code>/", my_view),
]
```

Here's the same thing, with `my_view` wrapped in `cache_page`:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
```

(continues on next page)

(continued from previous page)

```
    path("foo/<int:code>/", cache_page(60 * 15)(my_view)),
]
```

3.11.4 Template fragment caching

If you're after even more control, you can also cache template fragments using the `cache` template tag. To give your template access to this tag, put `{% load cache %}` near the top of your template.

The `{% cache %}` template tag caches the contents of the block for a given amount of time. It takes at least two arguments: the cache timeout, in seconds, and the name to give the cache fragment. The fragment is cached forever if timeout is `None`. The name will be taken as is, do not use a variable. For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

Sometimes you might want to cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment. For example, you might want a separate cached copy of the sidebar used in the previous example for every user of your site. Do this by passing one or more additional arguments, which may be variables with or without filters, to the `{% cache %}` template tag to uniquely identify the cache fragment:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

If *USE_I18N* is set to `True` the per-site middleware cache will respect the active language. For the `cache` template tag you could use one of the translation-specific variables available in templates to achieve the same result:

```
{% load i18n %}
{% load cache %}

{% get_current_language as LANGUAGE_CODE %}

{% cache 600 welcome LANGUAGE_CODE %}
    {% translate "Welcome to example.com" %}
{% endcache %}
```

The cache timeout can be a template variable, as long as the template variable resolves to an integer value.

For example, if the template variable `my_timeout` is set to the value `600`, then the following two examples are equivalent:

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

This feature is useful in avoiding repetition in templates. You can set the timeout in a variable, in one place, and reuse that value.

By default, the cache tag will try to use the cache called "template_fragments". If no such cache exists, it will fall back to using the default cache. You may select an alternate cache backend to use with the `using` keyword argument, which must be the last argument to the tag.

```
{% cache 300 local-thing ...  using="localcache" %}
```

It is considered an error to specify a cache name that is not configured.

`django.core.cache.utils.make_template_fragment_key`(fragment_name, vary_on=None)

If you want to obtain the cache key used for a cached fragment, you can use `make_template_fragment_key`. `fragment_name` is the same as second argument to the `cache` template tag; `vary_on` is a list of all additional arguments passed to the tag. This function can be useful for invalidating or overwriting a cached item, for example:

```
>>> from django.core.cache import cache
>>> from django.core.cache.utils import make_template_fragment_key
# cache key for {% cache 500 sidebar username %}
>>> key = make_template_fragment_key("sidebar", [username])
>>> cache.delete(key)  # invalidates cached template fragment
True
```

3.11.5 The low-level cache API

Sometimes, caching an entire rendered page doesn't gain you very much and is, in fact, inconvenient overkill.

Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn't want to cache the entire result (since some of the data changes often), but you'd still want to cache the results that rarely change.

For cases like this, Django exposes a low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python objects can be pickled; refer to the Python documentation for more information about pickling.)

Accessing the cache

`django.core.cache.caches`

You can access the caches configured in the *CACHES* setting through a dict-like object: `django.core.cache.caches`. Repeated requests for the same alias in the same thread will return the same object.

```
>>> from django.core.cache import caches
>>> cache1 = caches["myalias"]
>>> cache2 = caches["myalias"]
>>> cache1 is cache2
True
```

If the named key does not exist, `InvalidCacheBackendError` will be raised.

To provide thread-safety, a different instance of the cache backend will be returned for each thread.

`django.core.cache.cache`

As a shortcut, the default cache is available as `django.core.cache.cache`:

```
>>> from django.core.cache import cache
```

This object is equivalent to `caches['default']`.

Basic usage

The basic interface is:

`cache.set`(key, value, timeout=DEFAULT_TIMEOUT, version=None)

```
>>> cache.set("my_key", "hello, world!", 30)
```

`cache.get`(key, default=None, version=None)

```
>>> cache.get("my_key")
'hello, world!'
```

`key` should be a `str`, and `value` can be any picklable Python object.

The `timeout` argument is optional and defaults to the `timeout` argument of the appropriate backend in the *CACHES* setting (explained above). It's the number of seconds the value should be stored in the cache. Passing in `None` for `timeout` will cache the value forever. A `timeout` of `0` won't cache the value.

If the object doesn't exist in the cache, `cache.get()` returns `None`:

```
>>> # Wait 30 seconds for 'my_key' to expire...
>>> cache.get("my_key")
None
```

If you need to determine whether the object exists in the cache and you have stored a literal value `None`, use a sentinel object as the default:

```
>>> sentinel = object()
>>> cache.get("my_key", sentinel) is sentinel
False
>>> # Wait 30 seconds for 'my_key' to expire...
>>> cache.get("my_key", sentinel) is sentinel
True
```

`cache.get()` can take a `default` argument. This specifies which value to return if the object doesn't exist in the cache:

```
>>> cache.get("my_key", "has expired")
'has expired'
```

`cache.add`(key, value, timeout=DEFAULT_TIMEOUT, version=None)

To add a key only if it doesn't already exist, use the `add()` method. It takes the same parameters as `set()`, but it will not attempt to update the cache if the key specified is already present:

```
>>> cache.set("add_key", "Initial value")
>>> cache.add("add_key", "New value")
>>> cache.get("add_key")
'Initial value'
```

If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise.

`cache.get_or_set`(key, default, timeout=DEFAULT_TIMEOUT, version=None)

If you want to get a key's value or set a value if the key isn't in the cache, there is the `get_or_set()` method. It takes the same parameters as `get()` but the default is set as the new cache value for that key, rather than returned:

```
>>> cache.get("my_new_key")  # returns None
>>> cache.get_or_set("my_new_key", "my new value", 100)
'my new value'
```

You can also pass any callable as a default value:

```
>>> import datetime
>>> cache.get_or_set("some-timestamp-key", datetime.datetime.now)
datetime.datetime(2014, 12, 11, 0, 15, 49, 457920)
```

cache.**get_many**(keys, version=None)

There's also a `get_many()` interface that only hits the cache once. `get_many()` returns a dictionary with all the keys you asked for that actually exist in the cache (and haven't expired):

```
>>> cache.set("a", 1)
>>> cache.set("b", 2)
>>> cache.set("c", 3)
>>> cache.get_many(["a", "b", "c"])
{'a': 1, 'b': 2, 'c': 3}
```

cache.**set_many**(dict, timeout)

To set multiple values more efficiently, use `set_many()` to pass a dictionary of key-value pairs:

```
>>> cache.set_many({"a": 1, "b": 2, "c": 3})
>>> cache.get_many(["a", "b", "c"])
{'a': 1, 'b': 2, 'c': 3}
```

Like `cache.set()`, `set_many()` takes an optional `timeout` parameter.

On supported backends (memcached), `set_many()` returns a list of keys that failed to be inserted.

cache.**delete**(key, version=None)

You can delete keys explicitly with `delete()` to clear the cache for a particular object:

```
>>> cache.delete("a")
True
```

`delete()` returns `True` if the key was successfully deleted, `False` otherwise.

cache.**delete_many**(keys, version=None)

If you want to clear a bunch of keys at once, `delete_many()` can take a list of keys to be cleared:

```
>>> cache.delete_many(["a", "b", "c"])
```

cache.**clear**()

Finally, if you want to delete all the keys in the cache, use `cache.clear()`. Be careful with this; `clear()` will remove everything from the cache, not just the keys set by your application:

```
>>> cache.clear()
```

cache.**touch**(key, timeout=DEFAULT_TIMEOUT, version=None)

`cache.touch()` sets a new expiration for a key. For example, to update a key to expire 10 seconds from now:

```
>>> cache.touch("a", 10)
True
```

Like other methods, the `timeout` argument is optional and defaults to the `TIMEOUT` option of the appropriate backend in the *CACHES* setting.

`touch()` returns `True` if the key was successfully touched, `False` otherwise.

`cache.incr`(key, delta=1, version=None)

`cache.decr`(key, delta=1, version=None)

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will be incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call. A ValueError will be raised if you attempt to increment or decrement a nonexistent cache key:

```
>>> cache.set("num", 1)
>>> cache.incr("num")
2
>>> cache.incr("num", 10)
12
>>> cache.decr("num")
11
>>> cache.decr("num", 5)
6
```

> **Note**
>
> `incr()`/`decr()` methods are not guaranteed to be atomic. On those backends that support atomic increment/decrement (most notably, the memcached backend), increment and decrement operations will be atomic. However, if the backend doesn't natively provide an increment/decrement operation, it will be implemented using a two-step retrieve/update.

`cache.close()`

You can close the connection to your cache with `close()` if implemented by the cache backend.

```
>>> cache.close()
```

> **Note**
>
> For caches that don't implement `close` methods it is a no-op.

> **Note**
>
> The async variants of base methods are prefixed with `a`, e.g. `cache.aadd()` or `cache.adelete_many()`. See Asynchronous support for more details.

Cache key prefixing

If you are sharing a cache instance between servers, or between your production and development environments, it's possible for data cached by one server to be used by another server. If the format of cached data is different between servers, this can lead to some very hard to diagnose problems.

To prevent this, Django provides the ability to prefix all cache keys used by a server. When a particular cache key is saved or retrieved, Django will automatically prefix the cache key with the value of the *KEY_PREFIX* cache setting.

By ensuring each Django instance has a different *KEY_PREFIX*, you can ensure that there will be no collisions in cache values.

Cache versioning

When you change running code that uses cached values, you may need to purge any existing cached values. The easiest way to do this is to flush the entire cache, but this can lead to the loss of cache values that are still valid and useful.

Django provides a better way to target individual cache values. Django's cache framework has a system-wide version identifier, specified using the *VERSION* cache setting. The value of this setting is automatically combined with the cache prefix and the user-provided cache key to obtain the final cache key.

By default, any key request will automatically include the site default cache key version. However, the primitive cache functions all include a `version` argument, so you can specify a particular cache key version to set or get. For example:

```
>>> # Set version 2 of a cache key
>>> cache.set("my_key", "hello world!", version=2)
>>> # Get the default version (assuming version=1)
>>> cache.get("my_key")
None
>>> # Get version 2 of the same key
>>> cache.get("my_key", version=2)
'hello world!'
```

The version of a specific key can be incremented and decremented using the `incr_version()` and `decr_version()` methods. This enables specific keys to be bumped to a new version, leaving other keys unaffected. Continuing our previous example:

```
>>> # Increment the version of 'my_key'
>>> cache.incr_version("my_key")
>>> # The default version still isn't available
>>> cache.get("my_key")
None
# Version 2 isn't available, either
>>> cache.get("my_key", version=2)
None
>>> # But version 3 *is* available
>>> cache.get("my_key", version=3)
'hello world!'
```

Cache key transformation

As described in the previous two sections, the cache key provided by a user is not used verbatim – it is combined with the cache prefix and key version to provide a final cache key. By default, the three parts are joined using colons to produce a final string:

```
def make_key(key, key_prefix, version):
    return "%s:%s:%s" % (key_prefix, version, key)
```

If you want to combine the parts in different ways, or apply other processing to the final key (e.g., taking a hash digest of the key parts), you can provide a custom key function.

The *KEY_FUNCTION* cache setting specifies a dotted-path to a function matching the prototype of `make_key()` above. If provided, this custom key function will be used instead of the default key combining function.

Cache key warnings

Memcached, the most commonly-used production cache backend, does not allow cache keys longer than 250 characters or containing whitespace or control characters, and using such keys will cause an exception. To encourage cache-portable code and minimize unpleasant surprises, the other built-in cache backends issue a warning (`django.core.cache.backends.base.CacheKeyWarning`) if a key is used that would cause an error on memcached.

If you are using a production backend that can accept a wider range of keys (a custom backend, or one of the non-memcached built-in backends), and want to use this wider range without warnings, you can silence `CacheKeyWarning` with this code in the `management` module of one of your *INSTALLED_APPS*:

```
import warnings

from django.core.cache import CacheKeyWarning

```

(continues on next page)

(continued from previous page)

```
warnings.simplefilter("ignore", CacheKeyWarning)
```

If you want to instead provide custom key validation logic for one of the built-in backends, you can subclass it, override just the `validate_key` method, and follow the instructions for using a custom cache backend. For instance, to do this for the `locmem` backend, put this code in a module:

```
from django.core.cache.backends.locmem import LocMemCache


class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        """Custom validation, raising exceptions or warnings as needed."""
        ...
```

...and use the dotted Python path to this class in the *BACKEND* portion of your *CACHES* setting.

3.11.6 Asynchronous support

Django has developing support for asynchronous cache backends, but does not yet support asynchronous caching. It will be coming in a future release.

`django.core.cache.backends.base.BaseCache` has async variants of all base methods. By convention, the asynchronous versions of all methods are prefixed with `a`. By default, the arguments for both variants are the same:

```
>>> await cache.aset("num", 1)
>>> await cache.ahas_key("num")
True
```

3.11.7 Downstream caches

So far, this document has focused on caching your own data. But another type of caching is relevant to web development, too: caching performed by "downstream" caches. These are systems that cache pages for users even before the request reaches your website.

Here are a few examples of downstream caches:

- When using HTTP, your ISP (Internet Service Provider) may cache certain pages, so if you requested a page from `http://example.com/`, your ISP would send you the page without having to access example.com directly. The maintainers of example.com have no knowledge of this caching; the ISP sits between example.com and your web browser, handling all of the caching transparently. Such caching is not possible under HTTPS as it would constitute a man-in-the-middle attack.
- Your Django website may sit behind a proxy cache, such as Squid Web Proxy Cache (http://www.

squid-cache.org/), that caches pages for performance. In this case, each request first would be handled by the proxy, and it would be passed to your application only if needed.

- Your web browser caches pages, too. If a web page sends out the appropriate headers, your browser will use the local cached copy for subsequent requests to that page, without even contacting the web page again to see whether it has changed.

Downstream caching is a nice efficiency boost, but there's a danger to it: Many web pages' contents differ based on authentication and a host of other variables, and cache systems that blindly save pages based purely on URLs could expose incorrect or sensitive data to subsequent visitors to those pages.

For example, if you operate a web email system, then the contents of the "inbox" page depend on which user is logged in. If an ISP blindly cached your site, then the first user who logged in through that ISP would have their user-specific inbox page cached for subsequent visitors to the site. That's not cool.

Fortunately, HTTP provides a solution to this problem. A number of HTTP headers exist to instruct downstream caches to differ their cache contents depending on designated variables, and to tell caching mechanisms not to cache particular pages. We'll look at some of these headers in the sections that follow.

3.11.8 Using `Vary` headers

The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key. For example, if the contents of a web page depend on a user's language preference, the page is said to "vary on language."

By default, Django's cache system creates its cache keys using the requested fully-qualified URL – e.g., `"https://www.example.com/stories/2005/?order_by=author"`. This means every request to that URL will use the same cached version, regardless of user-agent differences such as cookies or language preferences. However, if this page produces different content based on some difference in request headers – such as a cookie, or a language, or a user-agent – you'll need to use the `Vary` header to tell caching mechanisms that the page output depends on those things.

To do this in Django, use the convenient *`django.views.decorators.vary.vary_on_headers()`* view decorator, like so:

```
from django.views.decorators.vary import vary_on_headers


@vary_on_headers("User-Agent")
def my_view(request): ...
```

In this case, a caching mechanism (such as Django's own cache middleware) will cache a separate version of the page for each unique user-agent.

The advantage to using the `vary_on_headers` decorator rather than manually setting the `Vary` header (using something like `response.headers['Vary'] = 'user-agent'`) is that the decorator adds to the `Vary` header

(which may already exist), rather than setting it from scratch and potentially overriding anything that was already in there.

You can pass multiple headers to `vary_on_headers()`:

```
@vary_on_headers("User-Agent", "Cookie")
def my_view(request): ...
```

This tells downstream caches to vary on both, which means each combination of user-agent and cookie will get its own cache value. For example, a request with the user-agent `Mozilla` and the cookie value `foo=bar` will be considered different from a request with the user-agent `Mozilla` and the cookie value `foo=ham`.

Because varying on cookie is so common, there's a *`django.views.decorators.vary.vary_on_cookie()`* decorator. These two views are equivalent:

```
@vary_on_cookie
def my_view(request): ...


@vary_on_headers("Cookie")
def my_view(request): ...
```

The headers you pass to `vary_on_headers` are not case sensitive; `"User-Agent"` is the same thing as `"user-agent"`.

You can also use a helper function, *`django.utils.cache.patch_vary_headers()`*, directly. This function sets, or adds to, the `Vary header`. For example:

```
from django.shortcuts import render
from django.utils.cache import patch_vary_headers


def my_view(request):
    ...
    response = render(request, "template_name", context)
    patch_vary_headers(response, ["Cookie"])
    return response
```

`patch_vary_headers` takes an *`HttpResponse`* instance as its first argument and a list/tuple of case-insensitive header names as its second argument.

For more on Vary headers, see the official Vary spec.

3.11.9 Controlling cache: Using other headers

Other problems with caching are the privacy of data and the question of where data should be stored in a cascade of caches.

A user usually faces two kinds of caches: their own browser cache (a private cache) and their provider's cache (a public cache). A public cache is used by multiple users and controlled by someone else. This poses problems with sensitive data–you don't want, say, your bank account number stored in a public cache. So web applications need a way to tell caches which data is private and which is public.

The solution is to indicate a page's cache should be "private." To do this in Django, use the *cache_control()* view decorator. Example:

```
from django.views.decorators.cache import cache_control


@cache_control(private=True)
def my_view(request): ...
```

This decorator takes care of sending out the appropriate HTTP header behind the scenes.

Note that the cache control settings "private" and "public" are mutually exclusive. The decorator ensures that the "public" directive is removed if "private" should be set (and vice versa). An example use of the two directives would be a blog site that offers both private and public entries. Public entries may be cached on any shared cache. The following code uses *patch_cache_control()*, the manual way to modify the cache control header (it is internally called by the *cache_control()* decorator):

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie


@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous:
        response = render_only_public_entries()
        patch_cache_control(response, public=True)
    else:
        response = render_private_and_public_entries(request.user)
        patch_cache_control(response, private=True)

    return response
```

You can control downstream caches in other ways as well (see RFC 9111 for details on HTTP caching). For example, even if you don't use Django's server-side cache framework, you can still tell clients to cache a view for a certain amount of time with the max-age directive:

```
from django.views.decorators.cache import cache_control


@cache_control(max_age=3600)
def my_view(request): ...
```

(If you do use the caching middleware, it already sets the `max-age` with the value of the *CACHE_MIDDLEWARE_SECONDS* setting. In that case, the custom `max_age` from the *cache_control()* decorator will take precedence, and the header values will be merged correctly.)

Any valid `Cache-Control` response directive is valid in `cache_control()`. Here are some more examples:

- `no_transform=True`
- `must_revalidate=True`
- `stale_while_revalidate=num_seconds`
- `no_cache=True`

The full list of known directives can be found in the IANA registry (note that not all of them apply to responses).

If you want to use headers to disable caching altogether, *never_cache()* is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. Example:

```
from django.views.decorators.cache import never_cache


@never_cache
def myview(request): ...
```

3.11.10 Order of `MIDDLEWARE`

If you use caching middleware, it's important to put each half in the right place within the *MIDDLEWARE* setting. That's because the cache middleware needs to know which headers by which to vary the cache storage. Middleware always adds something to the `Vary` response header when it can.

`UpdateCacheMiddleware` runs during the response phase, where middleware is run in reverse order, so an item at the top of the list runs last during the response phase. Thus, you need to make sure that `UpdateCacheMiddleware` appears before any other middleware that might add something to the `Vary` header. The following middleware modules do so:

- `SessionMiddleware` adds `Cookie`
- `GZipMiddleware` adds `Accept-Encoding`
- `LocaleMiddleware` adds `Accept-Language`

`FetchFromCacheMiddleware`, on the other hand, runs during the request phase, where middleware is applied first-to-last, so an item at the top of the list runs first during the request phase. The `FetchFromCacheMiddleware` also needs to run after other middleware updates the `Vary` header, so `FetchFromCacheMiddleware` must be after any item that does so.

3.12 Conditional View Processing

HTTP clients can send a number of headers to tell the server about copies of a resource that they have already seen. This is commonly used when retrieving a web page (using an HTTP `GET` request) to avoid sending all the data for something the client has already retrieved. However, the same headers can be used for all HTTP methods (`POST`, `PUT`, `DELETE`, etc.).

For each page (response) that Django sends back from a view, it might provide two HTTP headers: the `ETag` header and the `Last-Modified` header. These headers are optional on HTTP responses. They can be set by your view function, or you can rely on the *`ConditionalGetMiddleware`* middleware to set the `ETag` header.

When the client next requests the same resource, it might send along a header such as either If-Modified-Since or If-Unmodified-Since, containing the date of the last modification time it was sent, or either If-Match or If-None-Match, containing the last `ETag` it was sent. If the current version of the page matches the `ETag` sent by the client, or if the resource has not been modified, a 304 status code can be sent back, instead of a full response, telling the client that nothing has changed. Depending on the header, if the page has been modified or does not match the `ETag` sent by the client, a 412 status code (Precondition Failed) may be returned.

When you need more fine-grained control you may use per-view conditional processing functions.

3.12.1 The `condition` decorator

Sometimes (in fact, quite often) you can create functions to rapidly compute the ETag value or the last-modified time for a resource, without needing to do all the computations needed to construct the full view. Django can then use these functions to provide an "early bailout" option for the view processing. Telling the client that the content has not been modified since the last request, perhaps.

These two functions are passed as parameters to the `django.views.decorators.http.condition` decorator. This decorator uses the two functions (you only need to supply one, if you can't compute both quantities easily and quickly) to work out if the headers in the HTTP request match those on the resource. If they don't match, a new copy of the resource must be computed and your normal view is called.

The `condition` decorator's signature looks like this:

```
condition(etag_func=None, last_modified_func=None)
```

The two functions, to compute the ETag and the last modified time, will be passed the incoming `request` object and the same parameters, in the same order, as the view function they are helping to wrap. The function passed `last_modified_func` should return a standard datetime value specifying the last time the

resource was modified, or `None` if the resource doesn't exist. The function passed to the `etag` decorator should return a string representing the ETag for the resource, or `None` if it doesn't exist.

The decorator sets the `ETag` and `Last-Modified` headers on the response if they are not already set by the view and if the request's method is safe (`GET` or `HEAD`).

Using this feature usefully is probably best explained with an example. Suppose you have this pair of models, representing a small blog system:

```
import datetime
from django.db import models


class Blog(models.Model): ...


class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    published = models.DateTimeField(default=datetime.datetime.now)
    ...
```

If the front page, displaying the latest blog entries, only changes when you add a new blog entry, you can compute the last modified time very quickly. You need the latest `published` date for every entry associated with that blog. One way to do this would be:

```
def latest_entry(request, blog_id):
    return Entry.objects.filter(blog=blog_id).latest("published").published
```

You can then use this function to provide early detection of an unchanged page for your front page view:

```
from django.views.decorators.http import condition


@condition(last_modified_func=latest_entry)
def front_page(request, blog_id): ...
```

Be careful with the order of decorators

When `condition()` returns a conditional response, any decorators below it will be skipped and won't apply to the response. Therefore, any decorators that need to apply to both the regular view response and a conditional response must be above `condition()`. In particular, *`vary_on_cookie()`*, *`vary_on_headers()`*, and *`cache_control()`* should come first because RFC 9110 requires that the headers they set be present on 304 responses.

3.12.2 Shortcuts for only computing one value

As a general rule, if you can provide functions to compute both the ETag and the last modified time, you should do so. You don't know which headers any given HTTP client will send you, so be prepared to handle both. However, sometimes only one value is easy to compute and Django provides decorators that handle only ETag or only last-modified computations.

The `django.views.decorators.http.etag` and `django.views.decorators.http.last_modified` decorators are passed the same type of functions as the `condition` decorator. Their signatures are:

```
etag(etag_func)
last_modified(last_modified_func)
```

We could write the earlier example, which only uses a last-modified function, using one of these decorators:

```
@last_modified(latest_entry)
def front_page(request, blog_id): ...
```

...or:

```
def front_page(request, blog_id): ...


front_page = last_modified(latest_entry)(front_page)
```

Use `condition` when testing both conditions

It might look nicer to some people to try and chain the `etag` and `last_modified` decorators if you want to test both preconditions. However, this would lead to incorrect behavior.

```
# Bad code. Don't do this!
@etag(etag_func)
@last_modified(last_modified_func)
def my_view(request): ...


# End of bad code.
```

The first decorator doesn't know anything about the second and might answer that the response is not modified even if the second decorators would determine otherwise. The `condition` decorator uses both callback functions simultaneously to work out the right action to take.

3.12.3 Using the decorators with other HTTP methods

The `condition` decorator is useful for more than only `GET` and `HEAD` requests (`HEAD` requests are the same as `GET` in this situation). It can also be used to provide checking for `POST`, `PUT` and `DELETE` requests. In these situations, the idea isn't to return a "not modified" response, but to tell the client that the resource they are trying to change has been altered in the meantime.

For example, consider the following exchange between the client and server:

1. Client requests `/foo/`.
2. Server responds with some content with an ETag of `"abcd1234"`.
3. Client sends an HTTP `PUT` request to `/foo/` to update the resource. It also sends an `If-Match: "abcd1234"` header to specify the version it is trying to update.
4. Server checks to see if the resource has changed, by computing the ETag the same way it does for a `GET` request (using the same function). If the resource has changed, it will return a 412 status code, meaning "precondition failed".
5. Client sends a `GET` request to `/foo/`, after receiving a 412 response, to retrieve an updated version of the content before updating it.

The important thing this example shows is that the same functions can be used to compute the ETag and last modification values in all situations. In fact, you should use the same functions, so that the same values are returned every time.

> **Validator headers with non-safe request methods**
>
> The `condition` decorator only sets validator headers (`ETag` and `Last-Modified`) for safe HTTP methods, i.e. `GET` and `HEAD`. If you wish to return them in other cases, set them in your view. See RFC 9110 Section 9.3.4 to learn about the distinction between setting a validator header in response to requests made with `PUT` versus `POST`.

3.12.4 Comparison with middleware conditional processing

Django provides conditional `GET` handling via *`django.middleware.http.ConditionalGetMiddleware`*. While being suitable for many situations, the middleware has limitations for advanced usage:

- It's applied globally to all views in your project.
- It doesn't save you from generating the response, which may be expensive.
- It's only appropriate for HTTP `GET` requests.

You should choose the most appropriate tool for your particular problem here. If you have a way to compute ETags and modification times quickly and if some view takes a while to generate the content, you should consider using the `condition` decorator described in this document. If everything already runs fairly quickly,

stick to using the middleware and the amount of network traffic sent back to the clients will still be reduced if the view hasn't changed.

3.13 Composite primary keys

In Django, each model has a primary key. By default, this primary key consists of a single field.

In most cases, a single primary key should suffice. In database design, however, defining a primary key consisting of multiple fields is sometimes necessary.

To use a composite primary key, when defining a model set the `pk` attribute to be a *CompositePrimaryKey*:

```
class Product(models.Model):
    name = models.CharField(max_length=100)


class Order(models.Model):
    reference = models.CharField(max_length=20, primary_key=True)


class OrderLineItem(models.Model):
    pk = models.CompositePrimaryKey("product_id", "order_id")
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    order = models.ForeignKey(Order, on_delete=models.CASCADE)
    quantity = models.IntegerField()
```

This will instruct Django to create a composite primary key (`PRIMARY KEY (product_id, order_id)`) when creating the table.

A composite primary key is represented by a `tuple`:

```
>>> product = Product.objects.create(name="apple")
>>> order = Order.objects.create(reference="A755H")
>>> item = OrderLineItem.objects.create(product=product, order=order, quantity=1)
>>> item.pk
(1, "A755H")
```

You can assign a `tuple` to the *pk* attribute. This sets the associated field values:

```
>>> item = OrderLineItem(pk=(2, "B142C"))
>>> item.pk
(2, "B142C")
>>> item.product_id
2
```

(continues on next page)

(continued from previous page)

```
>>> item.order_id
"B142C"
```

A composite primary key can also be filtered by a `tuple`:

```
>>> OrderLineItem.objects.filter(pk=(1, "A755H")).count()
1
```

We're still working on composite primary key support for relational fields, including *`GenericForeignKey`* fields, and the Django admin. Models with composite primary keys cannot be registered in the Django admin at this time. You can expect to see this in future releases.

3.13.1 Migrating to a composite primary key

Django doesn't support migrating to, or from, a composite primary key after the table is created. It also doesn't support adding or removing fields from the composite primary key.

If you would like to migrate an existing table from a single primary key to a composite primary key, follow your database backend's instructions to do so.

Once the composite primary key is in place, add the `CompositePrimaryKey` field to your model. This allows Django to recognize and handle the composite primary key appropriately.

While migration operations (e.g. `AddField`, `AlterField`) on primary key fields are not supported, `makemigrations` will still detect changes.

In order to avoid errors, it's recommended to apply such migrations with `--fake`.

Alternatively, *`SeparateDatabaseAndState`* may be used to execute the backend-specific migrations and Django-generated migrations in a single operation.

3.13.2 Composite primary keys and relations

Relationship fields, including generic relations do not support composite primary keys.

For example, given the `OrderLineItem` model, the following is not supported:

```
class Foo(models.Model):
    item = models.ForeignKey(OrderLineItem, on_delete=models.CASCADE)
```

Because `ForeignKey` currently cannot reference models with composite primary keys.

To work around this limitation, `ForeignObject` can be used as an alternative:

```
class Foo(models.Model):
    item_order_id = models.CharField(max_length=20)
```

(continues on next page)

(continued from previous page)

```
    item_product_id = models.IntegerField()
    item = models.ForeignObject(
        OrderLineItem,
        on_delete=models.CASCADE,
        from_fields=("item_order_id", "item_product_id"),
        to_fields=("order_id", "product_id"),
    )
```

`ForeignObject` is much like `ForeignKey`, except that it doesn't create any columns (e.g. `item_id`), foreign key constraints or indexes in the database, and the `on_delete` argument is ignored.

> **Warning**
>
> `ForeignObject` is an internal API. This means it is not covered by our deprecation policy.

3.13.3 Composite primary keys and database functions

Many database functions only accept a single expression.

```
MAX("order_id")  -- OK
MAX("product_id", "order_id")  -- ERROR
```

In these cases, providing a composite primary key reference raises a `ValueError`, since it is composed of multiple column expressions. An exception is made for `Count`.

```
Max("order_id")  # OK
Max("pk")  # ValueError
Count("pk")  # OK
```

3.13.4 Composite primary keys in forms

As a composite primary key is a virtual field, a field which doesn't represent a single database column, this field is excluded from ModelForms.

For example, take the following form:

```
class OrderLineItemForm(forms.ModelForm):
    class Meta:
        model = OrderLineItem
        fields = "__all__"
```

This form does not have a form field `pk` for the composite primary key:

```
>>> OrderLineItemForm()
<OrderLineItemForm bound=False, valid=Unknown, fields=(product;order;quantity)>
```

Setting the primary composite field `pk` as a form field raises an unknown field *`FieldError`*.

> **Primary key fields are read only**
>
> If you change the value of a primary key on an existing object and then save it, a new object will be created alongside the old one (see *`Field.primary_key`*).
>
> This is also true of composite primary keys. Hence, you may want to set *`Field.editable`* to `False` on all primary key fields to exclude them from ModelForms.

3.13.5 Composite primary keys in model validation

Since `pk` is only a virtual field, including `pk` as a field name in the `exclude` argument of *`Model.clean_fields()`* has no effect. To exclude the composite primary key fields from model validation, specify each field individually. *`Model.validate_unique()`* can still be called with `exclude={"pk"}` to skip uniqueness checks.

3.13.6 Building composite primary key ready applications

Prior to the introduction of composite primary keys, the single field composing the primary key of a model could be retrieved by introspecting the *`primary_key`* attribute of its fields:

```
>>> pk_field = None
>>> for field in Product._meta.get_fields():
...     if field.primary_key:
...         pk_field = field
...         break
...
>>> pk_field
<django.db.models.fields.AutoField: id>
```

Now that a primary key can be composed of multiple fields the *`primary_key`* attribute can no longer be relied upon to identify members of the primary key as it will be set to `False` to maintain the invariant that at most one field per model will have this attribute set to `True`:

```
>>> pk_fields = []
>>> for field in OrderLineItem._meta.get_fields():
...     if field.primary_key:
...         pk_fields.append(field)
```

(continues on next page)

(continued from previous page)

```
...
>>> pk_fields
[]
```

In order to build application code that properly handles composite primary keys the *_meta.pk_fields* attribute should be used instead:

```
>>> Product._meta.pk_fields
[<django.db.models.fields.AutoField: id>]
>>> OrderLineItem._meta.pk_fields
[
    <django.db.models.fields.ForeignKey: product>,
    <django.db.models.fields.ForeignKey: order>
]
```

3.14 Cryptographic signing

The golden rule of web application security is to never trust data from untrusted sources. Sometimes it can be useful to pass data through an untrusted medium. Cryptographically signed values can be passed through an untrusted channel safe in the knowledge that any tampering will be detected.

Django provides both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in web applications.

You may also find signing useful for the following:

- Generating "recover my account" URLs for sending to users who have lost their password.
- Ensuring data stored in hidden form fields has not been tampered with.
- Generating one-time secret URLs for allowing temporary access to a protected resource, for example a downloadable file that a user has paid for.

3.14.1 Protecting SECRET_KEY and SECRET_KEY_FALLBACKS

When you create a new Django project using *startproject*, the `settings.py` file is generated automatically and gets a random *SECRET_KEY* value. This value is the key to securing signed data – it is vital you keep this secure, or attackers could use it to generate their own signed values.

SECRET_KEY_FALLBACKS can be used to rotate secret keys. The values will not be used to sign data, but if specified, they will be used to validate signed data and must be kept secure.

3.14.2 Using the low-level API

Django's signing methods live in the `django.core.signing` module. To sign a value, first instantiate a `Signer` instance:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign("My string")
>>> value
'My string:v9G-nxfz3iQGTXrePqYPlGvH79WTcIgj1QIQSUODTW0'
```

The signature is appended to the end of the string, following the colon. You can retrieve the original value using the `unsign` method:

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

If you pass a non-string value to `sign`, the value will be forced to string before being signed, and the `unsign` result will give you that string value:

```
>>> signed = signer.sign(2.5)
>>> original = signer.unsign(signed)
>>> original
'2.5'
```

If you wish to protect a list, tuple, or dictionary you can do so using the `sign_object()` and `unsign_object()` methods:

```
>>> signed_obj = signer.sign_object({"message": "Hello!"})
>>> signed_obj
'eyJtZXNzYWdlIjoiSGVsbG8hIn0:bzb48DBkB-bwLaCnUVB75r5VAPUEpzWJPrTb80JMIXM'
>>> obj = signer.unsign_object(signed_obj)
>>> obj
{'message': 'Hello!'}
```

See Protecting complex data structures for more details.

If the signature or value have been altered in any way, a `django.core.signing.BadSignature` exception will be raised:

```
>>> from django.core import signing
>>> value += "m"
>>> try:
```

(continues on next page)

(continued from previous page)

```
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
...
```

By default, the `Signer` class uses the *SECRET_KEY* setting to generate signatures. You can use a different secret by passing it to the `Signer` constructor:

```
>>> signer = Signer(key="my-other-secret")
>>> value = signer.sign("My string")
>>> value
'My string:o3DrrsT6JRB73t-HDymfDNbTSxfM1om2d8TiU1b1hWY'
```

class **Signer**(*, key=None, sep=':', salt=None, algorithm=None, fallback_keys=None)

Returns a signer which uses `key` to generate signatures and `sep` to separate values. `sep` cannot be in the URL safe base64 alphabet. This alphabet contains alphanumeric characters, hyphens, and underscores. `algorithm` must be an algorithm supported by `hashlib`, it defaults to `'sha256'`. `fallback_keys` is a list of additional values used to validate signed data, defaults to *SECRET_KEY_FALLBACKS*.

Using the `salt` argument

If you do not wish for every occurrence of a particular string to have the same signature hash, you can use the optional `salt` argument to the `Signer` class. Using a salt will seed the signing hash function with both the salt and your *SECRET_KEY*:

```
>>> signer = Signer()
>>> signer.sign("My string")
'My string:v9G-nxfz3iQGTXrePqYPlGvH79WTcIgj1QIQSUODTW0'
>>> signer.sign_object({"message": "Hello!"})
'eyJtZXNzYWdlIjoiSGVsbG8hIn0:bzb48DBkB-bwLaCnUVB75r5VAPUEpzWJPrTb80JMIXM'
>>> signer = Signer(salt="extra")
>>> signer.sign("My string")
'My string:YMD-FR6rof3heDkFRffdmG4pXbAZSOtb-aQxg3vmmfc'
>>> signer.unsign("My string:YMD-FR6rof3heDkFRffdmG4pXbAZSOtb-aQxg3vmmfc")
'My string'
>>> signer.sign_object({"message": "Hello!"})
'eyJtZXNzYWdlIjoiSGVsbG8hIn0:-UWSLCE-oUAHzhkHviYz3SOZYBjFKllEOyVZNuUtM-I'
>>> signer.unsign_object(
...     "eyJtZXNzYWdlIjoiSGVsbG8hIn0:-UWSLCE-oUAHzhkHviYz3SOZYBjFKllEOyVZNuUtM-I"
... )
{'message': 'Hello!'}
```

Using salt in this way puts the different signatures into different namespaces. A signature that comes from one namespace (a particular salt value) cannot be used to validate the same plaintext string in a different namespace that is using a different salt setting. The result is to prevent an attacker from using a signed string generated in one place in the code as input to another piece of code that is generating (and verifying) signatures using a different salt.

Unlike your *SECRET_KEY*, your salt argument does not need to stay secret.

Verifying timestamped values

`TimestampSigner` is a subclass of *Signer* that appends a signed timestamp to the value. This allows you to confirm that a signed value was created within a specified period of time:

```
>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign("hello")
>>> value
'hello:1stLqR:_rvr4oXCgT4HyfwjXaU39QvTnuNuUthFRCzNOy4Hqt0'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'
```

class `TimestampSigner`(*, key=None, sep=':', salt=None, algorithm='sha256')

sign(value)

Sign `value` and append current timestamp to it.

unsign(value, max_age=None)

Checks if `value` was signed less than `max_age` seconds ago, otherwise raises `SignatureExpired`. The `max_age` parameter can accept an integer or a `datetime.timedelta` object.

sign_object(obj, serializer=JSONSerializer, compress=False)

Encode, optionally compress, append current timestamp, and sign complex data structure (e.g. list, tuple, or dictionary).

unsign_object(signed_obj, serializer=JSONSerializer, max_age=None)

Checks if `signed_obj` was signed less than `max_age` seconds ago, otherwise raises `SignatureExpired`. The `max_age` parameter can accept an integer or a `datetime.timedelta` object.

Protecting complex data structures

If you wish to protect a list, tuple or dictionary you can do so using the `Signer.sign_object()` and `unsign_object()` methods, or signing module's `dumps()` or `loads()` functions (which are shortcuts for `TimestampSigner(salt='django.core.signing').sign_object()/unsign_object()`). These use JSON serialization under the hood. JSON ensures that even if your *SECRET_KEY* is stolen an attacker will not be able to execute arbitrary commands by exploiting the pickle format:

```
>>> from django.core import signing
>>> signer = signing.TimestampSigner()
>>> value = signer.sign_object({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1stLrZ:_QiOBHafwucBF9FyAr54qEs84ZO1UdsO1XiTJCvvdno'
>>> signer.unsign_object(value)
{'foo': 'bar'}
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1stLsC:JItq2ZVjmAK6ivrWI-v1Gk1QVf2hOF52oaEqhZHca7I'
>>> signing.loads(value)
{'foo': 'bar'}
```

Because of the nature of JSON (there is no native distinction between lists and tuples) if you pass in a tuple, you will get a list from `signing.loads(object)`:

```
>>> from django.core import signing
>>> value = signing.dumps(("a", "b", "c"))
>>> signing.loads(value)
['a', 'b', 'c']
```

dumps(obj, key=None, salt='django.core.signing', serializer=JSONSerializer, compress=False)

Returns URL-safe, signed base64 compressed JSON string. Serialized object is signed using *TimestampSigner*.

loads(string, key=None, salt='django.core.signing', serializer=JSONSerializer, max_age=None, fallback_keys=None)

Reverse of `dumps()`, raises `BadSignature` if signature fails. Checks `max_age` (in seconds) if given.

3.15 Sending email

Although Python provides a mail sending interface via the `smtplib` module, Django provides a couple of light wrappers over it. These wrappers are provided to make sending email extra quick, to help test email sending during development, and to provide support for platforms that can't use SMTP.

The code lives in the `django.core.mail` module.

3.15.1 Quick examples

Use *send_mail()* for straightforward email sending. For example, to send a plain text message:

```
from django.core.mail import send_mail

send_mail(
    "Subject here",
    "Here is the message.",
    "from@example.com",
    ["to@example.com"],
    fail_silently=False,
)
```

When additional email sending functionality is needed, use *EmailMessage* or *EmailMultiAlternatives*. For example, to send a multipart email that includes both HTML and plain text versions with a specific template and custom headers, you can use the following approach:

```
from django.core.mail import EmailMultiAlternatives
from django.template.loader import render_to_string

# First, render the plain text content.
text_content = render_to_string(
    "templates/emails/my_email.txt",
    context={"my_variable": 42},
)

# Secondly, render the HTML content.
html_content = render_to_string(
    "templates/emails/my_email.html",
    context={"my_variable": 42},
)

# Then, create a multipart email instance.
msg = EmailMultiAlternatives(
    "Subject here",
    text_content,
    "from@example.com",
    ["to@example.com"],
    headers={"List-Unsubscribe": "<mailto:unsub@example.com>"},
)
```

(continues on next page)

(continued from previous page)

```
# Lastly, attach the HTML content to the email instance and send.
msg.attach_alternative(html_content, "text/html")
msg.send()
```

Mail is sent using the SMTP host and port specified in the *EMAIL_HOST* and *EMAIL_PORT* settings. The *EMAIL_HOST_USER* and *EMAIL_HOST_PASSWORD* settings, if set, are used to authenticate to the SMTP server, and the *EMAIL_USE_TLS* and *EMAIL_USE_SSL* settings control whether a secure connection is used.

> **Note**
>
> The character set of email sent with `django.core.mail` will be set to the value of your *DEFAULT_CHARSET* setting.

3.15.2 `send_mail()`

send_mail(subject, message, from_email, recipient_list, fail_silently=False, auth_user=None, auth_password=None, connection=None, html_message=None)

In most cases, you can send email using `django.core.mail.send_mail()`.

The `subject`, `message`, `from_email` and `recipient_list` parameters are required.

- `subject`: A string.
- `message`: A string.
- `from_email`: A string. If `None`, Django will use the value of the *DEFAULT_FROM_EMAIL* setting.
- `recipient_list`: A list of strings, each an email address. Each member of `recipient_list` will see the other recipients in the "To:" field of the email message.
- `fail_silently`: A boolean. When it's `False`, `send_mail()` will raise an `smtplib.SMTPException` if an error occurs. See the `smtplib` docs for a list of possible exceptions, all of which are subclasses of `SMTPException`.
- `auth_user`: The optional username to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the *EMAIL_HOST_USER* setting.
- `auth_password`: The optional password to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the *EMAIL_HOST_PASSWORD* setting.
- `connection`: The optional email backend to use to send the mail. If unspecified, an instance of the default backend will be used. See the documentation on Email backends for more details.
- `html_message`: If `html_message` is provided, the resulting email will be a *multipart/alternative* email with `message` as the *text/plain* content type and `html_message` as the *text/html* content type.

The return value will be the number of successfully delivered messages (which can be `0` or `1` since it can only send one message).

3.15.3 `send_mass_mail()`

`send_mass_mail`(datatuple, fail_silently=False, auth_user=None, auth_password=None, connection=None)

`django.core.mail.send_mass_mail()` is intended to handle mass emailing.

`datatuple` is a tuple in which each element is in this format:

```
(subject, message, from_email, recipient_list)
```

`fail_silently`, `auth_user` and `auth_password` have the same functions as in *`send_mail()`*.

Each separate element of `datatuple` results in a separate email message. As in *`send_mail()`*, recipients in the same `recipient_list` will all see the other addresses in the email messages' "To:" field.

For example, the following code would send two different messages to two different sets of recipients; however, only one connection to the mail server would be opened:

```
message1 = (
    "Subject here",
    "Here is the message",
    "from@example.com",
    ["first@example.com", "other@example.com"],
)
message2 = (
    "Another Subject",
    "Here is another message",
    "from@example.com",
    ["second@test.com"],
)
send_mass_mail((message1, message2), fail_silently=False)
```

The return value will be the number of successfully delivered messages.

`send_mass_mail()` vs. `send_mail()`

The main difference between *`send_mass_mail()`* and *`send_mail()`* is that *`send_mail()`* opens a connection to the mail server each time it's executed, while *`send_mass_mail()`* uses a single connection for all of its messages. This makes *`send_mass_mail()`* slightly more efficient.

3.15.4 mail_admins()

mail_admins(subject, message, fail_silently=False, connection=None, html_message=None)

`django.core.mail.mail_admins()` is a shortcut for sending an email to the site admins, as defined in the *ADMINS* setting.

`mail_admins()` prefixes the subject with the value of the *EMAIL_SUBJECT_PREFIX* setting, which is `"[Django] "` by default.

The "From:" header of the email will be the value of the *SERVER_EMAIL* setting.

This method exists for convenience and readability.

If `html_message` is provided, the resulting email will be a *multipart/alternative* email with `message` as the *text/plain* content type and `html_message` as the *text/html* content type.

3.15.5 mail_managers()

mail_managers(subject, message, fail_silently=False, connection=None, html_message=None)

`django.core.mail.mail_managers()` is just like `mail_admins()`, except it sends an email to the site managers, as defined in the *MANAGERS* setting.

3.15.6 Examples

This sends a single email to john@example.com and jane@example.com, with them both appearing in the "To:":

```
send_mail(
    "Subject",
    "Message.",
    "from@example.com",
    ["john@example.com", "jane@example.com"],
)
```

This sends a message to john@example.com and jane@example.com, with them both receiving a separate email:

```
datatuple = (
    ("Subject", "Message.", "from@example.com", ["john@example.com"]),
    ("Subject", "Message.", "from@example.com", ["jane@example.com"]),
)
send_mass_mail(datatuple)
```

3.15.7 Preventing header injection

Header injection is a security exploit in which an attacker inserts extra email headers to control the "To:" and "From:" in email messages that your scripts generate.

The Django email functions outlined above all protect against header injection by forbidding newlines in header values. If any `subject`, `from_email` or `recipient_list` contains a newline (in either Unix, Windows or Mac style), the email function (e.g. *`send_mail()`*) will raise `django.core.mail.BadHeaderError` (a subclass of `ValueError`) and, hence, will not send the email. It's your responsibility to validate all data before passing it to the email functions.

If a `message` contains headers at the start of the string, the headers will be printed as the first bit of the email message.

Here's an example view that takes a `subject`, `message` and `from_email` from the request's POST data, sends that to admin@example.com and redirects to "/contact/thanks/" when it's done:

```
from django.core.mail import BadHeaderError, send_mail
from django.http import HttpResponse, HttpResponseRedirect


def send_email(request):
    subject = request.POST.get("subject", "")
    message = request.POST.get("message", "")
    from_email = request.POST.get("from_email", "")
    if subject and message and from_email:
        try:
            send_mail(subject, message, from_email, ["admin@example.com"])
        except BadHeaderError:
            return HttpResponse("Invalid header found.")
        return HttpResponseRedirect("/contact/thanks/")
    else:
        # In reality we'd use a form class
        # to get proper validation errors.
        return HttpResponse("Make sure all fields are entered and valid.")
```

3.15.8 The `EmailMessage` class

Django's *`send_mail()`* and *`send_mass_mail()`* functions are actually thin wrappers that make use of the *`EmailMessage`* class.

Not all features of the *`EmailMessage`* class are available through the *`send_mail()`* and related wrapper functions. If you wish to use advanced features, such as BCC'ed recipients, file attachments, or multi-part email, you'll need to create *`EmailMessage`* instances directly.

> **Note**
>
> This is a design feature. *send_mail()* and related functions were originally the only interface Django provided. However, the list of parameters they accepted was slowly growing over time. It made sense to move to a more object-oriented design for email messages and retain the original functions only for backwards compatibility.

EmailMessage is responsible for creating the email message itself. The email backend is then responsible for sending the email.

For convenience, *EmailMessage* provides a `send()` method for sending a single email. If you need to send multiple messages, the email backend API provides an alternative.

`EmailMessage` Objects

`class EmailMessage`

The *EmailMessage* class is initialized with the following parameters (in the given order, if positional arguments are used). All parameters are optional and can be set at any time prior to calling the `send()` method.

- `subject`: The subject line of the email.
- `body`: The body text. This should be a plain text message.
- `from_email`: The sender's address. Both `fred@example.com` and `"Fred" <fred@example.com>` forms are legal. If omitted, the *DEFAULT_FROM_EMAIL* setting is used.
- `to`: A list or tuple of recipient addresses.
- `bcc`: A list or tuple of addresses used in the "Bcc" header when sending the email.
- `connection`: An email backend instance. Use this parameter if you are sending the `EmailMessage` via `send()` and you want to use the same connection for multiple messages. If omitted, a new connection is created when `send()` is called. This parameter is ignored when using send_messages().
- `attachments`: A list of attachments to put on the message. These can be instances of `MIMEBase` or *EmailAttachment*, or a tuple with attributes `(filename, content, mimetype)`.

  Support for *EmailAttachment* items of `attachments` were added.
- `headers`: A dictionary of extra headers to put on the message. The keys are the header name, values are the header values. It's up to the caller to ensure header names and values are in the correct format for an email message. The corresponding attribute is `extra_headers`.
- `cc`: A list or tuple of recipient addresses used in the "Cc" header when sending the email.
- `reply_to`: A list or tuple of recipient addresses used in the "Reply-To" header when sending the email.

For example:

```
from django.core.mail import EmailMessage

email = EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to1@example.com", "to2@example.com"],
    ["bcc@example.com"],
    reply_to=["another@example.com"],
    headers={"Message-ID": "foo"},
)
```

The class has the following methods:

- `send(fail_silently=False)` sends the message. If a connection was specified when the email was constructed, that connection will be used. Otherwise, an instance of the default backend will be instantiated and used. If the keyword argument `fail_silently` is `True`, exceptions raised while sending the message will be quashed. An empty list of recipients will not raise an exception. It will return `1` if the message was sent successfully, otherwise `0`.
- `message()` constructs a `django.core.mail.SafeMIMEText` object (a subclass of Python's `MIMEText` class) or a `django.core.mail.SafeMIMEMultipart` object holding the message to be sent. If you ever need to extend the *`EmailMessage`* class, you'll probably want to override this method to put the content you want into the MIME object.
- `recipients()` returns a list of all the recipients of the message, whether they're recorded in the `to`, `cc` or `bcc` attributes. This is another method you might need to override when subclassing, because the SMTP server needs to be told the full list of recipients when the message is sent. If you add another way to specify recipients in your class, they need to be returned from this method as well.
- `attach()` creates a new file attachment and adds it to the message. There are two ways to call `attach()`:
  - You can pass it a single argument that is a `MIMEBase` instance. This will be inserted directly into the resulting message.
  - Alternatively, you can pass `attach()` three arguments: `filename`, `content` and `mimetype`. `filename` is the name of the file attachment as it will appear in the email, `content` is the data that will be contained inside the attachment and `mimetype` is the optional MIME type for the attachment. If you omit `mimetype`, the MIME content type will be guessed from the filename of the attachment.

    For example:

    ```
    message.attach("design.png", img_data, "image/png")
    ```

If you specify a `mimetype` of *message/rfc822*, it will also accept *django.core.mail.EmailMessage* and `email.message.Message`.

For a `mimetype` starting with *text/*, content is expected to be a string. Binary data will be decoded using UTF-8, and if that fails, the MIME type will be changed to *application/octet-stream* and the data will be attached unchanged.

In addition, *message/rfc822* attachments will no longer be base64-encoded in violation of RFC 2046 Section 5.2.1, which can cause issues with displaying the attachments in Evolution and Thunderbird.

- `attach_file()` creates a new attachment using a file from your filesystem. Call it with the path of the file to attach and, optionally, the MIME type to use for the attachment. If the MIME type is omitted, it will be guessed from the filename. You can use it like this:

```
message.attach_file("/images/weather_map.png")
```

For MIME types starting with *text/*, binary data is handled as in `attach()`.

class EmailAttachment

A named tuple to store attachments to an email.

The named tuple has the following indexes:

- `filename`
- `content`
- `mimetype`

Sending alternative content types

Sending multiple content versions

It can be useful to include multiple versions of the content in an email; the classic example is to send both text and HTML versions of a message. With Django's email library, you can do this using the *EmailMultiAlternatives* class.

class EmailMultiAlternatives

A subclass of *EmailMessage* that allows additional versions of the message body in the email via the *attach_alternative()* method. This directly inherits all methods (including the class initialization) from *EmailMessage*.

alternatives

A list of *EmailAlternative* named tuples. This is particularly useful in tests:

```
self.assertEqual(len(msg.alternatives), 1)
self.assertEqual(msg.alternatives[0].content, html_content)
self.assertEqual(msg.alternatives[0].mimetype, "text/html")
```

Alternatives should only be added using the *attach_alternative()* method, or passed to the constructor.

In older versions, `alternatives` was a list of regular tuples, as opposed to *EmailAlternative* named tuples.

`attach_alternative`(content, mimetype)

Attach an alternative representation of the message body in the email.

For example, to send a text and HTML combination, you could write:

```
from django.core.mail import EmailMultiAlternatives

subject = "hello"
from_email = "from@example.com"
to = "to@example.com"
text_content = "This is an important message."
html_content = "<p>This is an <strong>important</strong> message.</p>"
msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
msg.attach_alternative(html_content, "text/html")
msg.send()
```

`body_contains`(text)

Returns a boolean indicating whether the provided `text` is contained in the email `body` and in all attached MIME type `text/*` alternatives.

This can be useful when testing emails. For example:

```
def test_contains_email_content(self):
    subject = "Hello World"
    from_email = "from@example.com"
    to = "to@example.com"
    msg = EmailMultiAlternatives(subject, "I am content.", from_email, [to])
    msg.attach_alternative("<p>I am content.</p>", "text/html")

    self.assertIs(msg.body_contains("I am content"), True)
    self.assertIs(msg.body_contains("<p>I am content.</p>"), False)
```

`class EmailAlternative`

A named tuple to store alternative versions of email content.

The named tuple has the following indexes:

- `content`
- `mimetype`

Updating the default content type

By default, the MIME type of the `body` parameter in an *EmailMessage* is `"text/plain"`. It is good practice to leave this alone, because it guarantees that any recipient will be able to read the email, regardless of their mail client. However, if you are confident that your recipients can handle an alternative content type, you can use the `content_subtype` attribute on the *EmailMessage* class to change the main content type. The major type will always be `"text"`, but you can change the subtype. For example:

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html"  # Main content is now text/html
msg.send()
```

3.15.9 Email backends

The actual sending of an email is handled by the email backend.

The email backend class has the following methods:

- `open()` instantiates a long-lived email-sending connection.
- `close()` closes the current email-sending connection.
- `send_messages(email_messages)` sends a list of *EmailMessage* objects. If the connection is not open, this call will implicitly open the connection, and close the connection afterward. If the connection is already open, it will be left open after mail has been sent.

It can also be used as a context manager, which will automatically call `open()` and `close()` as needed:

```
from django.core import mail

with mail.get_connection() as connection:
    mail.EmailMessage(
        subject1,
        body1,
        from1,
        [to1],
        connection=connection,
    ).send()
    mail.EmailMessage(
        subject2,
        body2,
        from2,
        [to2],
        connection=connection,
    ).send()
```

Obtaining an instance of an email backend

The *get_connection()* function in `django.core.mail` returns an instance of the email backend that you can use.

get_connection(backend=None, fail_silently=False, *args, **kwargs)

By default, a call to `get_connection()` will return an instance of the email backend specified in *EMAIL_BACKEND*. If you specify the `backend` argument, an instance of that backend will be instantiated.

The `fail_silently` argument controls how the backend should handle errors. If `fail_silently` is True, exceptions during the email sending process will be silently ignored.

All other arguments are passed directly to the constructor of the email backend.

Django ships with several email sending backends. With the exception of the SMTP backend (which is the default), these backends are only useful during testing and development. If you have special email sending requirements, you can write your own email backend.

SMTP backend

class backends.smtp.**EmailBackend**(host=None, port=None, username=None, password=None, use_tls=None, fail_silently=False, use_ssl=None, timeout=None, ssl_keyfile=None, ssl_certfile=None, **kwargs)

This is the default backend. Email will be sent through a SMTP server.

The value for each argument is retrieved from the matching setting if the argument is `None`:

- `host`: *EMAIL_HOST*
- `port`: *EMAIL_PORT*
- `username`: *EMAIL_HOST_USER*
- `password`: *EMAIL_HOST_PASSWORD*
- `use_tls`: *EMAIL_USE_TLS*
- `use_ssl`: *EMAIL_USE_SSL*
- `timeout`: *EMAIL_TIMEOUT*
- `ssl_keyfile`: *EMAIL_SSL_KEYFILE*
- `ssl_certfile`: *EMAIL_SSL_CERTFILE*

The SMTP backend is the default configuration inherited by Django. If you want to specify it explicitly, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.smtp.EmailBackend"
```

If unspecified, the default `timeout` will be the one provided by `socket.getdefaulttimeout()`, which defaults to `None` (no timeout).

Console backend

Instead of sending out real emails the console backend just writes the emails that would be sent to the standard output. By default, the console backend writes to `stdout`. You can use a different stream-like object by providing the `stream` keyword argument when constructing the connection.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

File backend

The file backend writes emails to a file. A new file is created for each new session that is opened on this backend. The directory to which the files are written is either taken from the *EMAIL_FILE_PATH* setting or from the `file_path` keyword when creating a connection with *get_connection()*.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
EMAIL_FILE_PATH = "/tmp/app-messages"  # change this to a proper location
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

In-memory backend

The `'locmem'` backend stores messages in a special attribute of the `django.core.mail` module. The `outbox` attribute is created when the first message is sent. It's a list with an *EmailMessage* instance for each message that would be sent.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.locmem.EmailBackend"
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development and testing.

Django's test runner automatically uses this backend for testing.

Dummy backend

As the name suggests the dummy backend does nothing with your messages. To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.dummy.EmailBackend"
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

Defining a custom email backend

If you need to change how emails are sent you can write your own email backend. The *EMAIL_BACKEND* setting in your settings file is then the Python import path for your backend class.

Custom email backends should subclass `BaseEmailBackend` that is located in the `django.core.mail.backends.base` module. A custom email backend must implement the `send_messages(email_messages)` method. This method receives a list of *EmailMessage* instances and returns the number of successfully delivered messages. If your backend has any concept of a persistent session or connection, you should also implement the `open()` and `close()` methods. Refer to `smtp.EmailBackend` for a reference implementation.

Sending multiple emails

Establishing and closing an SMTP connection (or any other network connection, for that matter) is an expensive process. If you have a lot of emails to send, it makes sense to reuse an SMTP connection, rather than creating and destroying a connection every time you want to send an email.

There are two ways you tell an email backend to reuse a connection.

Firstly, you can use the `send_messages()` method on a connection. This takes a list of *EmailMessage* (or subclass) instances, and sends them all using that single connection. As a consequence, any *connection* set on an individual message is ignored.

For example, if you have a function called `get_notification_email()` that returns a list of *EmailMessage* objects representing some periodic email you wish to send out, you could send these emails using a single call to send_messages:

```
from django.core import mail

connection = mail.get_connection()  # Use default email connection
messages = get_notification_email()
connection.send_messages(messages)
```

In this example, the call to `send_messages()` opens a connection on the backend, sends the list of messages, and then closes the connection again.

The second approach is to use the `open()` and `close()` methods on the email backend to manually control the connection. `send_messages()` will not manually open or close the connection if it is already open, so if you manually open the connection, you can control when it is closed. For example:

```
from django.core import mail

connection = mail.get_connection()

# Manually open the connection
connection.open()

# Construct an email message that uses the connection
email1 = mail.EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to1@example.com"],
    connection=connection,
)
email1.send()  # Send the email

# Construct two more messages
email2 = mail.EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to2@example.com"],
)
email3 = mail.EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to3@example.com"],
)

# Send the two emails in a single call -
connection.send_messages([email2, email3])
# The connection was already open so send_messages() doesn't close it.
# We need to manually close the connection.
connection.close()
```

3.15.10 Configuring email for development

There are times when you do not want Django to send emails at all. For example, while developing a website, you probably don't want to send out thousands of emails – but you may want to validate that emails will be sent to the right people under the right conditions, and that those emails will contain the correct content.

The easiest way to configure email for local development is to use the console email backend. This backend redirects all email to `stdout`, allowing you to inspect the content of mail.

The file email backend can also be useful during development – this backend dumps the contents of every SMTP connection to a file that can be inspected at your leisure.

Another approach is to use a "dumb" SMTP server that receives the emails locally and displays them to the terminal, but does not actually send anything. The aiosmtpd package provides a way to accomplish this:

```
python -m pip install aiosmtpd

python -m aiosmtpd -n -l localhost:8025
```

This command will start a minimal SMTP server listening on port 8025 of localhost. This server prints to standard output all email headers and the email body. You then only need to set the *EMAIL_HOST* and *EMAIL_PORT* accordingly. For a more detailed discussion of SMTP server options, see the documentation of the aiosmtpd module.

For information about unit-testing the sending of emails in your application, see the Email services section of the testing documentation.

3.16 Internationalization and localization

3.16.1 Translation

Overview

In order to make a Django project translatable, you have to add a minimal number of hooks to your Python code and templates. These hooks are called translation strings. They tell Django: "This text should be translated into the end user's language, if a translation for this text is available in that language." It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

Django then provides utilities to extract the translation strings into a message file. This file is a convenient way for translators to provide the equivalent of the translation strings in the target language. Once the translators have filled in the message file, it must be compiled. This process relies on the GNU gettext toolset.

Once this is done, Django takes care of translating web apps on the fly in each available language, according to users' language preferences.

Django's internationalization hooks are on by default, and that means there's a bit of i18n-related overhead in certain places of the framework. If you don't use internationalization, you should take the two seconds to

set *USE_I18N = False* in your settings file. Then Django will make some optimizations so as not to load the internationalization machinery.

> **Note**
>
> Make sure you've activated translation for your project (the fastest way is to check if *MIDDLEWARE* includes *django.middleware.locale.LocaleMiddleware*). If you haven't yet, see How Django discovers language preference.

Internationalization: in Python code

Standard translation

Specify a translation string by using the function *gettext()*. It's convention to import this as a shorter alias, _, to save typing.

> **Note**
>
> Python's standard library `gettext` module installs `_()` into the global namespace, as an alias for `gettext()`. In Django, we have chosen not to follow this practice, for a couple of reasons:
>
> 1. Sometimes, you should use *gettext_lazy()* as the default translation method for a particular file. Without `_()` in the global namespace, the developer has to think about which is the most appropriate translation function.
> 2. The underscore character (_) is used to represent "the previous result" in Python's interactive shell and doctest tests. Installing a global `_()` function causes interference. Explicitly importing `gettext()` as `_()` avoids this problem.

> **What functions may be aliased as _?**
>
> Because of how `xgettext` (used by *makemessages*) works, only functions that take a single string argument can be imported as _:
>
> - *gettext()*
> - *gettext_lazy()*

In this example, the text `"Welcome to my site."` is marked as a translation string:

```
from django.http import HttpResponse
from django.utils.translation import gettext as _
```

(continues on next page)

(continued from previous page)

```
def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

You could code this without using the alias. This example is identical to the previous one:

```
from django.http import HttpResponse
from django.utils.translation import gettext


def my_view(request):
    output = gettext("Welcome to my site.")
    return HttpResponse(output)
```

Translation works on computed values. This example is identical to the previous two:

```
def my_view(request):
    words = ["Welcome", "to", "my", "site."]
    output = _(" ".join(words))
    return HttpResponse(output)
```

Translation works on variables. Again, here's an identical example:

```
def my_view(request):
    sentence = "Welcome to my site."
    output = _(sentence)
    return HttpResponse(output)
```

(The caveat with using variables or computed values, as in the previous two examples, is that Django's translation-string-detecting utility, *django-admin makemessages*, won't be able to find these strings. More on *makemessages* later.)

The strings you pass to `_()` or `gettext()` can take placeholders, specified with Python's standard named-string interpolation syntax. Example:

```
def my_view(request, m, d):
    output = _("Today is %(month)s %(day)s.") % {"month": m, "day": d}
    return HttpResponse(output)
```

This technique lets language-specific translations reorder the placeholder text. For example, an English translation may be `"Today is November 26."`, while a Spanish translation may be `"Hoy es 26 de noviembre."` – with the month and the day placeholders swapped.

For this reason, you should use named-string interpolation (e.g., `%(day)s`) instead of positional interpolation (e.g., `%s` or `%d`) whenever you have more than a single parameter. If you used positional interpolation, translations wouldn't be able to reorder placeholder text.

Since string extraction is done by the `xgettext` command, only syntaxes supported by `gettext` are supported by Django. Python f-strings cannot be used directly with `gettext` functions because f-string expressions are evaluated before they reach `gettext`. This means `_(f"Welcome {name}")` will not work as expected, as the variable is substituted before translation occurs. Instead, use named-string interpolation:

```
# Good
_("Welcome %(name)s") % {"name": name}

# Good
_("Welcome {name}").format(name=name)

# Bad
_(f"Welcome {name}")  # f-string evaluated before translation.
```

JavaScript template strings need `gettext` 0.21+.

Comments for translators

If you would like to give translators hints about a translatable string, you can add a comment prefixed with the `Translators` keyword on the line preceding the string, e.g.:

```
def my_view(request):
    # Translators: This message appears on the home page only
    output = gettext("Welcome to my site.")
```

The comment will then appear in the resulting `.po` file associated with the translatable construct located below it and should also be displayed by most translation tools.

> **Note**
>
> Just for completeness, this is the corresponding fragment of the resulting `.po` file:
>
> ```
> #. Translators: This message appears on the home page only
> # path/to/python/file.py:123
> msgid "Welcome to my site."
> msgstr ""
> ```

This also works in templates. See Comments for translators in templates for more details.

Marking strings as no-op

Use the function *django.utils.translation.gettext_noop()* to mark a string as a translation string without translating it. The string is later translated from a variable.

Use this if you have constant strings that should be stored in the source language because they are exchanged over systems or users – such as strings in a database – but should be translated at the last possible point in time, such as when the string is presented to the user.

Pluralization

Use the function *django.utils.translation.ngettext()* to specify pluralized messages.

`ngettext()` takes three arguments: the singular translation string, the plural translation string and the number of objects.

This function is useful when you need your Django application to be localizable to languages where the number and complexity of plural forms is greater than the two forms used in English ('object' for the singular and 'objects' for all the cases where `count` is different from one, irrespective of its value.)

For example:

```
from django.http import HttpResponse
from django.utils.translation import ngettext


def hello_world(request, count):
    page = ngettext(
        "there is %(count)d object",
        "there are %(count)d objects",
        count,
    ) % {
        "count": count,
    }
    return HttpResponse(page)
```

In this example the number of objects is passed to the translation languages as the `count` variable.

Note that pluralization is complicated and works differently in each language. Comparing `count` to 1 isn't always the correct rule. This code looks sophisticated, but will produce incorrect results for some languages:

```
from django.utils.translation import ngettext
from myapp.models import Report

count = Report.objects.count()
```

(continues on next page)

(continued from previous page)

```
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural

text = ngettext(
    "There is %(count)d %(name)s available.",
    "There are %(count)d %(name)s available.",
    count,
) % {"count": count, "name": name}
```

Don't try to implement your own singular-or-plural logic; it won't be correct. In a case like this, consider something like the following:

```
text = ngettext(
    "There is %(count)d %(name)s object available.",
    "There are %(count)d %(name)s objects available.",
    count,
) % {
    "count": count,
    "name": Report._meta.verbose_name,
}
```

Note

When using `ngettext()`, make sure you use a single name for every extrapolated variable included in the literal. In the examples above, note how we used the `name` Python variable in both translation strings. This example, besides being incorrect in some languages as noted above, would fail:

```
text = ngettext(
    "There is %(count)d %(name)s available.",
    "There are %(count)d %(plural_name)s available.",
    count,
) % {
    "count": Report.objects.count(),
    "name": Report._meta.verbose_name,
    "plural_name": Report._meta.verbose_name_plural,
}
```

You would get an error when running *`django-admin compilemessages`*:

```
a format specification for argument 'name', as in 'msgstr[0]', doesn't exist in 'msgid
↪'
```

Contextual markers

Sometimes words have several meanings, such as `"May"` in English, which refers to a month name and to a verb. To enable translators to translate these words correctly in different contexts, you can use the *django.utils.translation.pgettext()* function, or the *django.utils.translation.npgettext()* function if the string needs pluralization. Both take a context string as the first variable.

In the resulting `.po` file, the string will then appear as often as there are different contextual markers for the same string (the context will appear on the `msgctxt` line), allowing the translator to give a different translation for each of them.

For example:

```
from django.utils.translation import pgettext

month = pgettext("month name", "May")
```

or:

```
from django.db import models
from django.utils.translation import pgettext_lazy


class MyThing(models.Model):
    name = models.CharField(
        help_text=pgettext_lazy("help text for MyThing model", "This is the help text")
    )
```

will appear in the `.po` file as:

```
msgctxt "month name"
msgid "May"
msgstr ""
```

Contextual markers are also supported by the *translate* and *blocktranslate* template tags.

Lazy translation

Use the lazy versions of translation functions in *django.utils.translation* (easily recognizable by the `lazy` suffix in their names) to translate strings lazily – when the value is accessed rather than when they're called.

These functions store a lazy reference to the string – not the actual translation. The translation itself will be done when the string is used in a string context, such as in template rendering.

This is essential when calls to these functions are located in code paths that are executed at module load time.

This is something that can easily happen when defining models, forms and model forms, because Django implements these such that their fields are actually class-level attributes. For that reason, make sure to use lazy translations in the following cases:

Model fields and relationships `verbose_name` and `help_text` option values

For example, to translate the help text of the name field in the following model, do the following:

```
from django.db import models
from django.utils.translation import gettext_lazy as _


class MyThing(models.Model):
    name = models.CharField(help_text=_("This is the help text"))
```

You can mark names of *ForeignKey*, *ManyToManyField* or *OneToOneField* relationship as translatable by using their *verbose_name* options:

```
class MyThing(models.Model):
    kind = models.ForeignKey(
        ThingKind,
        on_delete=models.CASCADE,
        related_name="kinds",
        verbose_name=_("kind"),
    )
```

Just like you would do in *verbose_name* you should provide a lowercase verbose name text for the relation as Django will automatically titlecase it when required.

Model verbose names values

It is recommended to always provide explicit *verbose_name* and *verbose_name_plural* options rather than relying on the fallback English-centric and somewhat naïve determination of verbose names Django performs by looking at the model's class name:

```
from django.db import models
from django.utils.translation import gettext_lazy as _


class MyThing(models.Model):
    name = models.CharField(_("name"), help_text=_("This is the help text"))
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        verbose_name = _("my thing")
        verbose_name_plural = _("my things")
```

Model methods `description` argument to the `@display` decorator

For model methods, you can provide translations to Django and the admin site with the `description` argument to the *`display()`* decorator:

```
from django.contrib import admin
from django.db import models
from django.utils.translation import gettext_lazy as _


class MyThing(models.Model):
    kind = models.ForeignKey(
        ThingKind,
        on_delete=models.CASCADE,
        related_name="kinds",
        verbose_name=_("kind"),
    )

    @admin.display(description=_("Is it a mouse?"))
    def is_mouse(self):
        return self.kind.type == MOUSE_TYPE
```

Working with lazy translation objects

The result of a `gettext_lazy()` call can be used wherever you would use a string (a `str` object) in other Django code, but it may not work with arbitrary Python code. For example, the following won't work because the requests library doesn't handle `gettext_lazy` objects:

```
body = gettext_lazy("I \u2764 Django")  # (Unicode :heart:)
requests.post("https://example.com/send", data={"body": body})
```

You can avoid such problems by casting `gettext_lazy()` objects to text strings before passing them to non-Django code:

```
requests.post("https://example.com/send", data={"body": str(body)})
```

If you don't like the long `gettext_lazy` name, you can alias it as `_` (underscore), like so:

```
from django.db import models
from django.utils.translation import gettext_lazy as _


class MyThing(models.Model):
    name = models.CharField(help_text=_("This is the help text"))
```

Using `gettext_lazy()` and `ngettext_lazy()` to mark strings in models and utility functions is a common operation. When you're working with these objects elsewhere in your code, you should ensure that you don't accidentally convert them to strings, because they should be converted as late as possible (so that the correct locale is in effect). This necessitates the use of the helper function described next.

Lazy translations and plural

When using lazy translation for a plural string (`n[p]gettext_lazy`), you generally don't know the `number` argument at the time of the string definition. Therefore, you are authorized to pass a key name instead of an integer as the `number` argument. Then `number` will be looked up in the dictionary under that key during string interpolation. Here's example:

```
from django import forms
from django.core.exceptions import ValidationError
from django.utils.translation import ngettext_lazy


class MyForm(forms.Form):
    error_message = ngettext_lazy(
        "You only provided %(num)d argument",
        "You only provided %(num)d arguments",
        "num",
    )

    def clean(self):
        # ...
        if error:
            raise ValidationError(self.error_message % {"num": number})
```

If the string contains exactly one unnamed placeholder, you can interpolate directly with the `number` argument:

```
class MyForm(forms.Form):
    error_message = ngettext_lazy(
        "You provided %d argument",
```

(continues on next page)

(continued from previous page)

```
        "You provided %d arguments",
    )

    def clean(self):
        # ...
        if error:
            raise ValidationError(self.error_message % number)
```

Formatting strings: `format_lazy()`

Python's `str.format()` method will not work when either the `format_string` or any of the arguments to `str.format()` contains lazy translation objects. Instead, you can use *`django.utils.text.format_lazy()`*, which creates a lazy object that runs the `str.format()` method only when the result is included in a string. For example:

```
from django.utils.text import format_lazy
from django.utils.translation import gettext_lazy

...
name = gettext_lazy("John Lennon")
instrument = gettext_lazy("guitar")
result = format_lazy("{name}: {instrument}", name=name, instrument=instrument)
```

In this case, the lazy translations in `result` will only be converted to strings when `result` itself is used in a string (usually at template rendering time).

Other uses of lazy in delayed translations

For any other case where you would like to delay the translation, but have to pass the translatable string as argument to another function, you can wrap this function inside a lazy call yourself. For example:

```
from django.utils.functional import lazy
from django.utils.translation import gettext_lazy as _


def to_lower(string):
    return string.lower()


to_lower_lazy = lazy(to_lower, str)
```

And then later:

```
lazy_string = to_lower_lazy(_("My STRING!"))
```

Localized names of languages

`get_language_info`(lang_code)

The `get_language_info()` function provides detailed information about languages:

```
>>> from django.utils.translation import activate, get_language_info
>>> activate("fr")
>>> li = get_language_info("de")
>>> print(li["name"], li["name_local"], li["name_translated"], li["bidi"])
German Deutsch Allemand False
```

The `name`, `name_local`, and `name_translated` attributes of the dictionary contain the name of the language in English, in the language itself, and in your current active language respectively. The `bidi` attribute is True only for bi-directional languages.

The source of the language information is the `django.conf.locale` module. Similar access to this information is available for template code. See below.

Internationalization: in template code

Translations in Django templates uses two template tags and a slightly different syntax than in Python code. To give your template access to these tags, put `{% load i18n %}` toward the top of your template. As with all template tags, this tag needs to be loaded in all templates which use translations, even those templates that extend from other templates which have already loaded the `i18n` tag.

> **Warning**
>
> Translated strings will not be escaped when rendered in a template. This allows you to include HTML in translations, for example for emphasis, but potentially dangerous characters (e.g. `"`) will also be rendered unchanged.

`translate` template tag

The `{% translate %}` template tag translates either a constant string (enclosed in single or double quotes) or variable content:

```
<title>{% translate "This is the title." %}</title>
<title>{% translate myvar %}</title>
```

If the `noop` option is present, variable lookup still takes place but the translation is skipped. This is useful when "stubbing out" content that will require translation in the future:

```
<title>{% translate "myvar" noop %}</title>
```

Internally, inline translations use a *gettext()* call.

In case a template var (`myvar` above) is passed to the tag, the tag will first resolve such variable to a string at run-time and then look up that string in the message catalogs.

It's not possible to mix a template variable inside a string within `{% translate %}`. If your translations require strings with variables (placeholders), use *{% blocktranslate %}* instead.

If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% translate "This is the title" as the_title %}

<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get a string you can use in multiple places in a template or so you can use the output as an argument for other template tags or filters:

```
{% translate "starting point" as start %}
{% translate "end point" as end %}
{% translate "La Grande Boucle" as race %}

<h1>
  <a href="/" title="{% blocktranslate %}Back to '{{ race }}' homepage{%␣
↪endblocktranslate %}">{{ race }}</a>
</h1>
<p>
{% for stage in tour_stages %}
    {% cycle start end %}: {{ stage }}{% if forloop.counter|divisibleby:2 %}<br>{% else
↪%}, {% endif %}
{% endfor %}
</p>
```

`{% translate %}` also supports contextual markers using the `context` keyword:

```
{% translate "May" context "month name" %}
```

`blocktranslate` template tag

Contrarily to the *`translate`* tag, the `blocktranslate` tag allows you to mark complex sentences consisting of literals and variable content for translation by making use of placeholders:

```
{% blocktranslate %}This string will have {{ value }} inside.{% endblocktranslate %}
```

To translate a template expression – say, accessing object attributes or using template filters – you need to bind the expression to a local variable for use within the translation block. Examples:

```
{% blocktranslate with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktranslate %}

{% blocktranslate with myvar=value|filter %}
This will have {{ myvar }} inside.
{% endblocktranslate %}
```

You can use multiple expressions inside a single `blocktranslate` tag:

```
{% blocktranslate with book_t=book|title author_t=author|title %}
This is {{ book_t }} by {{ author_t }}
{% endblocktranslate %}
```

> **Note**
>
> The previous more verbose format is still supported: `{% blocktranslate with book|title as book_t and author|title as author_t %}`

Other block tags (for example *`{% for %}`* or *`{% if %}`*) are not allowed inside a `blocktranslate` tag.

If resolving one of the block arguments fails, `blocktranslate` will fall back to the default language by deactivating the currently active language temporarily with the *`deactivate_all()`* function.

This tag also provides for pluralization. To use it:

- Designate and bind a counter value with the name `count`. This value will be the one used to select the right plural form.
- Specify both the singular and plural forms separating them with the `{% plural %}` tag within the `{% blocktranslate %}` and `{% endblocktranslate %}` tags.

An example:

```
{% blocktranslate count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktranslate %}
```

A more complex example:

```
{% blocktranslate with amount=article.price count years=i.length %}
That will cost $ {{ amount }} per year.
{% plural %}
That will cost $ {{ amount }} per {{ years }} years.
{% endblocktranslate %}
```

When you use both the pluralization feature and bind values to local variables in addition to the counter value, keep in mind that the `blocktranslate` construct is internally converted to an `ngettext` call. This means the same notes regarding ngettext variables apply.

Reverse URL lookups cannot be carried out within the `blocktranslate` and should be retrieved (and stored) beforehand:

```
{% url 'path.to.view' arg arg2 as the_url %}
{% blocktranslate %}
This is a URL: {{ the_url }}
{% endblocktranslate %}
```

If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% blocktranslate asvar the_title %}The title is {{ title }}.{% endblocktranslate %}
<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get a string you can use in multiple places in a template or so you can use the output as an argument for other template tags or filters.

`{% blocktranslate %}` also supports contextual markers using the `context` keyword:

```
{% blocktranslate with name=user.username context "greeting" %}Hi {{ name }}{%␣
↪endblocktranslate %}
```

Another feature `{% blocktranslate %}` supports is the `trimmed` option. This option will remove newline characters from the beginning and the end of the content of the `{% blocktranslate %}` tag, replace any whitespace at the beginning and end of a line and merge all lines into one using a space character to separate them. This is quite useful for indenting the content of a `{% blocktranslate %}` tag without having the

indentation characters end up in the corresponding entry in the `.po` file, which makes the translation process easier.

For instance, the following `{% blocktranslate %}` tag:

```
{% blocktranslate trimmed %}
  First sentence.
  Second paragraph.
{% endblocktranslate %}
```

will result in the entry `"First sentence. Second paragraph."` in the `.po` file, compared to `"\n First sentence.\n Second paragraph.\n"`, if the `trimmed` option had not been specified.

String literals passed to tags and filters

You can translate string literals passed as arguments to tags and filters by using the familiar `_()` syntax:

```
{% some_tag _("Page not found") value|yesno:_("yes,no") %}
```

In this case, both the tag and the filter will see the translated string, so they don't need to be aware of translations.

> **Note**
>
> In this example, the translation infrastructure will be passed the string `"yes,no"`, not the individual strings `"yes"` and `"no"`. The translated string will need to contain the comma so that the filter parsing code knows how to split up the arguments. For example, a German translator might translate the string `"yes,no"` as `"ja,nein"` (keeping the comma intact).

Comments for translators in templates

Just like with Python code, these notes for translators can be specified using comments, either with the *comment* tag:

```
{% comment %}Translators: View verb{% endcomment %}
{% translate "View" %}

{% comment %}Translators: Short intro blurb{% endcomment %}
<p>{% blocktranslate %}A multiline translatable
literal.{% endblocktranslate %}</p>
```

or with the `{# ... #}` one-line comment constructs:

```
{# Translators: Label of a button that triggers search #}
<button type="submit">{% translate "Go" %}</button>

{# Translators: This is a text of the base template #}
{% blocktranslate %}Ambiguous translatable block of text{% endblocktranslate %}
```

Note

Just for completeness, these are the corresponding fragments of the resulting `.po` file:

```
#. Translators: View verb
# path/to/template/file.html:10
msgid "View"
msgstr ""

#. Translators: Short intro blurb
# path/to/template/file.html:13
msgid ""
"A multiline translatable"
"literal."
msgstr ""

# ...

#. Translators: Label of a button that triggers search
# path/to/template/file.html:100
msgid "Go"
msgstr ""

#. Translators: This is a text of the base template
# path/to/template/file.html:103
msgid "Ambiguous translatable block of text"
msgstr ""
```

Switching language in templates

If you want to select a language within a template, you can use the `language` template tag:

```
{% load i18n %}
```

(continues on next page)

(continued from previous page)

```
{% get_current_language as LANGUAGE_CODE %}
<!-- Current language: {{ LANGUAGE_CODE }} -->
<p>{% translate "Welcome to our page" %}</p>

{% language 'en' %}
    {% get_current_language as LANGUAGE_CODE %}
    <!-- Current language: {{ LANGUAGE_CODE }} -->
    <p>{% translate "Welcome to our page" %}</p>
{% endlanguage %}
```

While the first occurrence of "Welcome to our page" uses the current language, the second will always be in English.

Other tags

These tags also require a `{% load i18n %}`.

`get_available_languages`

`{% get_available_languages as LANGUAGES %}` returns a list of tuples in which the first element is the language code and the second is the language name (translated into the currently active locale).

`get_current_language`

`{% get_current_language as LANGUAGE_CODE %}` returns the current user's preferred language as a string. Example: `en-us`. See How Django discovers language preference.

`get_current_language_bidi`

`{% get_current_language_bidi as LANGUAGE_BIDI %}` returns the current locale's direction. If `True`, it's a right-to-left language, e.g. Hebrew, Arabic. If `False` it's a left-to-right language, e.g. English, French, German, etc.

`i18n` context processor

If you enable the *`django.template.context_processors.i18n`* context processor, then each `RequestContext` will have access to `LANGUAGES`, `LANGUAGE_CODE`, and `LANGUAGE_BIDI` as defined above.

get_language_info

You can also retrieve information about any of the available languages using provided template tags and filters. To get information about a single language, use the `{% get_language_info %}` tag:

```
{% get_language_info for LANGUAGE_CODE as lang %}
{% get_language_info for "pl" as lang %}
```

You can then access the information:

```
Language code: {{ lang.code }}<br>
Name of language: {{ lang.name_local }}<br>
Name in English: {{ lang.name }}<br>
Bi-directional: {{ lang.bidi }}
Name in the active language: {{ lang.name_translated }}
```

get_language_info_list

You can also use the `{% get_language_info_list %}` template tag to retrieve information for a list of languages (e.g. active languages as specified in *LANGUAGES*). See the section about the set_language redirect view for an example of how to display a language selector using `{% get_language_info_list %}`.

In addition to *LANGUAGES* style list of tuples, `{% get_language_info_list %}` supports lists of language codes. If you do this in your view:

```
context = {"available_languages": ["en", "es", "fr"]}
return render(request, "mytemplate.html", context)
```

you can iterate over those languages in the template:

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

Template filters

There are also some filters available for convenience:

- `{{ LANGUAGE_CODE|language_name }}` ("German")
- `{{ LANGUAGE_CODE|language_name_local }}` ("Deutsch")
- `{{ LANGUAGE_CODE|language_bidi }}` (False)
- `{{ LANGUAGE_CODE|language_name_translated }}` ("německy", when active language is Czech)

Internationalization: in JavaScript code

Adding translations to JavaScript poses some problems:

- JavaScript code doesn't have access to a `gettext` implementation.
- JavaScript code doesn't have access to `.po` or `.mo` files; they need to be delivered by the server.
- The translation catalogs for JavaScript should be kept as small as possible.

Django provides an integrated solution for these problems: It passes the translations into JavaScript, so you can call `gettext`, etc., from within JavaScript.

The main solution to these problems is the following `JavaScriptCatalog` view, which generates a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings.

The `JavaScriptCatalog` view

class JavaScriptCatalog

A view that produces a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings.

Attributes

domain

Translation domain containing strings to add in the view output. Defaults to `'djangojs'`.

packages

A list of *application names* among installed applications. Those apps should contain a `locale` directory. All those catalogs plus all catalogs found in *LOCALE_PATHS* (which are always included) are merged into one catalog. Defaults to `None`, which means that all available translations from all *INSTALLED_APPS* are provided in the JavaScript output.

Example with default values:

```
from django.views.i18n import JavaScriptCatalog

urlpatterns = [
    path("jsi18n/", JavaScriptCatalog.as_view(), name="javascript-catalog"),
]
```

Example with custom packages:

```
urlpatterns = [
    path(
        "jsi18n/myapp/",
        JavaScriptCatalog.as_view(packages=["your.app.label"]),
        name="javascript-catalog",
```

(continues on next page)

(continued from previous page)

```
    ),
]
```

If your root URLconf uses *i18n_patterns()*, `JavaScriptCatalog` must also be wrapped by `i18n_patterns()` for the catalog to be correctly generated.

Example with `i18n_patterns()`:

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    path("jsi18n/", JavaScriptCatalog.as_view(), name="javascript-catalog"),
)
```

The precedence of translations is such that the packages appearing later in the `packages` argument have higher precedence than the ones appearing at the beginning. This is important in the case of clashing translations for the same literal.

If you use more than one `JavaScriptCatalog` view on a site and some of them define the same strings, the strings in the catalog that was loaded last take precedence.

Using the JavaScript translation catalog

To use the catalog, pull in the dynamically generated script like this:

```
<script src="{% url 'javascript-catalog' %}"></script>
```

This uses reverse URL lookup to find the URL of the JavaScript catalog view. When the catalog is loaded, your JavaScript code can use the following methods:

- `gettext`
- `ngettext`
- `interpolate`
- `get_format`
- `gettext_noop`
- `pgettext`
- `npgettext`
- `pluralidx`

gettext

The `gettext` function behaves similarly to the standard `gettext` interface within your Python code:

```
document.write(gettext("this is to be translated"))
```

ngettext

The `ngettext` function provides an interface to pluralize words and phrases:

```
const objectCount = 1 // or 0, or 2, or 3, ...
const string = ngettext(
    'literal for the singular case',
    'literal for the plural case',
    objectCount
);
```

interpolate

The `interpolate` function supports dynamically populating a format string. The interpolation syntax is borrowed from Python, so the `interpolate` function supports both positional and named interpolation:

- Positional interpolation: `obj` contains a JavaScript Array object whose elements values are then sequentially interpolated in their corresponding `fmt` placeholders in the same order they appear. For example:

  ```
  const formats = ngettext(
    'There is %s object. Remaining: %s',
    'There are %s objects. Remaining: %s',
    11
  );
  const string = interpolate(formats, [11, 20]);
  // string is 'There are 11 objects. Remaining: 20'
  ```

- Named interpolation: This mode is selected by passing the optional boolean `named` parameter as `true`. `obj` contains a JavaScript object or associative array. For example:

  ```
  const data = {
    count: 10,
    total: 50
  };

  const formats = ngettext(
  ```

(continues on next page)

(continued from previous page)

```
    'Total: %(total)s, there is %(count)s object',
    'there are %(count)s of a total of %(total)s objects',
    data.count
);
const string = interpolate(formats, data, true);
```

You shouldn't go over the top with string interpolation, though: this is still JavaScript, so the code has to make repeated regular-expression substitutions. This isn't as fast as string interpolation in Python, so keep it to those cases where you really need it (for example, in conjunction with `ngettext` to produce proper pluralizations).

get_format

The `get_format` function has access to the configured i18n formatting settings and can retrieve the format string for a given setting name:

```
document.write(get_format('DATE_FORMAT'));
// 'N j, Y'
```

It has access to the following settings:

- *DATE_FORMAT*
- *DATE_INPUT_FORMATS*
- *DATETIME_FORMAT*
- *DATETIME_INPUT_FORMATS*
- *DECIMAL_SEPARATOR*
- *FIRST_DAY_OF_WEEK*
- *MONTH_DAY_FORMAT*
- *NUMBER_GROUPING*
- *SHORT_DATE_FORMAT*
- *SHORT_DATETIME_FORMAT*
- *THOUSAND_SEPARATOR*
- *TIME_FORMAT*
- *TIME_INPUT_FORMATS*
- *YEAR_MONTH_FORMAT*

This is useful for maintaining formatting consistency with the Python-rendered values.

gettext_noop

This emulates the `gettext` function but does nothing, returning whatever is passed to it:

```
document.write(gettext_noop("this will not be translated"))
```

This is useful for stubbing out portions of the code that will need translation in the future.

pgettext

The `pgettext` function behaves like the Python variant (*pgettext()*), providing a contextually translated word:

```
document.write(pgettext("month name", "May"))
```

npgettext

The `npgettext` function also behaves like the Python variant (*npgettext()*), providing a pluralized contextually translated word:

```
document.write(npgettext('group', 'party', 1));
// party
document.write(npgettext('group', 'party', 2));
// parties
```

pluralidx

The `pluralidx` function works in a similar way to the *pluralize* template filter, determining if a given `count` should use a plural form of a word or not:

```
document.write(pluralidx(0));
// true
document.write(pluralidx(1));
// false
document.write(pluralidx(2));
// true
```

In the simplest case, if no custom pluralization is needed, this returns `false` for the integer `1` and `true` for all other numbers.

However, pluralization is not this simple in all languages. If the language does not support pluralization, an empty value is provided.

Additionally, if there are complex rules around pluralization, the catalog view will render a conditional expression. This will evaluate to either a `true` (should pluralize) or `false` (should not pluralize) value.

The `JSONCatalog` view

`class JSONCatalog`

In order to use another client-side library to handle translations, you may want to take advantage of the `JSONCatalog` view. It's similar to *`JavaScriptCatalog`* but returns a JSON response.

See the documentation for *`JavaScriptCatalog`* to learn about possible values and use of the `domain` and `packages` attributes.

The response format is as follows:

```
{
    "catalog": {
        # Translations catalog
    },
    "formats": {
        # Language formats for date, time, etc.
    },
    "plural": "..."  # Expression for plural forms, or null.
}
```

Note on performance

The various JavaScript/JSON i18n views generate the catalog from `.mo` files on every request. Since its output is constant, at least for a given version of a site, it's a good candidate for caching.

Server-side caching will reduce CPU load. It's easily implemented with the *`cache_page()`* decorator. To trigger cache invalidation when your translations change, provide a version-dependent key prefix, as shown in the example below, or map the view at a version-dependent URL:

```
from django.views.decorators.cache import cache_page
from django.views.i18n import JavaScriptCatalog

# The value returned by get_version() must change when translations change.
urlpatterns = [
    path(
        "jsi18n/",
        cache_page(86400, key_prefix="jsi18n-%s" % get_version())(
            JavaScriptCatalog.as_view()
        ),
        name="javascript-catalog",
    ),
]
```

Client-side caching will save bandwidth and make your site load faster. If you're using ETags (*ConditionalGetMiddleware*), you're already covered. Otherwise, you can apply conditional decorators. In the following example, the cache is invalidated whenever you restart your application server:

```
from django.utils import timezone
from django.views.decorators.http import last_modified
from django.views.i18n import JavaScriptCatalog

last_modified_date = timezone.now()

urlpatterns = [
    path(
        "jsi18n/",
        last_modified(lambda req, **kw: last_modified_date)(
            JavaScriptCatalog.as_view()
        ),
        name="javascript-catalog",
    ),
]
```

You can even pre-generate the JavaScript catalog as part of your deployment procedure and serve it as a static file. This radical technique is implemented in django-statici18n.

Internationalization: in URL patterns

Django provides two mechanisms to internationalize URL patterns:

- Adding the language prefix to the root of the URL patterns to make it possible for *LocaleMiddleware* to detect the language to activate from the requested URL.
- Making URL patterns themselves translatable via the *django.utils.translation.gettext_lazy()* function.

> **Warning**
>
> Using either one of these features requires that an active language be set for each request; in other words, you need to have *django.middleware.locale.LocaleMiddleware* in your *MIDDLEWARE* setting.

Language prefix in URL patterns

i18n_patterns(*urls, prefix_default_language=True)

This function can be used in a root URLconf and Django will automatically prepend the current active language code to all URL patterns defined within *i18n_patterns()*.

Setting `prefix_default_language` to `False` removes the prefix from the default language (*LANGUAGE_CODE*). This can be useful when adding translations to existing site so that the current URLs won't change.

Example URL patterns:

```
from django.conf.urls.i18n import i18n_patterns
from django.urls import include, path

from about import views as about_views
from news import views as news_views
from sitemap.views import sitemap

urlpatterns = [
    path("sitemap.xml", sitemap, name="sitemap-xml"),
]

news_patterns = (
    [
        path("", news_views.index, name="index"),
        path("category/<slug:slug>/", news_views.category, name="category"),
        path("<slug:slug>/", news_views.details, name="detail"),
    ],
    "news",
)

urlpatterns += i18n_patterns(
    path("about/", about_views.main, name="about"),
    path("news/", include(news_patterns, namespace="news")),
)
```

After defining these URL patterns, Django will automatically add the language prefix to the URL patterns that were added by the `i18n_patterns` function. Example:

```
>>> from django.urls import reverse
>>> from django.utils.translation import activate

>>> activate("en")
>>> reverse("sitemap-xml")
'/sitemap.xml'
>>> reverse("news:index")
'/en/news/'
```

(continues on next page)

(continued from previous page)

```
>>> activate("nl")
>>> reverse("news:detail", kwargs={"slug": "news-slug"})
'/nl/news/news-slug/'
```

With `prefix_default_language=False` and `LANGUAGE_CODE='en'`, the URLs will be:

```
>>> activate("en")
>>> reverse("news:index")
'/news/'

>>> activate("nl")
>>> reverse("news:index")
'/nl/news/'
```

> **Warning**
>
> *i18n_patterns()* is only allowed in a root URLconf. Using it within an included URLconf will throw an *ImproperlyConfigured* exception.

> **Warning**
>
> Ensure that you don't have non-prefixed URL patterns that might collide with an automatically-added language prefix.

Translating URL patterns

URL patterns can also be marked translatable using the *gettext_lazy()* function. Example:

```
from django.conf.urls.i18n import i18n_patterns
from django.urls import include, path
from django.utils.translation import gettext_lazy as _

from about import views as about_views
from news import views as news_views
from sitemaps.views import sitemap

urlpatterns = [
    path("sitemap.xml", sitemap, name="sitemap-xml"),
]
```

(continues on next page)

(continued from previous page)

```
news_patterns = (
    [
        path("", news_views.index, name="index"),
        path(_("category/<slug:slug>/"), news_views.category, name="category"),
        path("<slug:slug>/", news_views.details, name="detail"),
    ],
    "news",
)

urlpatterns += i18n_patterns(
    path(_("about/"), about_views.main, name="about"),
    path(_("news/"), include(news_patterns, namespace="news")),
)
```

After you've created the translations, the *reverse()* function will return the URL in the active language. Example:

```
>>> from django.urls import reverse
>>> from django.utils.translation import activate

>>> activate("en")
>>> reverse("news:category", kwargs={"slug": "recent"})
'/en/news/category/recent/'

>>> activate("nl")
>>> reverse("news:category", kwargs={"slug": "recent"})
'/nl/nieuws/categorie/recent/'
```

Warning

In most cases, it's best to use translated URLs only within a language code prefixed block of patterns (using *i18n_patterns()*), to avoid the possibility that a carelessly translated URL causes a collision with a non-translated URL pattern.

Reversing in templates

If localized URLs get reversed in templates they always use the current language. To link to a URL in another language use the *language* template tag. It enables the given language in the enclosed template section:

```
{% load i18n %}

{% get_available_languages as languages %}

{% translate "View this category in:" %}
{% for lang_code, lang_name in languages %}
    {% language lang_code %}
    <a href="{% url 'category' slug=category.slug %}">{{ lang_name }}</a>
    {% endlanguage %}
{% endfor %}
```

The *language* tag expects the language code as the only argument.

Localization: how to create language files

Once the string literals of an application have been tagged for later translation, the translation themselves need to be written (or obtained). Here's how that works.

Message files

The first step is to create a message file for a new language. A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a `.po` file extension.

Django comes with a tool, *django-admin makemessages*, that automates the creation and upkeep of these files.

> **Gettext utilities**
>
> The `makemessages` command (and `compilemessages` discussed later) use commands from the GNU gettext toolset: `xgettext`, `msgfmt`, `msgmerge` and `msguniq`.
>
> The minimum version of the `gettext` utilities supported is 0.19.

To create or update a message file, run this command:

```
django-admin makemessages -l de
```

...where `de` is the locale name for the message file you want to create. For example, `pt_BR` for Brazilian Portuguese, `de_AT` for Austrian German or `id` for Indonesian.

The script should be run from one of two places:

- The root directory of your Django project (the one that contains `manage.py`).
- The root directory of one of your Django apps.

The script runs over your project source tree or your application source tree and pulls out all strings marked for translation (see How Django discovers translations and be sure *LOCALE_PATHS* is configured correctly). It creates (or updates) a message file in the directory `locale/LANG/LC_MESSAGES`. In the `de` example, the file will be `locale/de/LC_MESSAGES/django.po`.

When you run `makemessages` from the root directory of your project, the extracted strings will be automatically distributed to the proper message files. That is, a string extracted from a file of an app containing a `locale` directory will go in a message file under that directory. A string extracted from a file of an app without any `locale` directory will either go in a message file under the directory listed first in *LOCALE_PATHS* or will generate an error if *LOCALE_PATHS* is empty.

By default *django-admin makemessages* examines every file that has the `.html`, `.txt` or `.py` file extension. If you want to override that default, use the *--extension* or `-e` option to specify the file extensions to examine:

```
django-admin makemessages -l de -e txt
```

Separate multiple extensions with commas and/or use `-e` or `--extension` multiple times:

```
django-admin makemessages -l de -e html,txt -e xml
```

Warning

When creating message files from JavaScript source code you need to use the special `djangojs` domain, not `-e js`.

Using Jinja2 templates?

makemessages doesn't understand the syntax of Jinja2 templates. To extract strings from a project containing Jinja2 templates, use Message Extracting from Babel instead.

Here's an example `babel.cfg` configuration file:

```
# Extraction from Python source files
[python: **.py]

# Extraction from Jinja2 templates
[jinja2: **.jinja]
extensions = jinja2.ext.with_
```

Make sure you list all extensions you're using! Otherwise Babel won't recognize the tags defined by these

extensions and will ignore Jinja2 templates containing them entirely.

Babel provides similar features to *makemessages*, can replace it in general, and doesn't depend on `gettext`. For more information, read its documentation about working with message catalogs.

No gettext?

If you don't have the `gettext` utilities installed, *makemessages* will create empty files. If that's the case, either install the `gettext` utilities or copy the English message file (`locale/en/LC_MESSAGES/django.po`) if available and use it as a starting point, which is an empty translation file.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so *makemessages* works, see gettext on Windows for more information.

Each `.po` file contains a small bit of metadata, such as the translation maintainer's contact information, but the bulk of the file is a list of messages – mappings between translation strings and the actual translated text for the particular language.

For example, if your Django app contained a translation string for the text `"Welcome to my site."`, like so:

```
_("Welcome to my site.")
```

...then *django-admin makemessages* will have created a `.po` file containing the following snippet – a message:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

A quick explanation:

- `msgid` is the translation string, which appears in the source. Don't change it.
- `msgstr` is where you put the language-specific translation. It starts out empty, so it's your responsibility to change it. Make sure you keep the quotes around your translation.
- As a convenience, each message includes, in the form of a comment line prefixed with `#` and located above the `msgid` line, the filename and line number from which the translation string was gleaned.

Long messages are a special case. There, the first string directly after the `msgstr` (or `msgid`) is an empty string. Then the content itself will be written over the next few lines as one string per line. Those strings are

directly concatenated. Don't forget trailing spaces within the strings; otherwise, they'll be tacked together without whitespace!

> **Mind your charset**
>
> Due to the way the `gettext` tools work internally and because we want to allow non-ASCII source strings in Django's core and your applications, you must use UTF-8 as the encoding for your `.po` files (the default when `.po` files are created). This means that everybody will be using the same encoding, which is important when Django processes the `.po` files.

> **Fuzzy entries**
>
> *makemessages* sometimes generates translation entries marked as fuzzy, e.g. when translations are inferred from previously translated strings. By default, fuzzy entries are not processed by *compilemessages*.

To reexamine all source code and templates for new translation strings and update all message files for all languages, run this:

```
django-admin makemessages -a
```

Compiling message files

After you create your message file – and each time you make changes to it – you'll need to compile it into a more efficient form, for use by `gettext`. Do this with the *django-admin compilemessages* utility.

This tool runs over all available `.po` files and creates `.mo` files, which are binary files optimized for use by `gettext`. In the same directory from which you ran *django-admin makemessages*, run *django-admin compilemessages* like this:

```
django-admin compilemessages
```

That's it. Your translations are ready for use.

> **Working on Windows?**
>
> If you're using Windows and need to install the GNU gettext utilities so *django-admin compilemessages* works see gettext on Windows for more information.

> **`.po` files: Encoding and BOM usage.**

Django only supports `.po` files encoded in UTF-8 and without any BOM (Byte Order Mark) so if your text editor adds such marks to the beginning of files by default then you will need to reconfigure it.

Troubleshooting: `gettext()` incorrectly detects `python-format` in strings with percent signs

In some cases, such as strings with a percent sign followed by a space and a string conversion type (e.g. `_("10% interest")`), *`gettext()`* incorrectly flags strings with `python-format`.

If you try to compile message files with incorrectly flagged strings, you'll get an error message like `number of format specifications in 'msgid' and 'msgstr' does not match` or `'msgstr' is not a valid Python format string, unlike 'msgid'`.

To workaround this, you can escape percent signs by adding a second percent sign:

```
from django.utils.translation import gettext as _

output = _("10%% interest")
```

Or you can use `no-python-format` so that all percent signs are treated as literals:

```
# xgettext:no-python-format
output = _("10% interest")
```

Creating message files from JavaScript source code

You create and update the message files the same way as the other Django message files – with the *`django-admin makemessages`* tool. The only difference is you need to explicitly specify what in gettext parlance is known as a domain in this case the `djangojs` domain, by providing a `-d djangojs` parameter, like this:

```
django-admin makemessages -d djangojs -l de
```

This would create or update the message file for JavaScript for German. After updating message files, run *`django-admin compilemessages`* the same way as you do with normal Django message files.

`gettext` on Windows

This is only needed for people who either want to extract message IDs or compile message files (`.po`). Translation work itself involves editing existing files of this type, but if you want to create your own message files, or want to test or compile a changed message file, download a precompiled binary installer.

You may also use `gettext` binaries you have obtained elsewhere, so long as the `xgettext --version` command works properly. Do not attempt to use Django translation utilities with a `gettext` package if the

command `xgettext --version` entered at a Windows command prompt causes a popup window saying "`xgettext.exe` has generated errors and will be closed by Windows".

Customizing the `makemessages` command

If you want to pass additional parameters to `xgettext`, you need to create a custom *`makemessages`* command and override its `xgettext_options` attribute:

```
from django.core.management.commands import makemessages


class Command(makemessages.Command):
    xgettext_options = makemessages.Command.xgettext_options + ["--keyword=mytrans"]
```

If you need more flexibility, you could also add a new argument to your custom *`makemessages`* command:

```
from django.core.management.commands import makemessages


class Command(makemessages.Command):
    def add_arguments(self, parser):
        super().add_arguments(parser)
        parser.add_argument(
            "--extra-keyword",
            dest="xgettext_keywords",
            action="append",
        )

    def handle(self, *args, **options):
        xgettext_keywords = options.pop("xgettext_keywords")
        if xgettext_keywords:
            self.xgettext_options = makemessages.Command.xgettext_options[:] + [
                "--keyword=%s" % kwd for kwd in xgettext_keywords
            ]
        super().handle(*args, **options)
```

Miscellaneous

The `set_language` redirect view

`set_language`(request)

As a convenience, Django comes with a view, *`django.views.i18n.set_language()`*, that sets a user's language preference and redirects to a given URL or, by default, back to the previous page.

Activate this view by adding the following line to your URLconf:

```
path("i18n/", include("django.conf.urls.i18n")),
```

(Note that this example makes the view available at `/i18n/setlang/`.)

> **Warning**
>
> Make sure that you don't include the above URL within *i18n_patterns()* - it needs to be language-independent itself to work correctly.

The view expects to be called via the `POST` method, with a `language` parameter set in request. If session support is enabled, the view saves the language choice in the user's session. It also saves the language choice in a cookie that is named `django_language` by default. (The name can be changed through the *LANGUAGE_COOKIE_NAME* setting.)

After setting the language choice, Django looks for a `next` parameter in the `POST` or `GET` data. If that is found and Django considers it to be a safe URL (i.e. it doesn't point to a different host and uses a safe scheme), a redirect to that URL will be performed. Otherwise, Django may fall back to redirecting the user to the URL from the `Referer` header or, if it is not set, to `/`, depending on the nature of the request:

- If the request accepts HTML content (based on its `Accept` HTTP header), the fallback will always be performed.
- If the request doesn't accept HTML, the fallback will be performed only if the `next` parameter was set. Otherwise a 204 status code (No Content) will be returned.

Here's example HTML template code:

```
{% load i18n %}

<form action="{% url 'set_language' %}" method="post">{% csrf_token %}
    <input name="next" type="hidden" value="{{ redirect_to }}">
    <select name="language">
        {% get_current_language as LANGUAGE_CODE %}
        {% get_available_languages as LANGUAGES %}
        {% get_language_info_list for LANGUAGES as languages %}
        {% for language in languages %}
            <option value="{{ language.code }}"{% if language.code == LANGUAGE_CODE %}␣
↪selected{% endif %}>
                {{ language.name_local }} ({{ language.code }})
            </option>
        {% endfor %}
    </select>
```

(continues on next page)

(continued from previous page)

```
    <input type="submit" value="Go">
</form>
```

In this example, Django looks up the URL of the page to which the user will be redirected in the `redirect_to` context variable.

Explicitly setting the active language

You may want to set the active language for the current session explicitly. Perhaps a user's language preference is retrieved from another system, for example. You've already been introduced to *django.utils.translation.activate()*. That applies to the current thread only. To persist the language for the entire session in a cookie, set the *LANGUAGE_COOKIE_NAME* cookie on the response:

```
from django.conf import settings
from django.http import HttpResponse
from django.utils import translation

user_language = "fr"
translation.activate(user_language)
response = HttpResponse(...)
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, user_language)
```

You would typically want to use both: *django.utils.translation.activate()* changes the language for this thread, and setting the cookie makes this preference persist in future requests.

Using translations outside views and templates

While Django provides a rich set of i18n tools for use in views and templates, it does not restrict the usage to Django-specific code. The Django translation mechanisms can be used to translate arbitrary texts to any language that is supported by Django (as long as an appropriate translation catalog exists, of course). You can load a translation catalog, activate it and translate text to language of your choice, but remember to switch back to original language, as activating a translation catalog is done on per-thread basis and such change will affect code running in the same thread.

For example:

```
from django.utils import translation


def welcome_translated(language):
    cur_language = translation.get_language()
    try:
```

(continues on next page)

(continued from previous page)

```
        translation.activate(language)
        text = translation.gettext("welcome")
    finally:
        translation.activate(cur_language)
    return text
```

Calling this function with the value `'de'` will give you `"Willkommen"`, regardless of *LANGUAGE_CODE* and language set by middleware.

Functions of particular interest are *django.utils.translation.get_language()* which returns the language used in the current thread, *django.utils.translation.activate()* which activates a translation catalog for the current thread, and *django.utils.translation.check_for_language()* which checks if the given language is supported by Django.

To help write more concise code, there is also a context manager *django.utils.translation.override()* that stores the current language on enter and restores it on exit. With it, the above example becomes:

```
from django.utils import translation


def welcome_translated(language):
    with translation.override(language):
        return translation.gettext("welcome")
```

Language cookie

A number of settings can be used to adjust language cookie options:

- *LANGUAGE_COOKIE_NAME*
- *LANGUAGE_COOKIE_AGE*
- *LANGUAGE_COOKIE_DOMAIN*
- *LANGUAGE_COOKIE_HTTPONLY*
- *LANGUAGE_COOKIE_PATH*
- *LANGUAGE_COOKIE_SAMESITE*
- *LANGUAGE_COOKIE_SECURE*

Implementation notes

Specialties of Django translation

Django's translation machinery uses the standard `gettext` module that comes with Python. If you know `gettext`, you might note these specialties in the way Django does translation:

- The string domain is `django` or `djangojs`. This string domain is used to differentiate between different programs that store their data in a common message-file library (usually `/usr/share/locale/`). The `django` domain is used for Python and template translation strings and is loaded into the global translation catalogs. The `djangojs` domain is only used for JavaScript translation catalogs to make sure that those are as small as possible.
- Django doesn't use `xgettext` alone. It uses Python wrappers around `xgettext` and `msgfmt`. This is mostly for convenience.

How Django discovers language preference

Once you've prepared your translations – or, if you want to use the translations that come with Django – you'll need to activate translation for your app.

Behind the scenes, Django has a very flexible model of deciding which language should be used – installation-wide, for a particular user, or both.

To set an installation-wide language preference, set *LANGUAGE_CODE*. Django uses this language as the default translation – the final attempt if no better matching translation is found through one of the methods employed by the locale middleware (see below).

If all you want is to run Django with your native language all you need to do is set *LANGUAGE_CODE* and make sure the corresponding message files and their compiled versions (`.mo`) exist.

If you want to let each individual user specify which language they prefer, then you also need to use the `LocaleMiddleware`. `LocaleMiddleware` enables language selection based on data from the request. It customizes content for each user.

To use `LocaleMiddleware`, add `'django.middleware.locale.LocaleMiddleware'` to your *MIDDLEWARE* setting. Because middleware order matters, follow these guidelines:

- Make sure it's one of the first middleware installed.
- It should come after `SessionMiddleware`, because `LocaleMiddleware` makes use of session data. And it should come before `CommonMiddleware` because `CommonMiddleware` needs an activated language in order to resolve the requested URL.
- If you use `CacheMiddleware`, put `LocaleMiddleware` after it.

For example, your *MIDDLEWARE* might look like this:

```
MIDDLEWARE = [
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.locale.LocaleMiddleware",
    "django.middleware.common.CommonMiddleware",
]
```

(For more on middleware, see the middleware documentation.)

`LocaleMiddleware` tries to determine the user's language preference by following this algorithm:

- First, it looks for the language prefix in the requested URL. This is only performed when you are using the `i18n_patterns` function in your root URLconf. See Internationalization: in URL patterns for more information about the language prefix and how to internationalize URL patterns.
- Failing that, it looks for a cookie.

  The name of the cookie used is set by the *LANGUAGE_COOKIE_NAME* setting. (The default name is `django_language`.)
- Failing that, it looks at the `Accept-Language` HTTP header. This header is sent by your browser and tells the server which language(s) you prefer, in order by priority. Django tries each language in the header until it finds one with available translations.
- Failing that, it uses the global *LANGUAGE_CODE* setting.

Notes:

- In each of these places, the language preference is expected to be in the standard language format, as a string. For example, Brazilian Portuguese is `pt-br`.
- If a base language is available but the sublanguage specified is not, Django uses the base language. For example, if a user specifies `de-at` (Austrian German) but Django only has `de` available, Django uses `de`.
- Only languages listed in the *LANGUAGES* setting can be selected. If you want to restrict the language selection to a subset of provided languages (because your application doesn't provide all those languages), set *LANGUAGES* to a list of languages. For example:

  ```
  LANGUAGES = [
      ("de", _("German")),
      ("en", _("English")),
  ]
  ```

  This example restricts languages that are available for automatic selection to German and English (and any sublanguage, like `de-ch` or `en-us`).
- If you define a custom *LANGUAGES* setting, as explained in the previous bullet, you can mark the language names as translation strings – but use *gettext_lazy()* instead of *gettext()* to avoid a circular import.

Here's a sample settings file:

```
from django.utils.translation import gettext_lazy as _

LANGUAGES = [
    ("de", _("German")),
    ("en", _("English")),
]
```

Once `LocaleMiddleware` determines the user's preference, it makes this preference available as `request.LANGUAGE_CODE` for each *`HttpRequest`*. Feel free to read this value in your view code. Here's an example:

```
from django.http import HttpResponse


def hello_world(request, count):
    if request.LANGUAGE_CODE == "de-at":
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Note that, with static (middleware-less) translation, the language is in `settings.LANGUAGE_CODE`, while with dynamic (middleware) translation, it's in `request.LANGUAGE_CODE`.

How Django discovers translations

At runtime, Django builds an in-memory unified catalog of literals-translations. To achieve this it looks for translations by following this algorithm regarding the order in which it examines the different file paths to load the compiled message files (`.mo`) and the precedence of multiple translations for the same literal:

1. The directories listed in *`LOCALE_PATHS`* have the highest precedence, with the ones appearing first having higher precedence than the ones appearing later.
2. Then, it looks for and uses if it exists a `locale` directory in each of the installed apps listed in *`INSTALLED_APPS`*. The ones appearing first have higher precedence than the ones appearing later.
3. Finally, the Django-provided base translation in django/conf/locale is used as a fallback.

See also

The translations for literals included in JavaScript assets are looked up following a similar but not identical algorithm. See *`JavaScriptCatalog`* for more details.

You can also put custom format files in the *`LOCALE_PATHS`* directories if you also set *`FORMAT_MODULE_PATH`*.

In all cases the name of the directory containing the translation is expected to be named using locale name notation. E.g. `de`, `pt_BR`, `es_AR`, etc. Untranslated strings for territorial language variants use the translations of the generic language. For example, untranslated `pt_BR` strings use `pt` translations.

This way, you can write applications that include their own translations, and you can override base translations in your project. Or, you can build a big project out of several apps and put all translations into one big common message file specific to the project you are composing. The choice is yours.

All message file repositories are structured the same way. They are:

- All paths listed in *LOCALE_PATHS* in your settings file are searched for `<language>/LC_MESSAGES/django.(po|mo)`
- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

To create message files, you use the *django-admin makemessages* tool. And you use *django-admin compilemessages* to produce the binary `.mo` files that are used by `gettext`.

You can also run *django-admin compilemessages --settings=path.to.settings* to make the compiler process all the directories in your *LOCALE_PATHS* setting.

Using a non-English base language

Django makes the general assumption that the original strings in a translatable project are written in English. You can choose another language, but you must be aware of certain limitations:

- `gettext` only provides two plural forms for the original messages, so you will also need to provide a translation for the base language to include all plural forms if the plural rules for the base language are different from English.
- When an English variant is activated and English strings are missing, the fallback language will not be the *LANGUAGE_CODE* of the project, but the original strings. For example, an English user visiting a site with *LANGUAGE_CODE* set to Spanish and original strings written in Russian will see Russian text rather than Spanish.

3.16.2 Format localization

Overview

Django's formatting system is capable of displaying dates, times and numbers in templates using the format specified for the current locale. It also handles localized input in forms.

Two users accessing the same content may see dates, times and numbers formatted in different ways, depending on the formats for their current locale.

> **Note**
>
> To enable number formatting with thousand separators, it is necessary to set *USE_THOUSAND_SEPARATOR = True* in your settings file. Alternatively, you could use *intcomma* to format numbers in your template.

> **Note**
>
> There is a related *USE_I18N* setting that controls if Django should activate translation. See Translation for more details.

Locale aware input in forms

When formatting is enabled, Django can use localized formats when parsing dates, times and numbers in forms. That means it tries different formats for different locales when guessing the format used by the user when inputting data on forms.

> **Note**
>
> Django uses different formats for displaying data to those it uses for parsing data. Most notably, the formats for parsing dates can't use the `%a` (abbreviated weekday name), `%A` (full weekday name), `%b` (abbreviated month name), `%B` (full month name), or `%p` (AM/PM).

To enable a form field to localize input and output data use its `localize` argument:

```
class CashRegisterForm(forms.Form):
    product = forms.CharField()
    revenue = forms.DecimalField(max_digits=4, decimal_places=2, localize=True)
```

Controlling localization in templates

Django tries to use a locale specific format whenever it outputs a value in a template.

However, it may not always be appropriate to use localized values – for example, if you're outputting JavaScript or XML that is designed to be machine-readable, you will always want unlocalized values. You may also want to use localization in selected templates, rather than using localization everywhere.

To allow for fine control over the use of localization, Django provides the `l10n` template library that contains the following tags and filters.

Template tags

localize

Enables or disables localization of template variables in the contained block.

To activate or deactivate localization for a template block, use:

```
{% load l10n %}

{% localize on %}
    {{ value }}
{% endlocalize %}

{% localize off %}
    {{ value }}
{% endlocalize %}
```

When localization is disabled, the localization settings formats are applied.

See *localize* and *unlocalize* for template filters that will do the same job on a per-variable basis.

Template filters

localize

Forces localization of a single value.

For example:

```
{% load l10n %}

{{ value|localize }}
```

To disable localization on a single value, use *unlocalize*. To control localization over a large section of a template, use the *localize* template tag.

unlocalize

Forces a single value to be printed without localization.

For example:

```
{% load l10n %}

{{ value|unlocalize }}
```

To force localization of a single value, use *localize*. To control localization over a large section of a template, use the *localize* template tag.

Returns a string representation for numbers (`int`, `float`, or `Decimal`) with the localization settings formats applied.

Creating custom format files

Django provides format definitions for many locales, but sometimes you might want to create your own, because a format file doesn't exist for your locale, or because you want to overwrite some of the values.

To use custom formats, specify the path where you'll place format files first. To do that, set your *FORMAT_MODULE_PATH* setting to the package where format files will exist, for instance:

```
FORMAT_MODULE_PATH = [
    "mysite.formats",
    "some_app.formats",
]
```

Files are not placed directly in this directory, but in a directory named as the locale, and must be named `formats.py`. Be careful not to put sensitive information in these files as values inside can be exposed if you pass the string to `django.utils.formats.get_format()` (used by the *date* template filter).

To customize the English formats, a structure like this would be needed:

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

where `formats.py` contains custom format definitions. For example:

```
THOUSAND_SEPARATOR = "\xa0"
```

to use a non-breaking space (Unicode `00A0`) as a thousand separator, instead of the default for English, a comma.

Limitations of the provided locale formats

Some locales use context-sensitive formats for numbers, which Django's localization system cannot handle automatically.

Switzerland (German)

The Swiss number formatting depends on the type of number that is being formatted. For monetary values, a comma is used as the thousand separator and a decimal point for the decimal separator. For all other numbers, a comma is used as decimal separator and a space as thousand separator. The locale format provided by Django uses the generic separators, a comma for decimal and a space for thousand separators.

3.16.3 Time zones

Overview

When support for time zones is enabled, Django stores datetime information in UTC in the database, uses time-zone-aware datetime objects internally, and converts them to the end user's time zone in forms. Templates will use the default time zone, but this can be updated to the end user's time zone through the use of filters and tags.

This is handy if your users live in more than one time zone and you want to display datetime information according to each user's wall clock.

Even if your website is available in only one time zone, it's still good practice to store data in UTC in your database. The main reason is daylight saving time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. This probably doesn't matter for your blog, but it's a problem if you over bill or under bill your customers by one hour, twice a year, every year. The solution to this problem is to use UTC in the code and use local time only when interacting with end users.

Time zone support is enabled by default. To disable it, set *`USE_TZ = False`* in your settings file.

Time zone support uses `zoneinfo`, which is part of the Python standard library from Python 3.9.

If you're wrestling with a particular problem, start with the time zone FAQ.

Concepts

Naive and aware datetime objects

Python's `datetime.datetime` objects have a `tzinfo` attribute that can be used to store time zone information, represented as an instance of a subclass of `datetime.tzinfo`. When this attribute is set and describes an offset, a datetime object is aware. Otherwise, it's naive.

You can use *`is_aware()`* and *`is_naive()`* to determine whether datetimes are aware or naive.

When time zone support is disabled, Django uses naive datetime objects in local time. This is sufficient for many use cases. In this mode, to obtain the current time, you would write:

```
import datetime

now = datetime.datetime.now()
```

When time zone support is enabled (*USE_TZ=True*), Django uses time-zone-aware datetime objects. If your code creates datetime objects, they should be aware too. In this mode, the example above becomes:

```
from django.utils import timezone

now = timezone.now()
```

Warning

Dealing with aware datetime objects isn't always intuitive. For instance, the `tzinfo` argument of the standard datetime constructor doesn't work reliably for time zones with DST. Using UTC is generally safe; if you're using other time zones, you should review the `zoneinfo` documentation carefully.

Note

Python's `datetime.time` objects also feature a `tzinfo` attribute, and PostgreSQL has a matching `time with time zone` type. However, as PostgreSQL's docs put it, this type "exhibits properties which lead to questionable usefulness".

Django only supports naive time objects and will raise an exception if you attempt to save an aware time object, as a timezone for a time with no associated date does not make sense.

Interpretation of naive datetime objects

When *USE_TZ* is `True`, Django still accepts naive datetime objects, in order to preserve backwards-compatibility. When the database layer receives one, it attempts to make it aware by interpreting it in the default time zone and raises a warning.

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. That's why you should always create aware datetime objects when time zone support is enabled. (See the `Using ZoneInfo section of the zoneinfo docs` for examples using the `fold` attribute to specify the offset that should apply to a datetime during a DST transition.)

In practice, this is rarely an issue. Django gives you aware datetime objects in the models and forms, and most often, new datetime objects are created from existing ones through `timedelta` arithmetic. The only datetime that's often created in application code is the current time, and *timezone.now()* automatically does the right thing.

Default time zone and current time zone

The default time zone is the time zone defined by the *TIME_ZONE* setting.

The current time zone is the time zone that's used for rendering.

You should set the current time zone to the end user's actual time zone with *activate()*. Otherwise, the default time zone is used.

> **Note**
>
> As explained in the documentation of *TIME_ZONE*, Django sets environment variables so that its process runs in the default time zone. This happens regardless of the value of *USE_TZ* and of the current time zone.
>
> When *USE_TZ* is `True`, this is useful to preserve backwards-compatibility with applications that still rely on local time. However, as explained above, this isn't entirely reliable, and you should always work with aware datetimes in UTC in your own code. For instance, use `fromtimestamp()` and set the `tz` parameter to `utc`.

Selecting the current time zone

The current time zone is the equivalent of the current locale for translations. However, there's no equivalent of the `Accept-Language` HTTP header that Django could use to determine the user's time zone automatically. Instead, Django provides time zone selection functions. Use them to build the time zone selection logic that makes sense for you.

Most websites that care about time zones ask users in which time zone they live and store this information in the user's profile. For anonymous users, they use the time zone of their primary audience or UTC. `zoneinfo.available_timezones()` provides a set of available timezones that you can use to build a map from likely locations to time zones.

Here's an example that stores the current timezone in the session. (It skips error handling entirely for the sake of simplicity.)

Add the following middleware to *MIDDLEWARE*:

```
import zoneinfo

from django.utils import timezone


class TimezoneMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, request):
        tzname = request.session.get("django_timezone")
        if tzname:
            timezone.activate(zoneinfo.ZoneInfo(tzname))
        else:
            timezone.deactivate()
        return self.get_response(request)
```

Create a view that can set the current timezone:

```
from django.shortcuts import redirect, render

# Prepare a map of common locations to timezone choices you wish to offer.
common_timezones = {
    "London": "Europe/London",
    "Paris": "Europe/Paris",
    "New York": "America/New_York",
}


def set_timezone(request):
    if request.method == "POST":
        request.session["django_timezone"] = request.POST["timezone"]
        return redirect("/")
    else:
        return render(request, "template.html", {"timezones": common_timezones})
```

Include a form in `template.html` that will `POST` to this view:

```
{% load tz %}
{% get_current_timezone as TIME_ZONE %}
<form action="{% url 'set_timezone' %}" method="POST">
    {% csrf_token %}
    <label for="timezone">Time zone:</label>
    <select name="timezone">
        {% for city, tz in timezones.items %}
        <option value="{{ tz }}"{% if tz == TIME_ZONE %} selected{% endif %}>{{ city }}</
↪option>
        {% endfor %}
    </select>
```

(continues on next page)

(continued from previous page)

```
    <input type="submit" value="Set">
</form>
```

Time zone aware input in forms

When you enable time zone support, Django interprets datetimes entered in forms in the current time zone and returns aware datetime objects in `cleaned_data`.

Converted datetimes that don't exist or are ambiguous because they fall in a DST transition will be reported as invalid values.

Time zone aware output in templates

When you enable time zone support, Django converts aware datetime objects to the current time zone when they're rendered in templates. This behaves very much like format localization.

> **Warning**
>
> Django doesn't convert naive datetime objects, because they could be ambiguous, and because your code should never produce naive datetimes when time zone support is enabled. However, you can force conversion with the template filters described below.

Conversion to local time isn't always appropriate – you may be generating output for computers rather than for humans. The following filters and tags, provided by the `tz` template tag library, allow you to control the time zone conversions.

Template tags

`localtime`

Enables or disables conversion of aware datetime objects to the current time zone in the contained block.

This tag has exactly the same effects as the *`USE_TZ`* setting as far as the template engine is concerned. It allows a more fine grained control of conversion.

To activate or deactivate conversion for a template block, use:

```
{% load tz %}

{% localtime on %}
    {{ value }}
{% endlocaltime %}
```

(continues on next page)

(continued from previous page)

```
{% localtime off %}
    {{ value }}
{% endlocaltime %}
```

> **Note**
>
> The value of *USE_TZ* isn't respected inside of a `{% localtime %}` block.

timezone

Sets or unsets the current time zone in the contained block. When the current time zone is unset, the default time zone applies.

```
{% load tz %}

{% timezone "Europe/Paris" %}
    Paris time: {{ value }}
{% endtimezone %}

{% timezone None %}
    Server time: {{ value }}
{% endtimezone %}
```

get_current_timezone

You can get the name of the current time zone using the `get_current_timezone` tag:

```
{% get_current_timezone as TIME_ZONE %}
```

Alternatively, you can activate the *tz()* context processor and use the `TIME_ZONE` context variable.

Template filters

These filters accept both aware and naive datetimes. For conversion purposes, they assume that naive datetimes are in the default time zone. They always return aware datetimes.

localtime

Forces conversion of a single value to the current time zone.

For example:

```
{% load tz %}

{{ value|localtime }}
```

utc

Forces conversion of a single value to UTC.

For example:

```
{% load tz %}

{{ value|utc }}
```

timezone

Forces conversion of a single value to an arbitrary timezone.

The argument must be an instance of a `tzinfo` subclass or a time zone name.

For example:

```
{% load tz %}

{{ value|timezone:"Europe/Paris" }}
```

Migration guide

Here's how to migrate a project that was started before Django supported time zones.

Database

PostgreSQL

The PostgreSQL backend stores datetimes as `timestamp with time zone`. In practice, this means it converts datetimes from the connection's time zone to UTC on storage, and from UTC to the connection's time zone on retrieval.

As a consequence, if you're using PostgreSQL, you can switch between `USE_TZ = False` and `USE_TZ = True` freely. The database connection's time zone will be set to *DATABASE-TIME_ZONE* or `UTC` respectively, so that Django obtains correct datetimes in all cases. You don't need to perform any data conversions.

> **Time zone settings**
>
> The *time zone* configured for the connection in the *DATABASES* setting is distinct from the general *TIME_ZONE* setting.

Other databases

Other backends store datetimes without time zone information. If you switch from `USE_TZ = False` to `USE_TZ = True`, you must convert your data from local time to UTC – which isn't deterministic if your local time has DST.

Code

The first step is to add *USE_TZ = True* to your settings file. At this point, things should mostly work. If you create naive datetime objects in your code, Django makes them aware when necessary.

However, these conversions may fail around DST transitions, which means you aren't getting the full benefits of time zone support yet. Also, you're likely to run into a few problems because it's impossible to compare a naive datetime with an aware datetime. Since Django now gives you aware datetimes, you'll get exceptions wherever you compare a datetime that comes from a model or a form with a naive datetime that you've created in your code.

So the second step is to refactor your code wherever you instantiate datetime objects to make them aware. This can be done incrementally. *django.utils.timezone* defines some handy helpers for compatibility code: *now()*, *is_aware()*, *is_naive()*, *make_aware()*, and *make_naive()*.

Finally, in order to help you locate code that needs upgrading, Django raises a warning when you attempt to save a naive datetime to the database:

```
RuntimeWarning: DateTimeField ModelName.field_name received a naive
datetime (2012-01-01 00:00:00) while time zone support is active.
```

During development, you can turn such warnings into exceptions and get a traceback by adding the following to your settings file:

```
import warnings

warnings.filterwarnings(
    "error",
    r"DateTimeField .* received a naive datetime",
    RuntimeWarning,
    r"django\.db\.models\.fields",
)
```

Fixtures

When serializing an aware datetime, the UTC offset is included, like this:

```
"2011-09-01T13:20:30+03:00"
```

While for a naive datetime, it isn't:

```
"2011-09-01T13:20:30"
```

For models with *DateTimeField*s, this difference makes it impossible to write a fixture that works both with and without time zone support.

Fixtures generated with `USE_TZ = False`, or before Django 1.4, use the "naive" format. If your project contains such fixtures, after you enable time zone support, you'll see `RuntimeWarning`s when you load them. To get rid of the warnings, you must convert your fixtures to the "aware" format.

You can regenerate fixtures with *loaddata* then *dumpdata*. Or, if they're small enough, you can edit them to add the UTC offset that matches your *TIME_ZONE* to each serialized datetime.

FAQ

Setup

1. I don't need multiple time zones. Should I enable time zone support?

   Yes. When time zone support is enabled, Django uses a more accurate model of local time. This shields you from subtle and unreproducible bugs around daylight saving time (DST) transitions.

   When you enable time zone support, you'll encounter some errors because you're using naive datetimes where Django expects aware datetimes. Such errors show up when running tests. You'll quickly learn how to avoid invalid operations.

   On the other hand, bugs caused by the lack of time zone support are much harder to prevent, diagnose and fix. Anything that involves scheduled tasks or datetime arithmetic is a candidate for subtle bugs that will bite you only once or twice a year.

For these reasons, time zone support is enabled by default in new projects, and you should keep it unless you have a very good reason not to.

2. I've enabled time zone support. Am I safe?

   Maybe. You're better protected from DST-related bugs, but you can still shoot yourself in the foot by carelessly turning naive datetimes into aware datetimes, and vice-versa.

   If your application connects to other systems – for instance, if it queries a web service – make sure datetimes are properly specified. To transmit datetimes safely, their representation should include the UTC offset, or their values should be in UTC (or both!).

   Finally, our calendar system contains interesting edge cases. For example, you can't always subtract one year directly from a given date:

```
>>> import datetime
>>> def one_year_before(value):  # Wrong example.
...     return value.replace(year=value.year - 1)
...
>>> one_year_before(datetime.datetime(2012, 3, 1, 10, 0))
datetime.datetime(2011, 3, 1, 10, 0)
>>> one_year_before(datetime.datetime(2012, 2, 29, 10, 0))
Traceback (most recent call last):
...
ValueError: day is out of range for month
```

   To implement such a function correctly, you must decide whether 2012-02-29 minus one year is 2011-02-28 or 2011-03-01, which depends on your business requirements.

3. How do I interact with a database that stores datetimes in local time?

   Set the *TIME_ZONE* option to the appropriate time zone for this database in the *DATABASES* setting.

   This is useful for connecting to a database that doesn't support time zones and that isn't managed by Django when *USE_TZ* is `True`.

Troubleshooting

1. My application crashes with `TypeError: can't compare offset-naive and offset-aware datetimes` – what's wrong?

   Let's reproduce this error by comparing a naive and an aware datetime:

```
>>> from django.utils import timezone
>>> aware = timezone.now()
>>> naive = timezone.make_naive(aware)
>>> naive == aware
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: can't compare offset-naive and offset-aware datetimes
```

If you encounter this error, most likely your code is comparing these two things:

- a datetime provided by Django – for instance, a value read from a form or a model field. Since you enabled time zone support, it's aware.
- a datetime generated by your code, which is naive (or you wouldn't be reading this).

Generally, the correct solution is to change your code to use an aware datetime instead.

If you're writing a pluggable application that's expected to work independently of the value of *USE_TZ*, you may find *django.utils.timezone.now()* useful. This function returns the current date and time as a naive datetime when `USE_TZ = False` and as an aware datetime when `USE_TZ = True`. You can add or subtract `datetime.timedelta` as needed.

2. I see lots of `RuntimeWarning: DateTimeField received a naive datetime (YYYY-MM-DD HH:MM:SS) while time zone support is active` – is that bad?

   When time zone support is enabled, the database layer expects to receive only aware datetimes from your code. This warning occurs when it receives a naive datetime. This indicates that you haven't finished porting your code for time zone support. Please refer to the migration guide for tips on this process.

   In the meantime, for backwards compatibility, the datetime is considered to be in the default time zone, which is generally what you expect.

3. `now.date()` is yesterday! (or tomorrow)

   If you've always used naive datetimes, you probably believe that you can convert a datetime to a date by calling its `date()` method. You also consider that a `date` is a lot like a `datetime`, except that it's less accurate.

   None of this is true in a time zone aware environment:

```
>>> import datetime
>>> import zoneinfo
>>> paris_tz = zoneinfo.ZoneInfo("Europe/Paris")
>>> new_york_tz = zoneinfo.ZoneInfo("America/New_York")
>>> paris = datetime.datetime(2012, 3, 3, 1, 30, tzinfo=paris_tz)
# This is the correct way to convert between time zones.
>>> new_york = paris.astimezone(new_york_tz)
>>> paris == new_york, paris.date() == new_york.date()
(True, False)
```

(continues on next page)

(continued from previous page)

```
>>> paris - new_york, paris.date() - new_york.date()
(datetime.timedelta(0), datetime.timedelta(1))
>>> paris
datetime.datetime(2012, 3, 3, 1, 30, tzinfo=zoneinfo.ZoneInfo(key='Europe/Paris'))
>>> new_york
datetime.datetime(2012, 3, 2, 19, 30, tzinfo=zoneinfo.ZoneInfo(key='America/New_York
↪'))
```

As this example shows, the same datetime has a different date, depending on the time zone in which it is represented. But the real problem is more fundamental.

A datetime represents a point in time. It's absolute: it doesn't depend on anything. On the contrary, a date is a calendaring concept. It's a period of time whose bounds depend on the time zone in which the date is considered. As you can see, these two concepts are fundamentally different, and converting a datetime to a date isn't a deterministic operation.

What does this mean in practice?

Generally, you should avoid converting a `datetime` to `date`. For instance, you can use the *`date`* template filter to only show the date part of a datetime. This filter will convert the datetime into the current time zone before formatting it, ensuring the results appear correctly.

If you really need to do the conversion yourself, you must ensure the datetime is converted to the appropriate time zone first. Usually, this will be the current timezone:

```
>>> from django.utils import timezone
>>> timezone.activate(zoneinfo.ZoneInfo("Asia/Singapore"))
# For this example, we set the time zone to Singapore, but here's how
# you would obtain the current time zone in the general case.
>>> current_tz = timezone.get_current_timezone()
>>> local = paris.astimezone(current_tz)
>>> local
datetime.datetime(2012, 3, 3, 8, 30, tzinfo=zoneinfo.ZoneInfo(key='Asia/Singapore'))
>>> local.date()
datetime.date(2012, 3, 3)
```

4. I get an error "`Are time zone definitions for your database installed?`"

   If you are using MySQL, see the Time zone definitions section of the MySQL notes for instructions on loading time zone definitions.

Usage

1. I have a string `"2012-02-21 10:28:45"` and I know it's in the `"Europe/Helsinki"` time zone. How do I turn that into an aware datetime?

   Here you need to create the required `ZoneInfo` instance and attach it to the naïve datetime:

   ```
   >>> import zoneinfo
   >>> from django.utils.dateparse import parse_datetime
   >>> naive = parse_datetime("2012-02-21 10:28:45")
   >>> naive.replace(tzinfo=zoneinfo.ZoneInfo("Europe/Helsinki"))
   datetime.datetime(2012, 2, 21, 10, 28, 45, tzinfo=zoneinfo.ZoneInfo(key='Europe/
   ↪Helsinki'))
   ```

2. How can I obtain the local time in the current time zone?

   Well, the first question is, do you really need to?

   You should only use local time when you're interacting with humans, and the template layer provides filters and tags to convert datetimes to the time zone of your choice.

   Furthermore, Python knows how to compare aware datetimes, taking into account UTC offsets when necessary. It's much easier (and possibly faster) to write all your model and view code in UTC. So, in most circumstances, the datetime in UTC returned by *`django.utils.timezone.now()`* will be sufficient.

   For the sake of completeness, though, if you really want the local time in the current time zone, here's how you can obtain it:

   ```
   >>> from django.utils import timezone
   >>> timezone.localtime(timezone.now())
   datetime.datetime(2012, 3, 3, 20, 10, 53, 873365, tzinfo=zoneinfo.ZoneInfo(key=
   ↪'Europe/Paris'))
   ```

   In this example, the current time zone is `"Europe/Paris"`.

3. How can I see all available time zones?

   `zoneinfo.available_timezones()` provides the set of all valid keys for IANA time zones available to your system. See the docs for usage considerations.

3.16.4 Overview

The goal of internationalization and localization is to allow a single web application to offer its content in languages and formats tailored to the audience.

Django has full support for translation of text, formatting of dates, times and numbers, and time zones.

Essentially, Django does two things:

- It allows developers and template authors to specify which parts of their apps should be translated or formatted for local languages and cultures.
- It uses these hooks to localize web apps for particular users according to their preferences.

Translation depends on the target language, and formatting usually depends on the target country. This information is provided by browsers in the `Accept-Language` header. However, the time zone isn't readily available.

3.16.5 Definitions

The words "internationalization" and "localization" often cause confusion; here's a simplified definition:

internationalization
: Preparing the software for localization. Usually done by developers.

localization
: Writing the translations and local formats. Usually done by translators.

More details can be found in the W3C Web Internationalization FAQ, the Wikipedia article or the GNU gettext documentation.

> **Warning**
>
> Translation is controlled by the *`USE_I18N`* setting. However, it involves internationalization and localization. The name of the setting is an unfortunate result of Django's history.

Here are some other terms that will help us to handle a common language:

locale name
: A locale name, either a language specification of the form `ll` or a combined language and country specification of the form `ll_CC`. Examples: `it`, `de_AT`, `es`, `pt_BR`, `sr_Latn`. The language part is always in lowercase. The country part is in titlecase if it has more than 2 characters, otherwise it's in uppercase. The separator is an underscore.

language code
: Represents the name of a language. Browsers send the names of the languages they accept in the `Accept-Language` HTTP header using this format. Examples: `it`, `de-at`, `es`, `pt-br`. Language codes are generally represented in lowercase, but the HTTP `Accept-Language` header is case-insensitive. The separator is a dash.

message file
: A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a `.po` file extension.

translation string
: A literal that can be translated.

format file
: A format file is a Python module that defines the data formats for a given locale.

3.17 Logging

See also

- How to configure and use logging
- Django logging reference

Python programmers will often use `print()` in their code as a quick and convenient debugging tool. Using the logging framework is only a little more effort than that, but it's much more elegant and flexible. As well as being useful for debugging, logging can also provide you with more - and better structured - information about the state and health of your application.

3.17.1 Overview

Django uses and extends Python's builtin `logging` module to perform system logging. This module is discussed in detail in Python's own documentation; this section provides a quick overview.

The cast of players

A Python logging configuration consists of four parts:

- Loggers
- Handlers
- Filters
- Formatters

Loggers

A logger is the entry point into the logging system. Each logger is a named bucket to which messages can be written for processing.

A logger is configured to have a log level. This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:

- `DEBUG`: Low level system information for debugging purposes
- `INFO`: General system information

- `WARNING`: Information describing a minor problem that has occurred.
- `ERROR`: Information describing a major problem that has occurred.
- `CRITICAL`: Information describing a critical problem that has occurred.

Each message that is written to the logger is a Log Record. Each log record also has a log level indicating the severity of that specific message. A log record can also contain useful metadata that describes the event that is being logged. This can include details such as a stack trace or an error code.

When a message is given to the logger, the log level of the message is compared to the log level of the logger. If the log level of the message meets or exceeds the log level of the logger itself, the message will undergo further processing. If it doesn't, the message will be ignored.

Once a logger has determined that a message needs to be processed, it is passed to a Handler.

Handlers

The handler is the engine that determines what happens to each message in a logger. It describes a particular logging behavior, such as writing a message to the screen, to a file, or to a network socket.

Like loggers, handlers also have a log level. If the log level of a log record doesn't meet or exceed the level of the handler, the handler will ignore the message.

A logger can have multiple handlers, and each handler can have a different log level. In this way, it is possible to provide different forms of notification depending on the importance of a message. For example, you could install one handler that forwards `ERROR` and `CRITICAL` messages to a paging service, while a second handler logs all messages (including `ERROR` and `CRITICAL` messages) to a file for later analysis.

Filters

A filter is used to provide additional control over which log records are passed from logger to handler.

By default, any log message that meets log level requirements will be handled. However, by installing a filter, you can place additional criteria on the logging process. For example, you could install a filter that only allows `ERROR` messages from a particular source to be emitted.

Filters can also be used to modify the logging record prior to being emitted. For example, you could write a filter that downgrades `ERROR` log records to `WARNING` records if a particular set of criteria are met.

Filters can be installed on loggers or on handlers; multiple filters can be used in a chain to perform multiple filtering actions.

Formatters

Ultimately, a log record needs to be rendered as text. Formatters describe the exact format of that text. A formatter usually consists of a Python formatting string containing LogRecord attributes; however, you can also write custom formatters to implement specific formatting behavior.

3.17.2 Security implications

The logging system handles potentially sensitive information. For example, the log record may contain information about a web request or a stack trace, while some of the data you collect in your own loggers may also have security implications. You need to be sure you know:

- what information is collected
- where it will subsequently be stored
- how it will be transferred
- who might have access to it.

To help control the collection of sensitive information, you can explicitly designate certain sensitive information to be filtered out of error reports – read more about how to filter error reports.

`AdminEmailHandler`

The built-in *`AdminEmailHandler`* deserves a mention in the context of security. If its `include_html` option is enabled, the email message it sends will contain a full traceback, with names and values of local variables at each level of the stack, plus the values of your Django settings (in other words, the same level of detail that is exposed in a web page when *`DEBUG`* is `True`).

It's generally not considered a good idea to send such potentially sensitive information over email. Consider instead using one of the many third-party services to which detailed logs can be sent to get the best of multiple worlds – the rich information of full tracebacks, clear management of who is notified and has access to the information, and so on.

3.17.3 Configuring logging

Python's logging library provides several techniques to configure logging, ranging from a programmatic interface to configuration files. By default, Django uses the dictConfig format.

In order to configure logging, you use *`LOGGING`* to define a dictionary of logging settings. These settings describe the loggers, handlers, filters and formatters that you want in your logging setup, and the log levels and other properties that you want those components to have.

By default, the *`LOGGING`* setting is merged with Django's default logging configuration using the following scheme.

If the `disable_existing_loggers` key in the *`LOGGING`* dictConfig is set to `True` (which is the `dictConfig` default if the key is missing) then all loggers from the default configuration will be disabled. Disabled loggers are not the same as removed; the logger will still exist, but will silently discard anything logged to it, not even propagating entries to a parent logger. Thus you should be very careful using `'disable_existing_loggers': True`; it's probably not what you want. Instead, you can set `disable_existing_loggers` to `False` and redefine some or all of the default loggers; or you can set *`LOGGING_CONFIG`* to `None` and handle logging config yourself.

Logging is configured as part of the general Django `setup()` function. Therefore, you can be certain that loggers are always ready for use in your project code.

Examples

The full documentation for dictConfig format is the best source of information about logging configuration dictionaries. However, to give you a taste of what is possible, here are several examples.

To begin, here's a small configuration that will allow you to output all log messages to the console:

Listing 31: `settings.py`

```
import os

LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
        },
    },
    "root": {
        "handlers": ["console"],
        "level": "WARNING",
    },
}
```

This configures the parent `root` logger to send messages with the `WARNING` level and higher to the console handler. By adjusting the level to `INFO` or `DEBUG` you can display more messages. This may be useful during development.

Next we can add more fine-grained logging. Here's an example of how to make the logging system print more messages from just the django named logger:

Listing 32: `settings.py`

```
import os

LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {
```

(continues on next page)

(continued from previous page)

```
            "class": "logging.StreamHandler",
        },
    },
    "root": {
        "handlers": ["console"],
        "level": "WARNING",
    },
    "loggers": {
        "django": {
            "handlers": ["console"],
            "level": os.getenv("DJANGO_LOG_LEVEL", "INFO"),
            "propagate": False,
        },
    },
}
```

By default, this config sends messages from the `django` logger of level `INFO` or higher to the console. This is the same level as Django's default logging config, except that the default config only displays log records when `DEBUG=True`. Django does not log many such `INFO` level messages. With this config, however, you can also set the environment variable `DJANGO_LOG_LEVEL=DEBUG` to see all of Django's debug logging which is very verbose as it includes all database queries.

You don't have to log to the console. Here's a configuration which writes all logging from the django named logger to a local file:

Listing 33: `settings.py`

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "file": {
            "level": "DEBUG",
            "class": "logging.FileHandler",
            "filename": "/path/to/django/debug.log",
        },
    },
    "loggers": {
        "django": {
            "handlers": ["file"],
            "level": "DEBUG",
```

(continues on next page)

(continued from previous page)

```
            "propagate": True,
        },
    },
}
```

If you use this example, be sure to change the `'filename'` path to a location that's writable by the user that's running the Django application.

Finally, here's an example of a fairly complex logging setup:

Listing 34: `settings.py`

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "verbose": {
            "format": "{levelname} {asctime} {module} {process:d} {thread:d} {message}",
            "style": "{",
        },
        "simple": {
            "format": "{levelname} {message}",
            "style": "{",
        },
    },
    "filters": {
        "special": {
            "()": "project.logging.SpecialFilter",
            "foo": "bar",
        },
        "require_debug_true": {
            "()": "django.utils.log.RequireDebugTrue",
        },
    },
    "handlers": {
        "console": {
            "level": "INFO",
            "filters": ["require_debug_true"],
            "class": "logging.StreamHandler",
            "formatter": "simple",
        },
```

(continues on next page)

(continued from previous page)

```
        "mail_admins": {
            "level": "ERROR",
            "class": "django.utils.log.AdminEmailHandler",
            "filters": ["special"],
        },
    },
    "loggers": {
        "django": {
            "handlers": ["console"],
            "propagate": True,
        },
        "django.request": {
            "handlers": ["mail_admins"],
            "level": "ERROR",
            "propagate": False,
        },
        "myproject.custom": {
            "handlers": ["console", "mail_admins"],
            "level": "INFO",
            "filters": ["special"],
        },
    },
}
```

This logging configuration does the following things:

- Identifies the configuration as being in 'dictConfig version 1' format. At present, this is the only dictConfig format version.
- Defines two formatters:
  - `simple`, that outputs the log level name (e.g., `DEBUG`) and the log message.

    The `format` string is a normal Python formatting string describing the details that are to be output on each logging line. The full list of detail that can be output can be found in Formatter Objects.
  - `verbose`, that outputs the log level name, the log message, plus the time, process, thread and module that generate the log message.
- Defines two filters:
  - `project.logging.SpecialFilter`, using the alias `special`. If this filter required additional arguments, they can be provided as additional keys in the filter configuration dictionary. In this case, the argument `foo` will be given a value of `bar` when instantiating `SpecialFilter`.
  - `django.utils.log.RequireDebugTrue`, which passes on records when *`DEBUG`* is `True`.

- Defines two handlers:
  - `console`, a `StreamHandler`, which prints any `INFO` (or higher) message to `sys.stderr`. This handler uses the `simple` output format.
  - `mail_admins`, an *`AdminEmailHandler`*, which emails any `ERROR` (or higher) message to the site *`ADMINS`*. This handler uses the `special` filter.
- Configures three loggers:
  - `django`, which passes all messages to the `console` handler.
  - `django.request`, which passes all `ERROR` messages to the `mail_admins` handler. In addition, this logger is marked to not propagate messages. This means that log messages written to `django.request` will not be handled by the `django` logger.
  - `myproject.custom`, which passes all messages at `INFO` or higher that also pass the `special` filter to two handlers – the `console`, and `mail_admins`. This means that all `INFO` level messages (or higher) will be printed to the console; `ERROR` and `CRITICAL` messages will also be output via email.

Custom logging configuration

If you don't want to use Python's dictConfig format to configure your logger, you can specify your own configuration scheme.

The *`LOGGING_CONFIG`* setting defines the callable that will be used to configure Django's loggers. By default, it points at Python's `logging.config.dictConfig()` function. However, if you want to use a different configuration process, you can use any other callable that takes a single argument. The contents of *`LOGGING`* will be provided as the value of that argument when logging is configured.

Disabling logging configuration

If you don't want to configure logging at all (or you want to manually configure logging using your own approach), you can set *`LOGGING_CONFIG`* to `None`. This will disable the configuration process for Django's default logging.

Setting *`LOGGING_CONFIG`* to `None` only means that the automatic configuration process is disabled, not logging itself. If you disable the configuration process, Django will still make logging calls, falling back to whatever default logging behavior is defined.

Here's an example that disables Django's logging configuration and then manually configures logging:

Listing 35: `settings.py`

```
LOGGING_CONFIG = None

import logging.config

logging.config.dictConfig(...)
```

Note that the default configuration process only calls *LOGGING_CONFIG* once settings are fully-loaded. In contrast, manually configuring the logging in your settings file will load your logging config immediately. As such, your logging config must appear after any settings on which it depends.

3.18 Pagination

Django provides high-level and low-level ways to help you manage paginated data – that is, data that's split across several pages, with "Previous/Next" links.

3.18.1 The `Paginator` class

Under the hood, all methods of pagination use the *Paginator* class. It does all the heavy lifting of actually splitting a `QuerySet` into *Page* objects.

3.18.2 Example

Give *Paginator* a list of objects, plus the number of items you'd like to have on each page, and it gives you methods for accessing the items for each page:

```
>>> from django.core.paginator import Paginator
>>> objects = ["john", "paul", "george", "ringo"]
>>> p = Paginator(objects, 2)

>>> p.count
4
>>> p.num_pages
2
>>> type(p.page_range)
<class 'range_iterator'>
>>> p.page_range
range(1, 3)

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
```

(continues on next page)

(continued from previous page)

```
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
Traceback (most recent call last):
...
EmptyPage: That page contains no results
>>> page2.previous_page_number()
1
>>> page2.start_index()  # The 1-based index of the first item on this page
3
>>> page2.end_index()  # The 1-based index of the last item on this page
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

Note

Note that you can give `Paginator` a list/tuple, a Django `QuerySet`, or any other object with a `count()` or `__len__()` method. When determining the number of objects contained in the passed object, `Paginator` will first try calling `count()`, then fallback to using `len()` if the passed object has no `count()` method. This allows objects such as Django's `QuerySet` to use a more efficient `count()` method when available.

3.18.3 Paginating a `ListView`

`django.views.generic.list.ListView` provides a builtin way to paginate the displayed list. You can do this by adding a *`paginate_by`* attribute to your view class, for example:

```
from django.views.generic import ListView

from myapp.models import Contact
```

(continues on next page)

(continued from previous page)

```
class ContactListView(ListView):
    paginate_by = 2
    model = Contact
```

This limits the number of objects per page and adds a `paginator` and `page_obj` to the `context`. To allow your users to navigate between pages, add links to the next and previous page, in your template like this:

```
{% for contact in page_obj %}
    {# Each "contact" is a Contact model object. #}
    {{ contact.full_name|upper }}<br>
    ...
{% endfor %}

<div class="pagination">
    <span class="step-links">
        {% if page_obj.has_previous %}
            <a href="?page=1">&laquo; first</a>
            <a href="?page={{ page_obj.previous_page_number }}">previous</a>
        {% endif %}

        <span class="current">
            Page {{ page_obj.number }} of {{ page_obj.paginator.num_pages }}.
        </span>

        {% if page_obj.has_next %}
            <a href="?page={{ page_obj.next_page_number }}">next</a>
            <a href="?page={{ page_obj.paginator.num_pages }}">last &raquo;</a>
        {% endif %}
    </span>
</div>
```

3.18.4 Using `Paginator` in a view function

Here's an example using *`Paginator`* in a view function to paginate a queryset:

```
from django.core.paginator import Paginator
from django.shortcuts import render

from myapp.models import Contact
```

(continues on next page)

(continued from previous page)

```
def listing(request):
    contact_list = Contact.objects.all()
    paginator = Paginator(contact_list, 25)  # Show 25 contacts per page.

    page_number = request.GET.get("page")
    page_obj = paginator.get_page(page_number)
    return render(request, "list.html", {"page_obj": page_obj})
```

In the template `list.html`, you can include navigation between pages in the same way as in the template for the `ListView` above.

3.19 Security in Django

This document is an overview of Django's security features. It includes advice on securing a Django-powered site.

3.19.1 Always sanitize user input

The golden rule of web application security is to never trust user-controlled data. Hence, all user input should be sanitized before being used in your application. See the forms documentation for details on validating user inputs in Django.

3.19.2 Cross site scripting (XSS) protection

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

```
<style class={{ var }}>...</style>
```

If `var` is set to `'class1 onmouseover=javascript:func()'`, this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML. (Quoting the attribute value would fix this case.)

It is also important to be particularly careful when using `is_safe` with custom template tags, the *safe* template tag, *mark_safe*, and when autoescape is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

3.19.3 Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.

Django has built-in protection against most types of CSRF attacks, providing you have enabled and used it where appropriate. However, as with any mitigation technique, there are limitations. For example, it is possible to disable the CSRF module globally or for particular views. You should only do this if you know what you are doing. There are other limitations if your site has subdomains that are outside of your control.

CSRF protection works by checking for a secret in each POST request. This ensures that a malicious user cannot "replay" a form POST to your website and have another logged in user unwittingly submit that form. The malicious user would have to know the secret, which is user specific (using a cookie).

When deployed with HTTPS, `CsrfViewMiddleware` will check that the HTTP referer header is set to a URL on the same origin (including subdomain and port). Because HTTPS provides additional security, it is imperative to ensure connections use HTTPS where it is available by forwarding insecure connection requests and using HSTS for supported browsers.

Be very careful with marking views with the `csrf_exempt` decorator unless it is absolutely necessary.

3.19.4 SQL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

Django's querysets are protected from SQL injection since their queries are constructed using query parameterization. A query's SQL code is defined separately from the query's parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.

Django also gives developers power to write raw queries or execute custom sql. These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using *extra()* and *RawSQL*.

3.19.5 Clickjacking protection

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site.

Django contains clickjacking protection in the form of the *X-Frame-Options middleware* which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.

The middleware is strongly recommended for any site that does not need to have its pages wrapped in a frame by third party sites, or only needs to allow that for a small section of the site.

3.19.6 SSL/HTTPS

It is always better for security to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information transferred between client and server, and in some cases – active network attackers – to alter data that is sent in either direction.

If you want the protection that HTTPS provides, and have enabled it on your server, there are some additional steps you may need:

- If necessary, set *SECURE_PROXY_SSL_HEADER*, ensuring that you have understood the warnings there thoroughly. Failure to do this can result in CSRF vulnerabilities, and failure to do it correctly can also be dangerous!
- Set *SECURE_SSL_REDIRECT* to `True`, so that requests over HTTP are redirected to HTTPS.

  Please note the caveats under *SECURE_PROXY_SSL_HEADER*. For the case of a reverse proxy, it may be easier or more secure to configure the main web server to do the redirect to HTTPS.
- Use 'secure' cookies.

  If a browser connects initially via HTTP, which is the default for most browsers, it is possible for existing cookies to be leaked. For this reason, you should set your *SESSION_COOKIE_SECURE* and *CSRF_COOKIE_SECURE* settings to `True`. This instructs the browser to only send these cookies over HTTPS connections. Note that this will mean that sessions will not work over HTTP, and the CSRF protection will prevent any POST data being accepted over HTTP (which will be fine if you are redirecting all HTTP traffic to HTTPS).
- Use HTTP Strict Transport Security (HSTS)

  HSTS is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS. Combined with redirecting requests over HTTP to HTTPS, this will ensure that connections always enjoy the added security of SSL provided one successful connection has occurred. HSTS may either be configured with *SECURE_HSTS_SECONDS*, *SECURE_HSTS_INCLUDE_SUBDOMAINS*, and *SECURE_HSTS_PRELOAD*, or on the web server.

3.19.7 Host header validation

Django uses the `Host` header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, a fake `Host` value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails.

Because even seemingly-secure web server configurations are susceptible to fake `Host` headers, Django validates `Host` headers against the *`ALLOWED_HOSTS`* setting in the *`django.http.HttpRequest.get_host()`* method.

This validation only applies via *`get_host()`*; if your code accesses the `Host` header directly from `request.META` you are bypassing this security protection.

For more details see the full *`ALLOWED_HOSTS`* documentation.

> **Warning**
>
> Previous versions of this document recommended configuring your web server to ensure it validates incoming HTTP `Host` headers. While this is still recommended, in many common web servers a configuration that seems to validate the `Host` header may not in fact do so. For instance, even if Apache is configured such that your Django site is served from a non-default virtual host with the `ServerName` set, it is still possible for an HTTP request to match this virtual host and supply a fake `Host` header. Thus, Django now requires that you set *`ALLOWED_HOSTS`* explicitly rather than relying on web server configuration.

Additionally, Django requires you to explicitly enable support for the `X-Forwarded-Host` header (via the *`USE_X_FORWARDED_HOST`* setting) if your configuration requires it.

3.19.8 Referrer policy

Browsers use the `Referer` header as a way to send information to a site about how users got there. By setting a Referrer Policy you can help to protect the privacy of your users, restricting under which circumstances the `Referer` header is set. See the referrer policy section of the security middleware reference for details.

3.19.9 Cross-origin opener policy

The cross-origin opener policy (COOP) header allows browsers to isolate a top-level window from other documents by putting them in a different context group so that they cannot directly interact with the top-level window. If a document protected by COOP opens a cross-origin popup window, the popup's `window.opener` property will be `null`. COOP protects against cross-origin attacks. See the cross-origin opener policy section of the security middleware reference for details.

3.19.10 Session security

Similar to the CSRF limitations requiring a site to be deployed such that untrusted users don't have access to any subdomains, *django.contrib.sessions* also has limitations. See the session topic guide section on security for details.

3.19.11 User-uploaded content

> **Note**
>
> Consider serving static files from a cloud service or CDN to avoid some of these issues.

- If your site accepts file uploads, it is strongly advised that you limit these uploads in your web server configuration to a reasonable size in order to prevent denial of service (DOS) attacks. In Apache, this can be easily set using the LimitRequestBody directive.
- If you are serving your own static files, be sure that handlers like Apache's `mod_php`, which would execute static files as code, are disabled. You don't want users to be able to execute arbitrary code by uploading and requesting a specially crafted file.
- Django's media upload handling poses some vulnerabilities when that media is served in ways that do not follow security best practices. Specifically, an HTML file can be uploaded as an image if that file contains a valid PNG header followed by malicious HTML. This file will pass verification of the library that Django uses for *ImageField* image processing (Pillow). When this file is subsequently displayed to a user, it may be displayed as HTML depending on the type and configuration of your web server.

  No bulletproof technical solution exists at the framework level to safely validate all user uploaded file content, however, there are some other steps you can take to mitigate these attacks:

  1. One class of attacks can be prevented by always serving user uploaded content from a distinct top-level or second-level domain. This prevents any exploit blocked by same-origin policy protections such as cross site scripting. For example, if your site runs on `example.com`, you would want to serve uploaded content (the *MEDIA_URL* setting) from something like `usercontent-example.com`. It's not sufficient to serve content from a subdomain like `usercontent.example.com`.
  2. Beyond this, applications may choose to define a list of allowable file extensions for user uploaded files and configure the web server to only serve such files.

3.19.12 Additional security topics

While Django provides good security protection out of the box, it is still important to properly deploy your application and take advantage of the security protection of the web server, operating system and other components.

- Make sure that your Python code is outside of the web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).

- Take care with any user uploaded files.
- Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or web server module to throttle these requests.
- Keep your *SECRET_KEY*, and *SECRET_KEY_FALLBACKS* if in use, secret.
- It is a good idea to limit the accessibility of your caching system and database using a firewall.
- Take a look at the Open Web Application Security Project (OWASP) Top 10 list which identifies some common vulnerabilities in web applications. While Django has tools to address some of the issues, other issues must be accounted for in the design of your project.
- Mozilla discusses various topics regarding web security. Their pages also include security principles that apply to any system.

3.20 Performance and optimization

This document provides an overview of techniques and tools that can help get your Django code running more efficiently - faster, and using fewer system resources.

3.20.1 Introduction

Generally one's first concern is to write code that works, whose logic functions as required to produce the expected output. Sometimes, however, this will not be enough to make the code work as efficiently as one would like.

In this case, what's needed is something - and in practice, often a collection of things - to improve the code's performance without, or only minimally, affecting its behavior.

3.20.2 General approaches

What are you optimizing *for*?

It's important to have a clear idea what you mean by 'performance'. There is not just one metric of it.

Improved speed might be the most obvious aim for a program, but sometimes other performance improvements might be sought, such as lower memory consumption or fewer demands on the database or network.

Improvements in one area will often bring about improved performance in another, but not always; sometimes one can even be at the expense of another. For example, an improvement in a program's speed might cause it to use more memory. Even worse, it can be self-defeating - if the speed improvement is so memory-hungry that the system starts to run out of memory, you'll have done more harm than good.

There are other trade-offs to bear in mind. Your own time is a valuable resource, more precious than CPU time. Some improvements might be too difficult to be worth implementing, or might affect the portability or maintainability of the code. Not all performance improvements are worth the effort.

So, you need to know what performance improvements you are aiming for, and you also need to know that you have a good reason for aiming in that direction - and for that you need:

Performance benchmarking

It's no good just guessing or assuming where the inefficiencies lie in your code.

Django tools

django-debug-toolbar is a very handy tool that provides insights into what your code is doing and how much time it spends doing it. In particular it can show you all the SQL queries your page is generating, and how long each one has taken.

Third-party panels are also available for the toolbar, that can (for example) report on cache performance and template rendering times.

Third-party services

There are a number of free services that will analyze and report on the performance of your site's pages from the perspective of a remote HTTP client, in effect simulating the experience of an actual user.

These can't report on the internals of your code, but can provide a useful insight into your site's overall performance, including aspects that can't be adequately measured from within Django environment.

There are also several paid-for services that perform a similar analysis, including some that are Django-aware and can integrate with your codebase to profile its performance far more comprehensively.

Get things right from the start

Some work in optimization involves tackling performance shortcomings, but some of the work can be built-in to what you'd do anyway, as part of the good practices you should adopt even before you start thinking about improving performance.

In this respect Python is an excellent language to work with, because solutions that look elegant and feel right usually are the best performing ones. As with most skills, learning what "looks right" takes practice, but one of the most useful guidelines is:

Work at the appropriate level

Django offers many different ways of approaching things, but just because it's possible to do something in a certain way doesn't mean that it's the most appropriate way to do it. For example, you might find that you could calculate the same thing - the number of items in a collection, perhaps - in a `QuerySet`, in Python, or in a template.

However, it will almost always be faster to do this work at lower rather than higher levels. At higher levels the system has to deal with objects through multiple levels of abstraction and layers of machinery.

That is, the database can typically do things faster than Python can, which can do them faster than the template language can:

```
# QuerySet operation on the database
# fast, because that's what databases are good at
my_bicycles.count()

# counting Python objects
# slower, because it requires a database query anyway, and processing
# of the Python objects
len(my_bicycles)
```

```
<!--
Django template filter
slower still, because it will have to count them in Python anyway,
and because of template language overheads
-->
{{ my_bicycles|length }}
```

Generally speaking, the most appropriate level for the job is the lowest-level one that it is comfortable to code for.

> **Note**
>
> The example above is merely illustrative.
>
> Firstly, in a real-life case you need to consider what is happening before and after your count to work out what's an optimal way of doing it in that particular context. The database optimization document describes a case where counting in the template would be better.
>
> Secondly, there are other options to consider: in a real-life case, `{{ my_bicycles.count }}`, which invokes the `QuerySet` `count()` method directly from the template, might be the most appropriate choice.

3.20.3 Caching

Often it is expensive (that is, resource-hungry and slow) to compute a value, so there can be huge benefit in saving the value to a quickly accessible cache, ready for the next time it's required.

It's a sufficiently significant and powerful technique that Django includes a comprehensive caching framework, as well as other smaller pieces of caching functionality.

The caching framework

Django's caching framework offers very significant opportunities for performance gains, by saving dynamic content so that it doesn't need to be calculated for each request.

For convenience, Django offers different levels of cache granularity: you can cache the output of specific views, or only the pieces that are difficult to produce, or even an entire site.

Implementing caching should not be regarded as an alternative to improving code that's performing poorly because it has been written badly. It's one of the final steps toward producing well-performing code, not a shortcut.

`cached_property`

It's common to have to call a class instance's method more than once. If that function is expensive, then doing so can be wasteful.

Using the *`cached_property`* decorator saves the value returned by a property; the next time the function is called on that instance, it will return the saved value rather than re-computing it. Note that this only works on methods that take `self` as their only argument and that it changes the method to a property.

Certain Django components also have their own caching functionality; these are discussed below in the sections related to those components.

3.20.4 Understanding laziness

Laziness is a strategy complementary to caching. Caching avoids recomputation by saving results; laziness delays computation until it's actually required.

Laziness allows us to refer to things before they are instantiated, or even before it's possible to instantiate them. This has numerous uses.

For example, lazy translation can be used before the target language is even known, because it doesn't take place until the translated string is actually required, such as in a rendered template.

Laziness is also a way to save effort by trying to avoid work in the first place. That is, one aspect of laziness is not doing anything until it has to be done, because it may not turn out to be necessary after all. Laziness can therefore have performance implications, and the more expensive the work concerned, the more there is to gain through laziness.

Python provides a number of tools for lazy evaluation, particularly through the generator and generator expression constructs. It's worth reading up on laziness in Python to discover opportunities for making use of lazy patterns in your code.

Laziness in Django

Django is itself quite lazy. A good example of this can be found in the evaluation of a `QuerySet`. QuerySets are lazy. Thus a `QuerySet` can be created, passed around and combined with other `QuerySet` instances, without actually incurring any trips to the database to fetch the items it describes. What gets passed around is the `QuerySet` object, not the collection of items that - eventually - will be required from the database.

On the other hand, certain operations will force the evaluation of a QuerySet. Avoiding the premature evaluation of a `QuerySet` can save making an expensive and unnecessary trip to the database.

Django also offers a *keep_lazy()* decorator. This allows a function that has been called with a lazy argument to behave lazily itself, only being evaluated when it needs to be. Thus the lazy argument - which could be an expensive one - will not be called upon for evaluation until it's strictly required.

3.20.5 Databases

Database optimization

Django's database layer provides various ways to help developers get the best performance from their databases. The database optimization documentation gathers together links to the relevant documentation and adds various tips that outline the steps to take when attempting to optimize your database usage.

Other database-related tips

Enabling Persistent connections can speed up connections to the database accounts for a significant part of the request processing time.

This helps a lot on virtualized hosts with limited network performance, for example.

3.20.6 HTTP performance

Middleware

Django comes with a few helpful pieces of middleware that can help optimize your site's performance. They include:

`ConditionalGetMiddleware`

Adds support for modern browsers to conditionally GET responses based on the `ETag` and `Last-Modified` headers. It also calculates and sets an ETag if needed.

`GZipMiddleware`

Compresses responses for all modern browsers, saving bandwidth and transfer time. Note that GZipMiddleware is currently considered a security risk, and is vulnerable to attacks that nullify the protection provided by TLS/SSL. See the warning in *GZipMiddleware* for more information.

Sessions

Using cached sessions

Using cached sessions may be a way to increase performance by eliminating the need to load session data from a slower storage source like the database and instead storing frequently used session data in memory.

Static files

Static files, which by definition are not dynamic, make an excellent target for optimization gains.

`ManifestStaticFilesStorage`

By taking advantage of web browsers' caching abilities, you can eliminate network hits entirely for a given file after the initial download.

`ManifestStaticFilesStorage` appends a content-dependent tag to the filenames of static files to make it safe for browsers to cache them long-term without missing future changes - when a file changes, so will the tag, so browsers will reload the asset automatically.

"Minification"

Several third-party Django tools and packages provide the ability to "minify" HTML, CSS, and JavaScript. They remove unnecessary whitespace, newlines, and comments, and shorten variable names, and thus reduce the size of the documents that your site publishes.

3.20.7 Template performance

Note that:

- using `{% block %}` is faster than using `{% include %}`
- heavily-fragmented templates, assembled from many small pieces, can affect performance

The cached template loader

Enabling the *`cached template loader`* often improves performance drastically, as it avoids compiling each template every time it needs to be rendered.

3.20.8 Using different versions of available software

It can sometimes be worth checking whether different and better-performing versions of the software that you're using are available.

These techniques are targeted at more advanced users who want to push the boundaries of performance of an already well-optimized Django site.

However, they are not magic solutions to performance problems, and they're unlikely to bring better than marginal gains to sites that don't already do the more basic things the right way.

> **Note**
>
> It's worth repeating: reaching for alternatives to software you're already using is never the first answer to performance problems. When you reach this level of optimization, you need a formal benchmarking solution.

Newer is often - but not always - better

It's fairly rare for a new release of well-maintained software to be less efficient, but the maintainers can't anticipate every possible use-case - so while being aware that newer versions are likely to perform better, don't assume that they always will.

This is true of Django itself. Successive releases have offered a number of improvements across the system, but you should still check the real-world performance of your application, because in some cases you may find that changes mean it performs worse rather than better.

Newer versions of Python, and also of Python packages, will often perform better too - but measure, rather than assume.

> **Note**
>
> Unless you've encountered an unusual performance problem in a particular version, you'll generally find better features, reliability, and security in a new release and that these benefits are far more significant than any performance you might win or lose.

Alternatives to Django's template language

For nearly all cases, Django's built-in template language is perfectly adequate. However, if the bottlenecks in your Django project seem to lie in the template system and you have exhausted other opportunities to remedy this, a third-party alternative may be the answer.

Jinja2 can offer performance improvements, particularly when it comes to speed.

Alternative template systems vary in the extent to which they share Django's templating language.

> **Note**
>
> If you experience performance issues in templates, the first thing to do is to understand exactly why. Using an alternative template system may prove faster, but the same gains may also be available without going to that trouble - for example, expensive processing and logic in your templates could be done more efficiently in your views.

Alternative software implementations

It may be worth checking whether Python software you're using has been provided in a different implementation that can execute the same code faster.

However: most performance problems in well-written Django sites aren't at the Python execution level, but rather in inefficient database querying, caching, and templates. If you're relying on poorly-written Python code, your performance problems are unlikely to be solved by having it execute faster.

Using an alternative implementation may introduce compatibility, deployment, portability, or maintenance issues. It goes without saying that before adopting a non-standard implementation you should ensure it provides sufficient performance gains for your application to outweigh the potential risks.

With these caveats in mind, you should be aware of:

PyPy

PyPy is an implementation of Python in Python itself (the 'standard' Python implementation is in C). PyPy can offer substantial performance gains, typically for heavyweight applications.

A key aim of the PyPy project is compatibility with existing Python APIs and libraries. Django is compatible, but you will need to check the compatibility of other libraries you rely on.

C implementations of Python libraries

Some Python libraries are also implemented in C, and can be much faster. They aim to offer the same APIs. Note that compatibility issues and behavior differences are not unknown (and not always immediately evident).

3.21 Serializing Django objects

Django's serialization framework provides a mechanism for "translating" Django models into other formats. Usually these other formats will be text-based and used for sending Django data over a wire, but it's possible for a serializer to handle any format (text-based or not).

> See also
>
> If you just want to get some data from your tables into a serialized form, you could use the *dumpdata* management command.

3.21.1 Serializing data

At the highest level, you can serialize data like this:

```
from django.core import serializers

data = serializers.serialize("xml", SomeModel.objects.all())
```

The arguments to the `serialize` function are the format to serialize the data to (see Serialization formats) and a *QuerySet* to serialize. (Actually, the second argument can be any iterator that yields Django model instances, but it'll almost always be a QuerySet).

`django.core.serializers.get_serializer`(format)

You can also use a serializer object directly:

```
XMLSerializer = serializers.get_serializer("xml")
xml_serializer = XMLSerializer()
xml_serializer.serialize(queryset)
data = xml_serializer.getvalue()
```

This is useful if you want to serialize data directly to a file-like object (which includes an *HttpResponse*):

```
with open("file.xml", "w") as out:
    xml_serializer.serialize(SomeModel.objects.all(), stream=out)
```

> **Note**
>
> Calling *get_serializer()* with an unknown format will raise a `django.core.serializers.SerializerDoesNotExist` exception.

Subset of fields

If you only want a subset of fields to be serialized, you can specify a `fields` argument to the serializer:

```
from django.core import serializers

data = serializers.serialize("xml", SomeModel.objects.all(), fields=["name", "size"])
```

In this example, only the `name` and `size` attributes of each model will be serialized. The primary key is always serialized as the `pk` element in the resulting output; it never appears in the `fields` part.

> **Note**

Depending on your model, you may find that it is not possible to deserialize a model that only serializes a subset of its fields. If a serialized object doesn't specify all the fields that are required by a model, the deserializer will not be able to save deserialized instances.

Inherited models

If you have a model that is defined using an abstract base class, you don't have to do anything special to serialize that model. Call the serializer on the object (or objects) that you want to serialize, and the output will be a complete representation of the serialized object.

However, if you have a model that uses multi-table inheritance, you also need to serialize all of the base classes for the model. This is because only the fields that are locally defined on the model will be serialized. For example, consider the following models:

```
class Place(models.Model):
    name = models.CharField(max_length=50)


class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
```

If you only serialize the Restaurant model:

```
data = serializers.serialize("xml", Restaurant.objects.all())
```

the fields on the serialized output will only contain the `serves_hot_dogs` attribute. The `name` attribute of the base class will be ignored.

In order to fully serialize your `Restaurant` instances, you will need to serialize the `Place` models as well:

```
all_objects = [*Restaurant.objects.all(), *Place.objects.all()]
data = serializers.serialize("xml", all_objects)
```

3.21.2 Deserializing data

Deserializing data is very similar to serializing it:

```
for obj in serializers.deserialize("xml", data):
    do_something_with(obj)
```

As you can see, the `deserialize` function takes the same format argument as `serialize`, a string or stream of data, and returns an iterator.

However, here it gets slightly complicated. The objects returned by the `deserialize` iterator aren't regular Django objects. Instead, they are special `DeserializedObject` instances that wrap a created – but unsaved

– object and any associated relationship data.

Calling `DeserializedObject.save()` saves the object to the database.

> **Note**
>
> If the `pk` attribute in the serialized data doesn't exist or is null, a new instance will be saved to the database.

This ensures that deserializing is a non-destructive operation even if the data in your serialized representation doesn't match what's currently in the database. Usually, working with these `DeserializedObject` instances looks something like:

```
for deserialized_object in serializers.deserialize("xml", data):
    if object_should_be_saved(deserialized_object):
        deserialized_object.save()
```

In other words, the usual use is to examine the deserialized objects to make sure that they are "appropriate" for saving before doing so. If you trust your data source you can instead save the object directly and move on.

The Django object itself can be inspected as `deserialized_object.object`. If fields in the serialized data do not exist on a model, a `DeserializationError` will be raised unless the `ignorenonexistent` argument is passed in as `True`:

```
serializers.deserialize("xml", data, ignorenonexistent=True)
```

3.21.3 Serialization formats

Django supports a number of serialization formats, some of which require you to install third-party Python modules:

Identifier	Information
`xml`	Serializes to and from a simple XML dialect.
`json`	Serializes to and from JSON.
`jsonl`	Serializes to and from JSONL.
`yaml`	Serializes to YAML (YAML Ain't a Markup Language). This serializer is only available if PyYAML is installed.

XML

The basic XML serialization format looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="123" model="sessions.session">
        <field type="DateTimeField" name="expire_date">2013-01-16T08:16:59.844560+00:00</
↪field>
        <!-- ... -->
    </object>
</django-objects>
```

The whole collection of objects that is either serialized or deserialized is represented by a `<django-objects>`-tag which contains multiple `<object>`-elements. Each such object has two attributes: "pk" and "model", the latter being represented by the name of the app ("sessions") and the lowercase name of the model ("session") separated by a dot.

Each field of the object is serialized as a `<field>`-element sporting the fields "type" and "name". The text content of the element represents the value that should be stored.

Foreign keys and other relational fields are treated a little bit differently:

```
<object pk="27" model="auth.permission">
    <!-- ... -->
    <field to="contenttypes.contenttype" name="content_type" rel="ManyToOneRel">9</field>
    <!-- ... -->
</object>
```

In this example we specify that the `auth.Permission` object with the PK 27 has a foreign key to the `contenttypes.ContentType` instance with the PK 9.

ManyToMany-relations are exported for the model that binds them. For instance, the `auth.User` model has such a relation to the `auth.Permission` model:

```
<object pk="1" model="auth.user">
    <!-- ... -->
    <field to="auth.permission" name="user_permissions" rel="ManyToManyRel">
        <object pk="46"></object>
        <object pk="47"></object>
    </field>
</object>
```

This example links the given user with the permission models with PKs 46 and 47.

> **Control characters**
>
> If the content to be serialized contains control characters that are not accepted in the XML 1.0 standard, the serialization will fail with a `ValueError` exception. Read also the W3C's explanation of HTML, XHTML, XML and Control Codes.

JSON

When staying with the same example data as before it would be serialized as JSON in the following way:

```
[
    {
        "pk": "4b678b301dfd8a4e0dad910de3ae245b",
        "model": "sessions.session",
        "fields": {
            "expire_date": "2013-01-16T08:16:59.844Z",
            # ...
        },
    }
]
```

The formatting here is a bit simpler than with XML. The whole collection is just represented as an array and the objects are represented by JSON objects with three properties: "pk", "model" and "fields". "fields" is again an object containing each field's name and value as property and property-value respectively.

Foreign keys have the PK of the linked object as property value. ManyToMany-relations are serialized for the model that defines them and are represented as a list of PKs.

Be aware that not all Django output can be passed unmodified to `json`. For example, if you have some custom type in an object to be serialized, you'll have to write a custom `json` encoder for it. Something like this will work:

```
from django.core.serializers.json import DjangoJSONEncoder


class LazyEncoder(DjangoJSONEncoder):
    def default(self, obj):
        if isinstance(obj, YourCustomType):
            return str(obj)
        return super().default(obj)
```

You can then pass `cls=LazyEncoder` to the `serializers.serialize()` function:

```
from django.core.serializers import serialize

serialize("json", SomeModel.objects.all(), cls=LazyEncoder)
```

Also note that GeoDjango provides a customized GeoJSON serializer.

DjangoJSONEncoder

class django.core.serializers.json.**DjangoJSONEncoder**

The JSON serializer uses `DjangoJSONEncoder` for encoding. A subclass of `JSONEncoder`, it handles these additional types:

datetime
: A string of the form `YYYY-MM-DDTHH:mm:ss.sssZ` or `YYYY-MM-DDTHH:mm:ss.sss+HH:MM` as defined in ECMA-262.

date
: A string of the form `YYYY-MM-DD` as defined in ECMA-262.

time
: A string of the form `HH:MM:ss.sss` as defined in ECMA-262.

timedelta
: A string representing a duration as defined in ISO-8601. For example, `timedelta(days=1, hours=2, seconds=3.4)` is represented as `'P1DT02H00M03.400000S'`.

`Decimal`, `Promise` (`django.utils.functional.lazy()` objects), `UUID`
: A string representation of the object.

JSONL

JSONL stands for JSON Lines. With this format, objects are separated by new lines, and each line contains a valid JSON object. JSONL serialized data looks like this:

```
{"pk": "4b678b301dfd8a4e0dad910de3ae245b", "model": "sessions.session", "fields": {...}}
{"pk": "88bea72c02274f3c9bf1cb2bb8cee4fc", "model": "sessions.session", "fields": {...}}
{"pk": "9cf0e26691b64147a67e2a9f06ad7a53", "model": "sessions.session", "fields": {...}}
```

JSONL can be useful for populating large databases, since the data can be processed line by line, rather than being loaded into memory all at once.

YAML

YAML serialization looks quite similar to JSON. The object list is serialized as a sequence mappings with the keys "pk", "model" and "fields". Each field is again a mapping with the key being name of the field and the value the value:

```
- model: sessions.session
  pk: 4b678b301dfd8a4e0dad910de3ae245b
  fields:
    expire_date: 2013-01-16 08:16:59.844560+00:00
```

Referential fields are again represented by the PK or sequence of PKs.

Custom serialization formats

In addition to the default formats, you can create a custom serialization format.

For example, let's consider a csv serializer and deserializer. First, define a `Serializer` and a `Deserializer` class. These can override existing serialization format classes:

Listing 36: `path/to/custom_csv_serializer.py`

```
import csv

from django.apps import apps
from django.core import serializers
from django.core.serializers.base import DeserializationError


class Serializer(serializers.python.Serializer):
    def get_dump_object(self, obj):
        dumped_object = super().get_dump_object(obj)
        row = [dumped_object["model"], str(dumped_object["pk"])]
        row += [str(value) for value in dumped_object["fields"].values()]
        return ",".join(row), dumped_object["model"]

    def end_object(self, obj):
        dumped_object_str, model = self.get_dump_object(obj)
        if self.first:
            fields = [field.name for field in apps.get_model(model)._meta.fields]
            header = ",".join(fields)
            self.stream.write(f"model,{header}\n")
        self.stream.write(f"{dumped_object_str}\n")
```

(continues on next page)

(continued from previous page)

```
    def getvalue(self):
        return super(serializers.python.Serializer, self).getvalue()


class Deserializer(serializers.python.Deserializer):
    def __init__(self, stream_or_string, **options):
        if isinstance(stream_or_string, bytes):
            stream_or_string = stream_or_string.decode()
        if isinstance(stream_or_string, str):
            stream_or_string = stream_or_string.splitlines()
        try:
            objects = csv.DictReader(stream_or_string)
        except Exception as exc:
            raise DeserializationError() from exc
        super().__init__(objects, **options)

    def _handle_object(self, obj):
        try:
            model_fields = apps.get_model(obj["model"])._meta.fields
            obj["fields"] = {
                field.name: obj[field.name]
                for field in model_fields
                if field.name in obj
            }
            yield from super()._handle_object(obj)
        except (GeneratorExit, DeserializationError):
            raise
        except Exception as exc:
            raise DeserializationError(f"Error deserializing object: {exc}") from exc
```

Then add the module containing the serializer definitions to your *SERIALIZATION_MODULES* setting:

```
SERIALIZATION_MODULES = {
    "csv": "path.to.custom_csv_serializer",
    "json": "django.core.serializers.json",
}
```

A `Deserializer` class definition was added to each of the provided serialization formats.

3.21.4 Natural keys

The default serialization strategy for foreign keys and many-to-many relations is to serialize the value of the primary key(s) of the objects in the relation. This strategy works well for most objects, but it can cause difficulty in some circumstances.

Consider the case of a list of objects that have a foreign key referencing *ContentType*. If you're going to serialize an object that refers to a content type, then you need to have a way to refer to that content type to begin with. Since `ContentType` objects are automatically created by Django during the database synchronization process, the primary key of a given content type isn't easy to predict; it will depend on how and when *migrate* was executed. This is true for all models which automatically generate objects, notably including *Permission*, *Group*, and *User*.

> **Warning**
>
> You should never include automatically generated objects in a fixture or other serialized data. By chance, the primary keys in the fixture may match those in the database and loading the fixture will have no effect. In the more likely case that they don't match, the fixture loading will fail with an *IntegrityError*.

There is also the matter of convenience. An integer id isn't always the most convenient way to refer to an object; sometimes, a more natural reference would be helpful.

It is for these reasons that Django provides natural keys. A natural key is a tuple of values that can be used to uniquely identify an object instance without using the primary key value.

Deserialization of natural keys

Consider the following two models:

```
from django.db import models


class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        constraints = [
            models.UniqueConstraint(
                fields=["first_name", "last_name"],
                name="unique_first_last_name",
```

(continues on next page)

(continued from previous page)

```
            ),
        ]


class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person, on_delete=models.CASCADE)
```

Ordinarily, serialized data for `Book` would use an integer to refer to the author. For example, in JSON, a Book might be serialized as:

```
...
{"pk": 1, "model": "store.book", "fields": {"name": "Mostly Harmless", "author": 42}}
...
```

This isn't a particularly natural way to refer to an author. It requires that you know the primary key value for the author; it also requires that this primary key value is stable and predictable.

However, if we add natural key handling to Person, the fixture becomes much more humane. To add natural key handling, you define a default Manager for Person with a `get_by_natural_key()` method. In the case of a Person, a good natural key might be the pair of first and last name:

```
from django.db import models


class PersonManager(models.Manager):
    def get_by_natural_key(self, first_name, last_name):
        return self.get(first_name=first_name, last_name=last_name)


class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    birthdate = models.DateField()

    objects = PersonManager()

    class Meta:
        constraints = [
            models.UniqueConstraint(
                fields=["first_name", "last_name"],
```

(continues on next page)

(continued from previous page)

```
                name="unique_first_last_name",
            ),
        ]
```

Now books can use that natural key to refer to `Person` objects:

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {"name": "Mostly Harmless", "author": ["Douglas", "Adams"]},
}
...
```

When you try to load this serialized data, Django will use the `get_by_natural_key()` method to resolve `["Douglas", "Adams"]` into the primary key of an actual `Person` object.

> **Note**
>
> Whatever fields you use for a natural key must be able to uniquely identify an object. This will usually mean that your model will have a uniqueness clause (either `unique=True` on a single field, or a `UniqueConstraint` or `unique_together` over multiple fields) for the field or fields in your natural key. However, uniqueness doesn't need to be enforced at the database level. If you are certain that a set of fields will be effectively unique, you can still use those fields as a natural key.

Deserialization of objects with no primary key will always check whether the model's manager has a `get_by_natural_key()` method and if so, use it to populate the deserialized object's primary key.

Serialization of natural keys

So how do you get Django to emit a natural key when serializing an object? Firstly, you need to add another method – this time to the model itself:

```
class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    birthdate = models.DateField()

    objects = PersonManager()

    class Meta:
```

(continues on next page)

(continued from previous page)

```
        constraints = [
            models.UniqueConstraint(
                fields=["first_name", "last_name"],
                name="unique_first_last_name",
            ),
        ]

    def natural_key(self):
        return (self.first_name, self.last_name)
```

That method should always return a natural key tuple – in this example, `(first name, last name)`. Then, when you call `serializers.serialize()`, you provide `use_natural_foreign_keys=True` or `use_natural_primary_keys=True` arguments:

```
>>> serializers.serialize(
...     "json",
...     [book1, book2],
...     indent=2,
...     use_natural_foreign_keys=True,
...     use_natural_primary_keys=True,
... )
```

When `use_natural_foreign_keys=True` is specified, Django will use the `natural_key()` method to serialize any foreign key reference to objects of the type that defines the method.

When `use_natural_primary_keys=True` is specified, Django will not provide the primary key in the serialized data of this object since it can be calculated during deserialization:

```
...
{
    "model": "store.person",
    "fields": {
        "first_name": "Douglas",
        "last_name": "Adams",
        "birth_date": "1952-03-11",
    },
}
...
```

This can be useful when you need to load serialized data into an existing database and you cannot guarantee that the serialized primary key value is not already in use, and do not need to ensure that deserialized objects retain the same primary keys.

If you are using *dumpdata* to generate serialized data, use the *dumpdata --natural-foreign* and *dumpdata --natural-primary* command line flags to generate natural keys.

> **Note**
>
> You don't need to define both `natural_key()` and `get_by_natural_key()`. If you don't want Django to output natural keys during serialization, but you want to retain the ability to load natural keys, then you can opt to not implement the `natural_key()` method.
>
> Conversely, if (for some strange reason) you want Django to output natural keys during serialization, but not be able to load those key values, just don't define the `get_by_natural_key()` method.

Natural keys and forward references

Sometimes when you use natural foreign keys you'll need to deserialize data where an object has a foreign key referencing another object that hasn't yet been deserialized. This is called a "forward reference".

For instance, suppose you have the following objects in your fixture:

```
...
{
    "model": "store.book",
    "fields": {"name": "Mostly Harmless", "author": ["Douglas", "Adams"]},
},
...
{"model": "store.person", "fields": {"first_name": "Douglas", "last_name": "Adams"}},
...
```

In order to handle this situation, you need to pass `handle_forward_references=True` to `serializers.deserialize()`. This will set the `deferred_fields` attribute on the `DeserializedObject` instances. You'll need to keep track of `DeserializedObject` instances where this attribute isn't `None` and later call `save_deferred_fields()` on them.

Typical usage looks like this:

```
objs_with_deferred_fields = []

for obj in serializers.deserialize("xml", data, handle_forward_references=True):
    obj.save()
    if obj.deferred_fields is not None:
        objs_with_deferred_fields.append(obj)

for obj in objs_with_deferred_fields:
    obj.save_deferred_fields()
```

For this to work, the `ForeignKey` on the referencing model must have `null=True`.

Dependencies during serialization

It's often possible to avoid explicitly having to handle forward references by taking care with the ordering of objects within a fixture.

To help with this, calls to *`dumpdata`* that use the *`dumpdata --natural-foreign`* option will serialize any model with a `natural_key()` method before serializing standard primary key objects.

However, this may not always be enough. If your natural key refers to another object (by using a foreign key or natural key to another object as part of a natural key), then you need to be able to ensure that the objects on which a natural key depends occur in the serialized data before the natural key requires them.

To control this ordering, you can define dependencies on your `natural_key()` methods. You do this by setting a `dependencies` attribute on the `natural_key()` method itself.

For example, let's add a natural key to the `Book` model from the example above:

```
class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person, on_delete=models.CASCADE)

    def natural_key(self):
        return (self.name,) + self.author.natural_key()
```

The natural key for a `Book` is a combination of its name and its author. This means that `Person` must be serialized before `Book`. To define this dependency, we add one extra line:

```
def natural_key(self):
    return (self.name,) + self.author.natural_key()


natural_key.dependencies = ["example_app.person"]
```

This definition ensures that all `Person` objects are serialized before any `Book` objects. In turn, any object referencing `Book` will be serialized after both `Person` and `Book` have been serialized.

3.22 Django settings

A Django settings file contains all the configuration of your Django installation. This document explains how settings work and which settings are available.

3.22.1 The basics

A settings file is just a Python module with module-level variables.

Here are a couple of example settings:

```
ALLOWED_HOSTS = ["www.example.com"]
DEBUG = False
DEFAULT_FROM_EMAIL = "webmaster@example.com"
```

> **Note**
>
> If you set *DEBUG* to `False`, you also need to properly set the *ALLOWED_HOSTS* setting.

Because a settings file is a Python module, the following apply:

- It doesn't allow for Python syntax errors.
- It can assign settings dynamically using normal Python syntax. For example:

  ```
  MY_SETTING = [str(i) for i in range(30)]
  ```

- It can import values from other settings files.

3.22.2 Designating the settings

`DJANGO_SETTINGS_MODULE`

When you use Django, you have to tell it which settings you're using. Do this by using an environment variable, *DJANGO_SETTINGS_MODULE*.

The value of *DJANGO_SETTINGS_MODULE* should be in Python path syntax, e.g. `mysite.settings`. Note that the settings module should be on the Python `sys.path`.

The `django-admin` utility

When using django-admin, you can either set the environment variable once, or explicitly pass in the settings module each time you run the utility.

Example (Unix Bash shell):

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

Example (Windows shell):

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

Use the `--settings` command-line argument to specify the settings manually:

```
django-admin runserver --settings=mysite.settings
```

On the server (`mod_wsgi`)

In your live server environment, you'll need to tell your WSGI application what settings file to use. Do that with `os.environ`:

```
import os

os.environ["DJANGO_SETTINGS_MODULE"] = "mysite.settings"
```

Read the Django mod_wsgi documentation for more information and other common elements to a Django WSGI application.

3.22.3 Default settings

A Django settings file doesn't have to define any settings if it doesn't need to. Each setting has a sensible default value. These defaults live in the module django/conf/global_settings.py.

Here's the algorithm Django uses in compiling settings:

- Load settings from `global_settings.py`.
- Load settings from the specified settings file, overriding the global settings as necessary.

Note that a settings file should not import from `global_settings`, because that's redundant.

Seeing which settings you've changed

The command `python manage.py diffsettings` displays differences between the current settings file and Django's default settings.

For more, see the *`diffsettings`* documentation.

3.22.4 Using settings in Python code

In your Django apps, use settings by importing the object `django.conf.settings`. Example:

```
from django.conf import settings

if settings.DEBUG:
```

(continues on next page)

(continued from previous page)

```
    # Do something
    ...
```

Note that `django.conf.settings` isn't a module – it's an object. So importing individual settings is not possible:

```
from django.conf.settings import DEBUG  # This won't work.
```

Also note that your code should not import from either `global_settings` or your own settings file. `django.conf.settings` abstracts the concepts of default settings and site-specific settings; it presents a single interface. It also decouples the code that uses settings from the location of your settings.

3.22.5 Altering settings at runtime

You shouldn't alter settings in your applications at runtime. For example, don't do this in a view:

```
from django.conf import settings

settings.DEBUG = True  # Don't do this!
```

The only place you should assign to settings is in a settings file.

3.22.6 Security

Because a settings file contains sensitive information, such as the database password, you should make every attempt to limit access to it. For example, change its file permissions so that only you and your web server's user can read it. This is especially important in a shared-hosting environment.

3.22.7 Available settings

For a full list of available settings, see the settings reference.

3.22.8 Creating your own settings

There's nothing stopping you from creating your own settings, for your own Django apps, but follow these guidelines:

- Setting names must be all uppercase.
- Don't reinvent an already-existing setting.

For settings that are sequences, Django itself uses lists, but this is only a convention.

3.22.9 Using settings without setting DJANGO_SETTINGS_MODULE

In some cases, you might want to bypass the *DJANGO_SETTINGS_MODULE* environment variable. For example, if you're using the template system by itself, you likely don't want to have to set up an environment variable pointing to a settings module.

In these cases, you can configure Django's settings manually. Do this by calling:

django.conf.settings.configure(default_settings, **settings)

Example:

```
from django.conf import settings

settings.configure(DEBUG=True)
```

Pass `configure()` as many keyword arguments as you'd like, with each keyword argument representing a setting and its value. Each argument name should be all uppercase, with the same name as the settings described above. If a particular setting is not passed to `configure()` and is needed at some later point, Django will use the default setting value.

Configuring Django in this fashion is mostly necessary – and, indeed, recommended – when you're using a piece of the framework inside a larger application.

Consequently, when configured via `settings.configure()`, Django will not make any modifications to the process environment variables (see the documentation of *TIME_ZONE* for why this would normally occur). It's assumed that you're already in full control of your environment in these cases.

Custom default settings

If you'd like default values to come from somewhere other than `django.conf.global_settings`, you can pass in a module or class that provides the default settings as the `default_settings` argument (or as the first positional argument) in the call to `configure()`.

In this example, default settings are taken from `myapp_defaults`, and the *DEBUG* setting is set to `True`, regardless of its value in `myapp_defaults`:

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

The following example, which uses `myapp_defaults` as a positional argument, is equivalent:

```
settings.configure(myapp_defaults, DEBUG=True)
```

Normally, you will not need to override the defaults in this fashion. The Django defaults are sufficiently tame that you can safely use them. Be aware that if you do pass in a new default module, it entirely replaces the

Django defaults, so you must specify a value for every possible setting that might be used in the code you are importing. Check in `django.conf.settings.global_settings` for the full list.

Either `configure()` or `DJANGO_SETTINGS_MODULE` is required

If you're not setting the *DJANGO_SETTINGS_MODULE* environment variable, you must call `configure()` at some point before using any code that reads settings.

If you don't set *DJANGO_SETTINGS_MODULE* and don't call `configure()`, Django will raise an `ImportError` exception the first time a setting is accessed.

If you set *DJANGO_SETTINGS_MODULE*, access settings values somehow, then call `configure()`, Django will raise a `RuntimeError` indicating that settings have already been configured. There is a property for this purpose:

`django.conf.settings.configured`

For example:

```
from django.conf import settings

if not settings.configured:
    settings.configure(myapp_defaults, DEBUG=True)
```

Also, it's an error to call `configure()` more than once, or to call `configure()` after any setting has been accessed.

It boils down to this: Use exactly one of either `configure()` or *DJANGO_SETTINGS_MODULE*. Not both, and not neither.

Calling `django.setup()` is required for "standalone" Django usage

If you're using components of Django "standalone" – for example, writing a Python script which loads some Django templates and renders them, or uses the ORM to fetch some data – there's one more step you'll need in addition to configuring settings.

After you've either set *DJANGO_SETTINGS_MODULE* or called `configure()`, you'll need to call *django.setup()* to load your settings and populate Django's application registry. For example:

```
import django
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
django.setup()
```

(continues on next page)

(continued from previous page)

```
# Now this script or any imported module can use any part of Django it needs.
from myapp import models
```

Note that calling `django.setup()` is only necessary if your code is truly standalone. When invoked by your web server, or through django-admin, Django will handle this for you.

`django.setup()` may only be called once.

Therefore, avoid putting reusable application logic in standalone scripts so that you have to import from the script elsewhere in your application. If you can't avoid that, put the call to `django.setup()` inside an `if` block:

```
if __name__ == "__main__":
    import django

    django.setup()
```

See also

The Settings Reference
: Contains the complete list of core and contrib app settings.

3.23 Signals

Django includes a "signal dispatcher" which helps decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain senders to notify a set of receivers that some action has taken place. They're especially useful when many pieces of code may be interested in the same events.

For example, a third-party app can register to be notified of settings changes:

```
from django.apps import AppConfig
from django.core.signals import setting_changed


def my_callback(sender, **kwargs):
    print("Setting changed!")


class MyAppConfig(AppConfig):
```

(continues on next page)

(continued from previous page)

```
    ...

    def ready(self):
        setting_changed.connect(my_callback)
```

Django's built-in signals let user code get notified of certain actions.

You can also define and send your own custom signals. See Defining and sending signals below.

> **Warning**
>
> Signals give the appearance of loose coupling, but they can quickly lead to code that is hard to understand, adjust and debug.
>
> Where possible you should opt for directly calling the handling code, rather than dispatching via a signal.

3.23.1 Listening to signals

To receive a signal, register a receiver function using the *`Signal.connect()`* method. The receiver function is called when the signal is sent. All of the signal's receiver functions are called one at a time, in the order they were registered.

`Signal.connect`(receiver, sender=None, weak=True, dispatch_uid=None)

Parameters

- **`receiver`** – The callback function which will be connected to this signal. See Receiver functions for more information.
- **`sender`** – Specifies a particular sender to receive signals from. See Connecting to signals sent by specific senders for more information.
- **`weak`** – Django stores signal handlers as weak references by default. Thus, if your receiver is a local function, it may be garbage collected. To prevent this, pass `weak=False` when you call the signal's `connect()` method.
- **`dispatch_uid`** – A unique identifier for a signal receiver in cases where duplicate signals may be sent. See Preventing duplicate signals for more information.

Let's see how this works by registering a signal that gets called after each HTTP request is finished. We'll be connecting to the *`request_finished`* signal.

Receiver functions

First, we need to define a receiver function. A receiver can be any Python function or method:

```
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Notice that the function takes a `sender` argument, along with wildcard keyword arguments (`**kwargs`); all signal handlers must take these arguments.

We'll look at senders a bit later, but right now look at the `**kwargs` argument. All signals send keyword arguments, and may change those keyword arguments at any time. In the case of *`request_finished`*, it's documented as sending no arguments, which means we might be tempted to write our signal handling as `my_callback(sender)`.

This would be wrong – in fact, Django will throw an error if you do so. That's because at any point arguments could get added to the signal and your receiver must be able to handle those new arguments.

Receivers may also be asynchronous functions, with the same signature but declared using `async def`:

```
async def my_callback(sender, **kwargs):
    await asyncio.sleep(5)
    print("Request finished!")
```

Signals can be sent either synchronously or asynchronously, and receivers will automatically be adapted to the correct call-style. See sending signals for more information.

Connecting receiver functions

There are two ways you can connect a receiver to a signal. You can take the manual connect route:

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

Alternatively, you can use a *`receiver()`* decorator:

receiver(signal, **kwargs)

Parameters

- **signal** – A signal or a list of signals to connect a function to.
- **kwargs** – Wildcard keyword arguments to pass to a function.

Here's how you connect with the decorator:

```
from django.core.signals import request_finished
from django.dispatch import receiver


@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Now, our `my_callback` function will be called each time a request finishes.

Where should this code live?

Strictly speaking, signal handling and registration code can live anywhere you like, although it's recommended to avoid the application's root module and its `models` module to minimize side-effects of importing code.

In practice, signal handlers are usually defined in a `signals` submodule of the application they relate to. Signal receivers are connected in the *`ready()`* method of your application configuration class. If you're using the *`receiver()`* decorator, import the `signals` submodule inside *`ready()`*, this will implicitly connect signal handlers:

```
from django.apps import AppConfig
from django.core.signals import request_finished


class MyAppConfig(AppConfig):
    ...

    def ready(self):
        # Implicitly connect signal handlers decorated with @receiver.
        from . import signals

        # Explicitly connect a signal handler.
        request_finished.connect(signals.my_callback)
```

Note

The *`ready()`* method may be executed more than once during testing, so you may want to guard your signals from duplication, especially if you're planning to send them within tests.

Connecting to signals sent by specific senders

Some signals get sent many times, but you'll only be interested in receiving a certain subset of those signals. For example, consider the *django.db.models.signals.pre_save* signal sent before a model gets saved. Most of the time, you don't need to know when any model gets saved – just when one specific model is saved.

In these cases, you can register to receive signals sent only by particular senders. In the case of *django.db.models.signals.pre_save*, the sender will be the model class being saved, so you can indicate that you only want signals sent by some model:

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel


@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs): ...
```

The `my_handler` function will only be called when an instance of `MyModel` is saved.

Different signals use different objects as their senders; you'll need to consult the built-in signal documentation for details of each particular signal.

Preventing duplicate signals

In some circumstances, the code connecting receivers to signals may run multiple times. This can cause your receiver function to be registered more than once, and thus called as many times for a signal event. For example, the *ready()* method may be executed more than once during testing. More generally, this occurs everywhere your project imports the module where you define the signals, because signal registration runs as many times as it is imported.

If this behavior is problematic (such as when using signals to send an email whenever a model is saved), pass a unique identifier as the `dispatch_uid` argument to identify your receiver function. This identifier will usually be a string, although any hashable object will suffice. The end result is that your receiver function will only be bound to the signal once for each unique `dispatch_uid` value:

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

3.23.2 Defining and sending signals

Your applications can take advantage of the signal infrastructure and provide its own signals.

> **When to use custom signals**
>
> Signals are implicit function calls which make debugging harder. If the sender and receiver of your custom signal are both within your project, you're better off using an explicit function call.

Defining signals

class Signal

All signals are *django.dispatch.Signal* instances.

For example:

```
import django.dispatch

pizza_done = django.dispatch.Signal()
```

This declares a `pizza_done` signal.

Sending signals

There are two ways to send signals synchronously in Django.

Signal.send(sender, **kwargs)

Signal.send_robust(sender, **kwargs)

Signals may also be sent asynchronously.

Signal.asend(sender, **kwargs)

Signal.asend_robust(sender, **kwargs)

To send a signal, call either *Signal.send()*, *Signal.send_robust()*, *await Signal.asend()*, or *await Signal.asend_robust()*. You must provide the `sender` argument (which is a class most of the time) and may provide as many other keyword arguments as you like.

For example, here's how sending our `pizza_done` signal might look:

```
class PizzaStore:
    ...

    def send_pizza(self, toppings, size):
```

(continues on next page)

(continued from previous page)

```
        pizza_done.send(sender=self.__class__, toppings=toppings, size=size)
        ...
```

All four methods return a list of tuple pairs `[(receiver, response), ...]`, representing the list of called receiver functions and their response values.

`send()` differs from `send_robust()` in how exceptions raised by receiver functions are handled. `send()` does not catch any exceptions raised by receivers; it simply allows errors to propagate. Thus not all receivers may be notified of a signal in the face of an error.

`send_robust()` catches all errors derived from Python's `Exception` class, and ensures all receivers are notified of the signal. If an error occurs, the error instance is returned in the tuple pair for the receiver that raised the error.

The tracebacks are present on the `__traceback__` attribute of the errors returned when calling `send_robust()`.

`asend()` is similar to `send()`, but it is a coroutine that must be awaited:

```
async def asend_pizza(self, toppings, size):
    await pizza_done.asend(sender=self.__class__, toppings=toppings, size=size)
    ...
```

Whether synchronous or asynchronous, receivers will be correctly adapted to whether `send()` or `asend()` is used. Synchronous receivers will be called using *`sync_to_async()`* when invoked via `asend()`. Asynchronous receivers will be called using *`async_to_sync()`* when invoked via `send()`. Similar to the case for middleware, there is a small performance cost to adapting receivers in this way. Note that in order to reduce the number of sync/async calling-style switches within a `send()` or `asend()` call, the receivers are grouped by whether or not they are async before being called. This means that an asynchronous receiver registered before a synchronous receiver may be executed after the synchronous receiver. In addition, async receivers are executed concurrently using `asyncio.gather()`.

All built-in signals, except those in the async request-response cycle, are dispatched using *`Signal.send()`*.

3.23.3 Disconnecting signals

`Signal.disconnect`(receiver=None, sender=None, dispatch_uid=None)

To disconnect a receiver from a signal, call *`Signal.disconnect()`*. The arguments are as described in *`Signal.connect()`*. The method returns `True` if a receiver was disconnected and `False` if not. When `sender` is passed as a lazy reference to `<app label>.<model>`, this method always returns `None`.

The `receiver` argument indicates the registered receiver to disconnect. It may be `None` if `dispatch_uid` is used to identify the receiver.

3.24 System check framework

The system check framework is a set of static checks for validating Django projects. It detects common problems and provides hints for how to fix them. The framework is extensible so you can easily add your own checks.

Checks can be triggered explicitly via the *check* command. Checks are triggered implicitly before most commands, including *runserver* and *migrate*. For performance reasons, checks are not run as part of the WSGI stack that is used in deployment. If you need to run system checks on your deployment server, trigger them explicitly using *check*.

Serious errors will prevent Django commands (such as *runserver*) from running at all. Minor problems are reported to the console. If you have inspected the cause of a warning and are happy to ignore it, you can hide specific warnings using the *SILENCED_SYSTEM_CHECKS* setting in your project settings file.

A full list of all checks that can be raised by Django can be found in the System check reference.

3.24.1 Writing your own checks

The framework is flexible and allows you to write functions that perform any other kind of check you may require. The following is an example stub check function:

```
from django.core.checks import Error, register


@register()
def example_check(app_configs, **kwargs):
    errors = []
    # ... your check logic here
    if check_failed:
        errors.append(
            Error(
                "an error",
                hint="A hint.",
                obj=checked_object,
                id="myapp.E001",
            )
        )
    return errors
```

The check function must accept an `app_configs` argument; this argument is the list of applications that should be inspected. If `None`, the check must be run on all installed apps in the project.

The check will receive a `databases` keyword argument. This is a list of database aliases whose connections may be used to inspect database level configuration. If `databases` is `None`, the check must not use any