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",
    },
    "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.
        """
        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:

```python
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):

```python
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:

```python
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)

    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:

```python
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
        )

    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


# 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.