django-class-fixtures Documentation

Release 0.9a1

JK Laiho

October 04, 2014

Contents

The purpose of django-class-fixtures is to augment Django's fixture system, used for loading initial data during `manage.py syncdb` calls, test data with the `TestCase.fixtures` iterable, and other types of data with manual invocations of `manage.py loaddata`.

While still supporting Django's traditional model serialization formats, django-class-fixtures provides pure Python **class-based fixtures**. They take the form of model instance definitions in a syntax that is very close to how you'd create new instances with `SomeModel.objects.create(**kwargs)`. The thinking goes: why do fixtures in JSON, XML or YAML, when you could just as well do it all with Python code?

This initial public release, 0.8, only implements the **loading** of class-based fixtures through a `loaddata` override. The fixtures have to be created by hand until the release of version 0.9, which brings `dumpdata` support and is the first version I'd expect anyone besides myself to actually use in real-world projects. See *Planned future releases* for details on future versions.

Even though 0.8 already has rather decent test coverage, use it at your own risk. The first somewhat safe production version will be 1.0. The standard non-liability clauses in `LICENSE.txt` apply then as well, of course.

See *Getting started* for an introduction. After that, check out *Using class-based fixtures* for some additional information about the capabilities of django-class-fixtures, and some neat tricks that class-based fixtures enable you to pull off.

Good luck!

CHAPTER 1

Installation

Dependencies:

- Python 2.6+
- Django 1.2+

Until version 1.0 is released (see *Planned future releases* for details), django-class-fixtures will only be available through a GitHub source checkout. Post-1.0, you can get it off PyPi using `easy_install` or `pip`.

The GitHub page for this project is at https://github.com/jklaiho/django-class-fixtures. Once you've grabbed the source, run `sudo python setup.py install` (you can most likely drop the `sudo` if you're using `virtualenv`).

To avoid waking up screaming in the middle of the night, you can run the test suite either with `python setup.py test` from the initial source checkout, or with `python manage.py test class_fixtures` from inside a Django project directory, once you've got one up and running.

CHAPTER 2

Configuration

There's not much of it.

After installing django-class-fixtures into your Python environment, insert `class_fixtures` into `INSTALLED_APPS` in the settings file of your Django project. Doesn't matter where. All this does is override the `loaddata` management command with a version that supports class-based fixtures. No models are installed, so no need for `syncdb` or schema migrations.

Note: No other `loaddata`-overriding apps should be present in `INSTALLED_APPS`. Depending on the order of the apps listed there, django-class_fixtures' override may not end up being the active one, and your class-based fixtures won't get loaded. Even if it is the active one, then your *other* `loaddata` override won't work, which is probably not what you want either.

If you wish to place fixtures outside of the `fixtures` directories of your Django apps (i.e. use "project-level" fixtures), use the `FIXTURE_PACKAGES` setting, an iterable similar to Django's own `FIXTURE_DIRS`, only containing dotted-path notation to Python packages containing fixture modules.

Example:

```
FIXTURE_PACKAGES = (
    'myproject.something.fixtures',
    'someplace.other.project_fixtures',
)
```

Obviously, the module paths listed must be valid and importable in the Python environment that your Django project lives in. Make sure they have `__init__.py` modules.

With that out of the way, check out the *Getting started* guide to, well, get started.

CHAPTER 3

Getting started

If you're not familiar with Django's existing fixture system, please look at the relevant Django documentation before going further. Things like why fixtures exist, how and when they are used and what they are good for are not covered here. Coming here, you've optimally used Django's fixtures for some time and, if you're like me, gotten a little frustrated with them.

If you haven't already, have a quick look at the *Configuration* document. It's very short. All you need to do, basically, is to add `class_fixtures` to your `INSTALLED_APPS`.

Let's dive in to the practical stuff right away.

3.1 Traditional fixtures vs. class-based fixtures

We'll use a band-related Django app called *bandaid* and its models to illustrate the use of the basic features of django-class-fixtures.

Assume a simplistic Django model inside `bandaid.models`, like this:

```
class Band(models.Model):
    name = models.CharField(max_length=255)
    genre = models.CharField(max_length=100)
```

A typical way of creating an instance of this model would look like this:

```
band = Band.objects.create(name="Bar Fighters", genre="Rock")
```

When serialized to JSON using the `dumpdata` management command, it would look like this:

```
[
    {
        "pk": 1,
        "model": "bandaid.band",
        "fields":
        {
            "name": "Bar Fighters",
            "genre": "Rock"
        }
    }
]
```

That `dumpdata` output is saved as `example_band.json` inside `bandaid/fixtures`. Our hypothetical *bandaid* app now looks like this:

```
bandaid/
    __init__.py
    models.py
    fixtures/
        example_band.json
```

The JSON serialization of the `Band` object is simple enough, but files with lots of serialized objects become cumbersome to deal with very quickly, especially when using schema migrations. We'd rather just deal with Python code all the way. It's more readable, as well as easier to modify by hand and keep current with schema changes.

To that end, here's the code of a Python module. It contains a single class-based fixture that contains a single object: a second instance of the `Band` model:

```
from class_fixtures.models import Fixture
from bandaid.models import Band

bands = Fixture(Band)
bands.add(2, name="Brutallica", genre="Metal")
```

That's it. Save that as `more_bands.py` in the *bandaid* app's `fixtures` directory. It becomes a *fixture module* that django-class-fixtures can use. You're good to go.

Wait, not quite.

You'll notice that `fixtures` is a simple data directory inside the *bandaid* app. It's not a Python package, which it needs to be for django-class-fixtures to find any fixture modules inside it. This is easily fixed: just add an `__init__.py` module inside it. The bandaid app now looks like this:

```
bandaid/
    __init__.py
    models.py
    fixtures/
        __init__.py
        example_band.json
        more_bands.py
```

Like magic, the sad and lonely `fixtures` directory is transformed into a subpackage of the *bandaid* app where Python modules can be imported from. As you can see, you can freely mix traditional serialized fixtures and class-based fixtures. You just have to make sure that the primary keys (the "pk" fields in JSON files and the first parameters to the `add()` calls in fixture modules) don't conflict with each other.

Note: A bit on terminology: in Django parlance, a "fixture" is the JSON/XML/YAML file containing serialized model objects, located inside one of the fixture directories.

As for django-class-fixtures, a "fixture" is a single instance of the `Fixture` class, many of which can and will appear inside a single *fixture module*, which is the actual .py file inside one of the fixture packages (i.e. directories with an `__init__.py` file).

All right, the basic infrastructure is in place. Let's look at what we just did with the code in that fixture module.

3.2 Basic use of class-based fixtures

First off, fixture modules need to import the `Fixture` class from `class_fixtures.models` and any model classes that you want to create fixtures for.

Note: While the `Fixture` class lives in `class_fixture.models`, it's not a Django model class. It's model-

related, though, and since Django apps need to have a `models.py` file anyway, it's as good a place as any for it.

Each `Fixture` instance is attached to a model class by giving the model as the first parameter to `Fixture`, like `Fixture(Band)` above. For purposes of organization or clarity, you can have multiple `Fixture` instances per model class:

```
decent_bands = Fixture(Band)
awful_bands = Fixture(Band)
```

Actually populating the `Fixture` instances with objects is done using the `add()` method. First, you give it the primary key and then, as keyword arguments, the same arguments you'd give a `Band.objects.create()` call:

```
decent_bands.add(3, name="Led Dirigible", genre="Rock")
awful_bands.add(4, name="Flaxxid Bizkit", genre="Crap")
```

Note that the primary keys must keep incrementing across `Fixture` instances, since both of them are still going to create `Band` objects into the same database table.

Note: If you are curious as to why the primary key needs to be hard-coded, see *Hard-coded primary keys*. It's not necessary for learning how class-based fixtures work, though, so if this is your first time around, it's best to just keep moving for now.

Of course, eventually you'll have to create relations between objects.

You'll be doing two kinds of relations with fixtures:

1. Relations to objects that do not yet exist in the database, but are instead created in the same fixture module as the objects that point to them.
2. Relations to objects that exist in the database prior to `loaddata` being run (e.g. objects defined in `initial_data` fixtures created during `syncdb`).

Let's look at these in that order.

3.3 Relationships between objects inside the same fixture module

We'll start with foreign keys and these two example models from a hypothetical app called *wage_slave*:

```
class Company(models.Model):
    name = models.CharField(max_length=100)

class Employee(models.Model):
    name = models.CharField(max_length=100)
    company = models.ForeignKey(Company)
    manager = models.ForeignKey('self', null=True)
```

We'll let code speak for itself first and then explain:

```
from class_fixtures import Fixture
from wage_slave.models import Company, Employee

companies = Fixture(Company)
employees = Fixture(Employee)

companies.add(1, name="FacelessCorp Inc.")
employees.add(1, name="Ty Rant", company=companies.fk(1))
employees.add(2, name="Sue Ecide-Risk", company=companies.fk(1), manager=employees.fk(1))
```

As is hopefully apparent, we're creating one `Company` and two `Employees`, one of which is the manager of the other one. The above demonstrates both a foreign key to another model (the `company` of both employees) and a recursive FK to the same model (Sue's `manager`). This is done using the `fk()` method of the target fixture instance, giving it the primary key.

Note: An aside: which would you rather look at and deal with: those few rows of Python, or their imagined JSON representations? Just sayin'.

Due to the foreign key to `Company`, `Employee` objects depend on their target `Company` instances existing before they are defined. `Fixture` instances handle dependency resolution behind the scenes, so as long as you have created the `companies` and `employees` instances first, it doesn't matter in what order you `add()` the actual model instance definitions to them. The proper loading order is determined automatically.

One-to-one relations work basically identically to foreign keys. To demonstrate, here's one more model:

```
class EmployeeHistory(models.Model):
    employee = models.OneToOneField(Employee)
    date_joined = models.DateField()
```

Using that model in the above scenario is as simple as you might guess:

```
# Remember to add EmployeeHistory to the import from wage_slave.models
histories = Fixture(EmployeeHistory)
histories.add(employee=employees.o2o(1), date_joined='2003-03-15')
```

The `o2o()` method works identically to `fk()`. In fact, internally it's the very same method, just a different alias. Picking the right one just makes fixture code more self-documenting.

The implicit OneToOneFields created by concrete model inheritance don't need explicit `o2o()` usage. Here's an example with an additional model that inherits from `bandaid.Band`:

```
# in bandaid.models
class MetalBand(Band):
    leather_pants_worn = models.BooleanField(default=False)

# in some fixture module
metalbands = Fixture(MetalBand)
metalbands.add(666, name="Judas Bishop", genre="Metal", leather_pants_worn=True)
```

Nothing too special happens here; it relies on Django's model inheritance functionality, where creating a `MetalBand` object will automatically create a `Band` object with the same primary key. You just need to be careful to not overlap the primary keys of any of the previously defined `Band` objects.

What about Many-to-many relationships? To demonstrate their use, we'll add a few more models to the *bandaid* app:

```
class Musician(models.Model):
    name = models.CharField(max_length=100)
    member_of = models.ManyToManyField(Band, through='Membership')

class Membership(models.Model):
    musician = models.ForeignKey(Musician)
    band = models.ForeignKey(Band)
    instrument = models.CharField(max_length=100)

class Roadie(models.Model):
    name = models.CharField(max_length=100)
    hauls_for = models.ManyToManyField(Band)
```

Again, let's look at some code first. Here's a revised form of `bandaid.fixtures.more_bands`:

```
from class_fixtures import Fixture
from bandaid.models import Band, Musician, Membership, Roadie

bands = Fixture(Band)
musicians = Fixture(Musician)
memberships = Fixture(Membership)
roadies = Fixture(Roadie)

bands.add(2, name="Brutallica", genre="Metal")
bands.add(3, name="Led Dirigible", genre="Rock")
bands.add(4, name="Flaxxid Bizkit", genre="Crap")
musicians.add(1, name="Lars Toorich")
# A "through" M2M, musician-to-band-via-membership
membership.add(1, musician=musicians.fk(1), band=bands.fk(2), instrument="Bongos")
# A normal M2M
roadies.add(1, name="Tats Brimhat", hauls_for=[bands.m2m(2)])
```

Not many surprises there. "Through" M2Ms are just a couple of foreign keys in the "middle" model in addition to any extra fields. The only thing of note is the direct assignment of a single-item list to the `hauls_for` ManyToManyField to create the M2M relation between Roadie and Band. We just inline the M2M relation directly to the object definition, just like with foreign keys earlier.

This is in contrast to Django, where you'd do the same like this:

```
brutallica = Band.objects.create(name="Brutallica", genre="Metal")
tats = Roadie.objects.create(name="Tats Brimhat")
# Either...
tats.hauls_for.add(brutallica)
# Or...
brutallica.roadie_set.add(tats)
```

So, to create the M2M relation using Django's ORM, you need an extra call to the `add()` method of the ManyRelatedManager (`hauls_for` or `roadie_set`, respectively) after creating the objects that relate to each other.

M2M relations, of course, can be defined from either end of the relation. This means that the following is also legal, and equivalent to the earlier example of creating the roadie-band relation inline:

```
# Create the M2M relation from the other end, i.e. the "target" of the
# Roadie.hauls_for ManyToManyField. As you'll recall, the add() statements
# can be in any order, so we can call roadies.m2m(1) before a Roadie with
# that primary key is added to the "roadies" Fixture instance.
bands.add(2, name="Brutallica", genre="Metal", roadie_set=[roadies.m2m(1)])
roadies.add(1, name="Tats Brimhat")
```

The argument to M2M fields needs to be an iterable, even if it just has the one element. To create several M2M relations, you just add elements to the iterable. For example, to create a severely overworked roadie for a bunch of bands:

```
roadies.add(1, name="Tats Brimhat", hauls_for=[
    bands.m2m(2),
    bands.m2m(3),
    bands.m2m(4),
])
```

As it happens, `m2m()` is also just an alias of `fk()` and `o2o()`. Internally, a special object called a *delayed related object loader* is created for all three relation types, and resolved to actual objects later on in the loading process.

3.4 Relations from fixture modules to pre-existing objects

Sometimes you'll need to define relations to objects that you know to exist in the database prior to the loading of your fixture module. Syntax-wise, this is actually somewhat simpler than the intra-module relations presented under the previous heading.

Those pre-existing objects could be coming from `syncdb` calls that load initial data, or from old-style Django-serialized fixtures that got loaded before our class-based fixture did, or even from other class-based fixture modules, ones we know to be loaded beforehand.

In our case, the *bandaid* app contains a single `Band` object serialized into the `example_band.json` file under `bandaid/fixtures`. We'll assume that it was loaded first, and create a relation to it inside the `more_bands.py` fixture module like this:

```
musicians.add(2, name="Dave Growl")
# "Bar Fighters" in example_band.json has 1 as its primary key
memberships.add(2, musician=musicians.fk(2), band=1, instrument="All of them")
```

As you can see from the `band` argument, instead of referring to the primary key of a not-yet-created object through the `fk()` method of the target `Fixture` instance (like `musicians.fk(2)`), you just give it the primary key of the pre-existing related object directly.

Note: Relations to pre-existing objects using natural key tuples instead of primary keys are covered in *Natural keys* in the *Additional Information* document.

Or, you can just retrieve the actual model object inside the fixture module and relate to that. This is an alternative to the previous example:

```
bf = Band.objects.get(name="Bar Fighters")
musicians.add(2, name="Dave Growl")
memberships.add(2, musician=musicians.fk(2), band=bf, instrument="All of them")
```

M2Ms work no different. Assuming we had some other JSON fixture that defined bands with PKs 5 and 6, we'd just do this to relate to them inside our fixture module:

```
roadies.add(2, name="Ciggy Tardust", hauls_for=[5,6])

# or, if we wish to relate to the actual objects,
# perhaps not even knowing or caring what the PKs are:

some_band = Band.objects.get(**some_kwargs)
other_band = Band.objects.get(**other_kwargs)
roadies.add(2, name="Ciggy Tardust", hauls_for=[some_band, other_band])
```

This business with creating relations to objects outside the current fixture module brings up a point that bears emphasizing:

Warning: Don't mix traditional fixtures with class-based fixtures unless you have a compelling reason to do so. If you do, be careful. Django-class-fixtures can handle dependencies inside a single fixture module, but you need to manually ensure that Django-style serialized fixtures are **always** loaded before the class-based fixture modules that relate to objects contained therein.
The same goes for dependencies between class-based fixture modules. django-class-fixtures doesn't currently support inter-module dependencies. If, through relation dependencies, you make assumptions about the loading order of the fixture modules, be very careful to actually load them in the correct order, always, without fail.

That concludes our coverage of the basic concepts and use of class-based fixtures. You're not done yet, though. Next, it's recommended you look at *Using class-based fixtures* for a brief look at actually using the fixture modules in various

scenarios.

For information on topics like using natural keys to create relations, more in-depth technical details and a few gotchas, see *Additional Information*.

CHAPTER 4

Using class-based fixtures

Since django-class-fixtures provides a drop-in replacement for Django's `loaddata`, you use it exactly as you would Django's fixtures, but with a few bonuses.

4.1 Tests

As you probably know, to use fixtures in Django, you set up `TestCase.fixtures`, like so:

```
class WhatLovelyTests(TestCase):
    fixtures = ["something", "other_things.json"]

    def test_random_stuff(self):
        (...)
```

`TestCase.fixtures` is an iterable of strings, corresponding to fixture file names (with or without specifying the extension), located under the `fixtures` directory of any app, or within any directory listed in `settings.FIXTURE_DIRS`. Django loads all fixtures with those names that it can find.

Note: One exception, which you know if you've learned Django's documentation on the topic: you can't have `something.json` **and** `something.xml` inside the `fixtures` directory of the same app and then refer to plain `"something"` inside `TestCase.fixtures`. Having those in two separate apps works fine and loads both, though.

That same example works as-is under our `loaddata` override, but django-class-fixtures allows you to put some other things in `TestCase.fixtures` as well:

```
from wage_slave.fixtures.some_module import employees
from bandaid.fixtures import other_bands

class ClassFixtureUsingTests(TestCase):
    fixtures = [
        employees, # 1
        other_bands, # 2
        "some_app_name", # 3
        "another_app.assorted_fixtures", # 4
        "something" # 5
    ]
```

1. An individual `Fixture` instance cherry-picked from its containing fixture module.
2. An individual fixture module. All `Fixture` instances contained within it are loaded.

3. The name of an app in `settings.INSTALLED_APPS`. All of its class-based fixtures are loaded in whatever order they are discovered. Traditional Django fixtures in that app are **not** loaded.
4. An "appname.fixture_module_name" string. This is an alternative to #2 that doesn't require you to import the fixture module. In this example, a fixture module called `assorted_fixtures.py` must reside in the `fixtures` subpackage/directory of `another_app`.
5. The name of a fixture module in one of the fixture directories.

Astute readers will have spotted a slight problem with #3, #4 and #5. Being strings, all would also be valid Django fixture names, and Django's `loaddata` would search for `some_app_name.(format)`, `another_app.assorted_fixtures.(format)` and `something.(format)` files, substituting (format) for all the supported serialization formats. So what gets loaded in case of duplicate names? This is covered in *Rules for fixture discovery and loading*. Of course, it helps if you don't mix traditional and class-based fixtures, if you can avoid it.

4.2 Initial data

By now, you may have guessed how initial data works. You stick your `Fixture` instances in a file called `initial_data.py`, and that's pretty much all there is to it. It will get loaded with `syncdb`.

In case you have both `initial_data.json` and `initial_data.py` in the same fixture directory/package, both will be loaded. The traditional JSON fixture will always be loaded first.

4.3 Manual `loaddata` calls

When running `python manage.py loaddata` with the name of a fixture on the command line, you obviously can't pass in a `Fixture` instance or an imported fixture module like you can in `TestCase.fixtures`. But in addition to the name of a fixture module, passing in the name of an app or an "appname.fixture_module_name" works (as described in options 3 to 5 in *the ways of using TestCase.fixtures* above).

Loading traditional Django fixtures works like before.

4.4 Dynamic approaches to fixture creation

It would be boring if django-class-fixtures didn't have any tricks up its sleeve. Here's a couple of ways to use the fact that the fixtures are just Python to your advantage.

4.4.1 Looping through structured data to create fixtures

Say you're tired of manually defining the primary keys for each model instance you add to a fixture. Well, here's a way of being a bit more terse, defining the field names just once instead of writing them out in every `add()` call, and automatically generating the required primary keys in situations when you know it's safe:

```
field_names = ("name", "genre")
data = [
    ("Bar Fighters", "Rock"),
    ("Brutallica", "Metal"),
    ("Led Dirigible", "Rock")
]
```

```
bands = Fixture(Band)
for i in range(len(data)):
    bands.add(i+1, **dict(zip(field_names, data[i])))
```

For a simplistic model like `Band` and such few instances, the above technique is a bit overkill. But for adding large amounts of instances of big models with more fields, it enables you to produce a lot less code.

Note: The above method is, in fact, one I've considered the `dumpdata` override coming in version 0.9 to use. To make it a bit cleaner, I may add a helper method to `Fixture` instances that takes the field name tuple and the data tuple as its parameters directly, doing the zip-dict dance internally.

The underlying point is to illustrate how you're not stuck with the canonical fixture construction method described in the examples around the documentation.

4.4.2 Inserting project-level data into app-level fixtures

One useful way of doing something beyond the capabilities of Django's serialized fixtures is to determine the data that `Fixture` instances contain at runtime.

The fixture discovery process doesn't care what else the fixture modules contain besides `Fixture` instances, so you can do all sort of coding gymnastics to produce the data contained in them.

One example: say you work at a company constructing e-commerce sites, each of which uses a payment processing app built in-house. Each site uses different merchant IDs and other content related to the payment processing app that you want stored in the database.

Since the app is reusable across sites, you'd like it to contain fixtures that you can configure on the project (i.e. site) level. You create a single data source per site in a predefined location (say, `projectfolder/site_customize`), from which the objects contained in the app-level `Fixture` instances get their site-specific data. Less custom settings, no site-specific modifications to the fixtures of the payment app. (This is functionality I wish I'd had at my fingertips on some previous projects.)

This is achievable by making the fixture module import the relevant data structures from a preset site-level location. Here's a simplified example, using two imaginary e-commerce sites, Cheese Emporium and Snake World, and their project-level customization for a payment app called Moolah:

```
# cheese_emporium/site_customize/payment.py
paypal_merchant_id = 12534768abcd
google_checkout_merchant_id = asdfqwerty

# snake_world/site_customize/payment.py
paypal_merchant_id = 87654321dcba
google_checkout_merchant_id = qwertyasdf

# In the fixture module, moolah/fixtures/merchant_data.py
# Determine what the project's root path is through a setting or
# something, import site_customize.payment from it.
processors = Fixture(PaymentProcessor)
processors.add(1, name="PayPal", merchant_id=payment.paypal_merchant_id)
processors.add(2, name="Google Checkout", merchant_id=payment.google_checkout_merchant_id)
```

No matter which site the app is attached to, the fixture module will insert the correct data when loaded.

4.4.3 And more, much more!

What else? Based on runtime conditions, leave out objects from, or add more objects to a `Fixture` instance, or determine what model to construct the instance with in the first place. The payment processor example above could easily be modified to only include PayPal for Cheese Emporium and Google Checkout for Snake World, based on some factors set on the site level and checked for in `if` clauses around the `add()` statements.

It is my sincere hope that django-class-fixtures enables its users to see fixtures in a new light. Some interesting new possibilities are there to be discovered.

Use your imagination. In the end, it's all just Python. Bend it to your will!

If you haven't already, now would be a good time to check out *Additional Information*.

CHAPTER 5

Additional Information

Here's some slightly more advanced information that you may find useful.

5.1 Natural keys

First of all, read and understand Django's natural key documentation.

Since version 0.8 of django-class-fixtures only supports `loaddata` but not `dumpdata`, we'll focus on the loading side of things.

To demonstrate natural keys, we'll use the following two models, `Competency` being the one with the custom manager required for natural key loading:

```
class CompetencyManager(models.Manager):
    def get_by_natural_key(self, framework, level):
        return self.get(framework=framework, level=level)

class Competency(models.Model):
    LEVEL_CHOICES = (
        (0, "None"),
        (1, "Beginner"),
        (2, "Intermediate"),
        (3, "Advanced"),
        (4, "Guru"),
    )
    framework = models.CharField(max_length=100)
    level = models.SmallIntegerField(choices=LEVEL_CHOICES)

    objects = CompetencyManager()

    class Meta(object):
        unique_together = (('framework', 'level'))


class JobPosting(models.Model):
    title = models.CharField(max_length=100)
    main_competency = models.ForeignKey(Competency, related_name='main_competency_for')
    additional_competencies = models.ManyToManyField(Competency, related_name='extra_competency_for')
```

Natural keys are meant to be used when relating from a fixture to an object that is already in the database, where defining the relation based on the target object's primary key is inconvenient or even impossible.

To demonstrate, we create a few `Competency` objects beforehand in a `manage.py shell` session:

```
rails_n00b = Competency.objects.create(framework='Ruby on Rails', level=1)
cake_adept = Competency.objects.create(framework='CakePHP', level=2)
spring_master = Competency.objects.create(framework='Spring', level=3)
django_guru = Competency.objects.create(framework='Django', level=4)
```

Instead of referring to those objects using primary keys (which we'd have to look up), we'd much rather just use a `(framework, level)` natural key tuple, since pairs of those will uniquely identify `Competency` objects in the database.

Here are the various ways of doing natural key relations from `JobPosting` fixtures to those pre-existing `Competency` objects:

```
jobs = Fixture(JobPosting)

# A single foreign key with a natural key tuple
jobs.add(1, title='Rails Intern', main_competency=('Ruby on Rails', 1))

# One FK, and a single M2M with a natural key tuple in a single-item list
jobs.add(2, title='Elder Django Deity', main_competency=('Django', 4),
    additional_competencies=[('Ruby on Rails', 1)])

# One FK, and several M2Ms with a list of multiple natural key tuples
jobs.add(3, title='A man of many talents', main_competency=('Spring', 3),
    additional_competencies=[('CakePHP', 2), ('Ruby on Rails', 1)]
)
```

As with normal primary key-based relations, foreign keys accept a single natural key tuple, whereas many-to-many fields require an iterable of them, even with a single item.

5.2 Raw mode

When Django deserializes fixtures, it doesn't actually call the `save()` method of the respective model classes directly. Instead, it uses `DeserializedObject`, a container class for pre-saved deserialized data, found in `django.core.serializers.base`. Here's the relevant bit from its `save()` method (as of Django 1.3):

```
# Call save on the Model baseclass directly. This bypasses any
# model-defined save. The save is also forced to be raw.
# This ensures that the data that is deserialized is literally
# what came from the file, not post-processed by pre_save/save
# methods.
models.Model.save_base(self.object, using=using, raw=True)
```

The comment probably doesn't need further explanation. `Fixture`, on the other hand, has an optional boolean `raw` parameter that defaults to `False`, meaning that in django-class-fixtures, saves are done in "normal" mode by default, using the `create()` method of the default manager of the actual model class.

Remember the *wage_slave* app from the introduction? Let's revisit it, adding some custom `save()` logic to demonstrate the use and effects of raw mode:

```
class Company(models.Model):
    name = models.CharField(max_length=100)

class Employee(models.Model):
    name = models.CharField(max_length=100)
    company = models.ForeignKey(Company)
    manager = models.ForeignKey('self', null=True)
    # New field, conditionally set to True in save()
```

```
    cog_in_the_machine = models.BooleanField(default=False)

    def save(self, *args, **kwargs):
        if ' corp' in self.company.name.lower():
            self.cog_in_the_machine = True
        super(Employee, self).save(*args, **kwargs)
```

The `save()` method of the `Employee` class examines the employing company, checking if its name contains something like "Corp." or "Corporation". If it does, then in a rather silly bit of social commentary, it deduces that this person is necessarily a dehumanized corporate drone, and sets the `cog_in_the_machine` boolean to True.

Raw mode will prevent this, however. Here's what happens with fixtures set to normal and raw mode:

```
company_fixture = Fixture(Company)
company_fixture.add(1, name='Bloatware Corporation')

employee_fixture = Fixture(Employee)
employee_fixture.add(1, name='Andy Depressant', company=company_fixture.fk(1), manager=None)
raw_employee_fixture = Fixture(Employee, raw=True)
raw_employee_fixture.add(2, name='Sadie Peon', company=company_fixture.fk(1), manager=None)
```

Once a fixture module containing those fixtures is loaded, we can check to see that in raw mode, Sadie was spared the humiliation:

```
>>> andy = Employee.objects.get(pk=1)
>>> sadie = Employee.objects.get(pk=2)
>>> andy.cog_in_the_machine
True
>>> sadie.cog_in_the_machine
False
```

Raw mode in django-class-fixtures is a feature I'd appreciate testing and feedback on. I'm not entirely sure about all the implications of it being set to either `True` or `False`; it just felt natural to leave it to False when dealing with Python code instead of, say, a JSON serialization.

If you run into use cases where `raw=True` is necessary, I'd be glad to hear about them. If you have a compelling argument why raw mode should be on by default, do tell. It's possible that when I get to work implementing a `dumpdata` override for 0.9, I'll set `raw=True` for all fixtures created programmatically with `dumpdata`, or even change the default mode, if testing produces results to support that action.

5.3 Multiple database support

I've tried to be diligent in making all of the database operations in django-class-fixtures work cleanly with multiple databases, and there are even a couple of tests for it, but I have no experience actually using Django's multiple database support in real-world environments.

Also, check out the note in *Differences in the output of loaddata* below for a possible minor caveat associated with custom database routers.

I'd appreciate more testing of this for feature parity with Django's fixture system.

5.4 Rules for fixture discovery and loading

The various forms of specifying fixtures for loading are detailed in *this example* of `TestCase.fixtures`. Options 1 and 2 there are always handled by django-class-fixtures alone, but the strings in options 3, 4 and 5 are all valid

monikers for Django fixtures, too, so there's some logic in place to determine who handles what.

The `loaddata` override looks at each fixture name it's given, and internally assigns either Django or django-class-fixtures to handle each, sometimes both. Some shadowing related to app names also takes place that may bite you in the ass if you're not careful. Here's a look at what happens with various types of strings:

File names with registered fixture extensions such as `"example_band.json"` are assigned to be handled directly by Django's `loaddata`, no questions asked. If you've written a custom serializer that uses some other format and extension than those provided by Django, the same applies for file names matching those. The "reserved" extensions are sourced from Django's serialization machinery, where custom stuff is also registered.

Strings with dots such as `"other.thing"` or `"bandaid.other_bands"` are split on the dots, and the first element of the resulting list is matched against the names of all installed apps. Of those two, `other` does not correspond to an app, so the whole string is passed on to be handled by Django's `loaddata`, matching files like `"other.thing.json"`.

`bandaid` is an installed app, however, so a fixture module called `other_bands` is searched for under the its `fixtures` subpackage. If found, it alone gets loaded and any further searches are not made on that name. If not found, a `FixtureUsageError` exception is raised.

Note: Raising an exception may seem harsh, when one could just pass the string on for Django to handle, but I wanted to make references to `appname.module_name` clearly distinct from other string forms to avoid cases where they would get passed on to Django due to a typo in the module name, and then silently ignored when files matching the name are not found, since Django does not raise errors in case of non-existent fixtures.

This means that any traditional fixture files that start with an app name followed by a dot, like `bandaid.anythingatall.json` are shadowed and not loaded, if referred to as `"bandaid.anythingatall"`. The solution is to always include the file extension in cases like these, triggering the automatic assignment for Django's `loaddata` as described above.

Arbitrary strings with no dots such as `"something"` are first checked against app names. If a match is found, no further determination takes place. All the fixture modules (but no traditional fixtures) of that app are loaded. This shadows both `"something.json"` files and fixture modules called `"something.py"`, so don't name your traditional fixtures or fixture modules the same as any apps.

Note: In this scenario, `"something.json"` is still accessible by referring to it with the .json extension and not just as `"something"`. Remember: explicit is better than implicit.

If `"something"` isn't the name of an app, it is assigned for Django to handle first. In addition, it is checked against the names of fixture modules in all valid fixture module locations. All matches are marked for loading in whatever order they are found (but always after Django has taken a crack at locating and loading traditional fixture files with that name first).

That means you can have as many fixture modules called `"something.py"` in as many `fixtures` directories or `FIXTURE_PACKAGES` locations as you like. You can also have `something.json` and `something.py` under the same directory/package, both will be loaded, Django first. No shadowing takes place at this stage.

5.5 Differences in the output of `loaddata`

Due to the process described above in *Rules for fixture discovery and loading*, when django-class-fixtures needs to fall back to Django's `loaddata`, it does so for a single fixture name parameter at a time. That is, if the parameters to `loaddata` include `"something.json"` and `"other_thing.json"`, that results in two runs of Django's `loaddata`, not one for both together. This is to ensure that the user-specified fixture ordering is preserved when mixing traditional and class-based fixtures; it won't do to just pick out all the traditional fixtures from the list and give them to the original `loaddata` command in one bunch.

The outputs from those two runs are stored and parsed for the loaded fixture counts. Those counts are then combined with the counts produced by any class-based fixtures.

Any extra messages produced by calls to Django's `loaddata` when verbosity is 2 or 3 are stored and displayed in order of appearance. This is followed by the summary row (shown as the only row with a verbosity of 1, too), which is either "No fixtures found." or "Installed **x** object(s) from **y** fixture(s)" to match what Django outputs.

Note: Django actually has a third form of the summary row: "Installed **x** object(s) (of **z**) from **y** fixture(s)." As far as I can tell by looking at the code of Django's `loaddata`, this comes into play when using multiple databases and a custom database router disallows the loading of instances of a certain model into a certain database.

As of 0.8, the **x** and **y** counts are parsed from that and correctly included in the final count, but the `loaddata` override's summary row does not parse or include the "(of **z**)" bit. I was too lazy to implement it, frankly. If you rely on it, patches are welcome, or you can hope that I find the motivation to implement it in a later version.

If you use scripts that rely on the precise output of `loaddata` (as part of Fabric deployments, for example), be sure to test them thoroughly. This is another area I'm happy to receive feedback about, there may be arguments for changing some aspect of django-class-fixtures' behaviour.

CHAPTER 6

Design Choices and Constraints

This section outlines some of the design choices made during the development of django-class-fixtures.

As a general rule, django-class-fixtures tries to work as much like Django's fixture loading mechanisms as possible or feasible. **Predictability** is the key: if you're used to working with Django's fixture system, django-class-fixtures should minimize any surprises.

6.1 Hard-coded primary keys

Using Python objects instead of serialized JSON files as the data source for the `loaddata` command, an option exists to not force the use of hard-coded primary keys. Relations from objects in class fixtures could easily be made both to other class fixtures and objects that already exist in the database prior to the loading of the fixture (such as content types created during `manage.py syncdb`).

In fact, early development versions of django-class-fixtures took this exact approach, using special objects called "relation tokens" and kwarg-based deferred object searches to build relations and their associated loading order dependency graphs.

While reasonably elegant, this approach posed several problems. First was the fact that django-class-fixtures needed to work with old-style serialized fixture files that do have hard-coded keys. Mixing hard-coded PKs with dynamically assigned primary keys is a recipe for disaster. This was circumvented by making sure that old-style fixtures always got loaded first, ensuring that class fixtures would not get overwritten and that relations didn't go haywire. Some dangers still remained, but this could be overcome with sufficient guidance of best practices in the documentation.

Another issue was that of code complexity. While hard-coding primary keys into Python code "feels dirty", doing so enables the code of django-class-fixtures to be a lot simpler and more terse, doing away with a lot of corner case handling and object identification needed for building relations.

The final nail in the coffin was the fact that if fixtures don't hard-code primary keys, then successive `loaddata` calls create duplicate objects instead of overwriting existing ones. Without a primary key, there was no reliable way of identifying if the object that a class fixture represents already existed in the database or not. While creating duplicate objects isn't "wrong" in an absolute sense, at least not much more than blindly overwriting any changes in existing objects (which is what Django does), the initial design choice to be predictable for users of old-style fixtures made it necessary to hard-code primary keys into class fixtures as well.

I wish it wasn't so, but then again, I wish for a lot of things that I can't have. If you discover a way to do it properly, do tell.

6.2 Fixture locations

Since class fixtures are Python objects, theoretically you could have a lot of flexibility with their placement, how to import them etc. But as Django expects apps to have `fixtures` subdirs containing serialized files, django-class-fixtures expects modules containing class fixtures to be there as well. The only difference is that `fixtures` must also contain an `__init__.py` module, turning the directory into a Python package. Serialized fixtures in the same directory will continue to work as before, since Django's `loaddata` only cares about the non-Python files inside it.

This "transparent" approach won't work with directories defined in `FIXTURE_DIRS`, however. It's an iterable of filesystem paths that—unlike app-based `fixtures` dirs—may not lie on Python's module search path at all, and even if they did, converting filesystem paths into importable module paths would be extremely impractical.

The `FIXTURE_DIRS` use case of having fixtures outside the apps is a valid one, so our own `FIXTURE_PACKAGES` setting aims to enable the same way of storing class fixture modules.

Just as Django's `loaddata` only looks at fixtures that are in the `fixtures` dir itself, not any subdirectories, our `loaddata` only discovers and uses fixture modules in the `fixtures` package. Nothing stops you from creating subpackages like `fixtures.regressiontests` from which you import stuff into modules contained directly under `fixtures`, of course.

CHAPTER 7

Changelog and Roadmap

7.1 Past releases

7.1.1 0.8

- Initial public release. `loaddata` support only for class-based fixtures, the necessary `dumpdata` override not yet implemented.

7.2 Planned future releases

Here's a feature roadmap for the foreseeable future. In case updates that break something are made to Django prior to 1.0, 0.x.y versions may be released. Otherwise, "micro" versions will only be released for post-1.0 bugfix releases. Prior to that, bugfixes will exist in development code only.

7.2.1 0.9

- Fix any issues discovered in the initial release.
- Implement a `dumpdata` override for serializing Django models into `Fixture` instances. You'll still have to manually direct the `dumpdata` output into .py files, same as now with, say, .json files. The output will try to be complete, including all necessary model imports, so that you don't need to modify the generated code by hand.

  Note: I know, "generated code" has a bad ring to it for Django users who know their history. However, the code inside fixture modules is so simple structurally that I'm confident the generating can be done reliably.

- Add proper class and exception documentation.
- Possibly: a helper method that formalizes the technique described in *Looping through structured data to create fixtures*.
- Possibly: integration with milkman. You'd be able to easily use `TestCase.fixtures` in your tests while still making good use of the randomly generated model instances, consolidating the creation of both random and predefined models in your fixture modules. The syntax could be something like:

  ```
  bands = Fixture(Band)
  musicians = Fixture(Musician)
  # You need the band to have specific non-random attributes
  bands.add(1, name="Arcade Water", genre="Indie")
  ```

```
# You don't care about the details of the musicians, apart from them
# being a part of a certain band; all other information is generated.
musicians.add_random(member_of=[1])
musicians.add_random(member_of=[1])
```

`add_random` would be available if *milkman* was on your Python path.

I haven't yet used *milkman* myself, and I don't know if this feature would be all that useful, but I thought I'd mention it here just in case someone thinks this would be great. If you do, let me know about it.

- Possibly: conditional loading of `Fixture` instances. Through some (I hope) elegant mechanism, allow individual `Fixture` instances to be able to determine whether some arbitrary prerequisites are met prior to being loaded. If they are not, either fail silently and keep going, or abort and raise an exception (depending on some parameter).

  This is different from what's described in *Dynamic approaches to fixture creation*, where the already possible methods of conditionally instantiating and populating `Fixture` instances are discussed. As of 0.8, there's no official way to conditionally prevent the loading process itself.

  This is just an idea at this stage, I'll need to come up with compelling use cases before putting in the work. Also, this may get a bit hairy with inter-fixture dependencies, at least in the "fail silently" case.

7.2.2 1.0

- The first version that can be recommended for use in production.
- Availability on PyPi.
- No new features are planned yet. The idea is to promote a post-0.9 trunk version to 1.0 once sufficient real-world testing has taken place and any issues discovered in 0.9 have been ironed out. Followed by bugfix/Django compatibility releases (micro versions) or small feature releases (minor versions) in perpetuity, or until...

7.2.3 2.0

- ...which is purely hypothetical at this point and may never actually see the light of day.
- Pipe dream: some type of integration with schema migrations, namely South or whatever may appear in Django core. This could take the form of having `Fixture` instantation happen against a specific generation of the model class (as frozen by the migration machinery) that is schema-compatible with the assumptions of the fixture (i.e. whatever schema was in place when the fixture was first created).

  Utterly fictional examples of `dumpdata` output after creating the fixture, with "0003_add_some_fields" being the latest migration applied that contains changes to the Band model:

  ```
  # With automatic lookup for the migration module
  bands = Fixture(Band, rev='0003_add_neat_fields')

  # With explicit import of a South-frozen model dictionary from
  # a migration module:
  from someapp.migrations.0003_add_neat_fields import models as m_0003
  bands = Fixture(m_0003['bandaid.Band'])

  bands.add(1, **stuff_as_of_0003)
  ```

  ...or something like that.

Later on, you change the schema of `Band` and create new migrations, making the fixture module outdated. `loaddata` would notice this, and there would be an automated mechanism in place to apply all post-fixture-creation migrations to all `Band` fixtures in the fixture module to modernize the schema of their contained objects before creating them. You would then get fresh `dumpdata` output with the modern schema, with which you would replace the old code in the fixture module. It could look like this, following the previous theoretical syntaxes:

```
bands = Fixture(Band, rev='0008_rename_this_to_that')

# or

from someapp.migrations.0008_rename_this_to_that import models as m_0008
bands = Fixture(m_0008['bandaid.Band'])

bands.add(1, **stuff_as_of_0008)
```

You get the picture.

This feature would rock so hard it's uncanny. No more manual updating of fixtures due to schema changes and migrations (save for piping the fresh `dumpdata` output into a .py file). Unfortunately, there are tons of open questions and hard problems here. This option hasn't yet been researched for any sort of feasibility at all. I'm mentioning it here just in case some enterprising Djangonaut is divinely inspired by the idea and decides to implement it. I'm not sure I'll ever have the skill or patience for it.