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.

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)
```

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)
```

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.

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",
            ),
        ]


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"],
                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:
        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.