Fixtures

See also:

- How to provide initial data for models

What is a fixture?

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.

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 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>
```

Where Django looks for fixtures?

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


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.