Contents

[Django Making Queries 4](#_Toc8219425)

[Making queries 4](#_Toc8219426)

[Django QuerySet API 9](#_Toc8219427)

[When QuerySets are evaluated 9](#_Toc8219428)

[Pickling QuerySets 10](#_Toc8219429)

[PANDAS-DJANGO MODEL 10](#_Toc8219430)

[Django ORM to insert data 10](#_Toc8219431)

[import data With external script 10](#_Toc8219432)

[Save Pandas DF as Django Model 10](#_Toc8219433)

[Insert 1,000 records into SQLite table using Django 11](#_Toc8219434)

[DJANGO-PANDAS 12](#_Toc8219435)

[Io.py 12](#_Toc8219436)

[**def** to\_fields(qs, fieldnames): 12](#_Toc8219437)

[def is\_values\_queryset(qs): 12](#_Toc8219438)

[**def** read\_frame(qs, fieldnames=(), index\_col=None, coerce\_float=False, verbose=True, datetime\_index=False): 12](#_Toc8219439)

[Managers.py 13](#_Toc8219440)

[**class** PassThroughManagerMixin(object): 13](#_Toc8219441)

[**def** \_\_init\_\_(self, queryset\_cls=None): 13](#_Toc8219442)

[**def** \_\_getattr\_\_(self, name): 14](#_Toc8219443)

[**def** \_\_dir\_\_(self): 14](#_Toc8219444)

[**def** get\_queryset(self): 14](#_Toc8219445)

[**def** for\_queryset\_class(cls, queryset\_cls): 14](#_Toc8219446)

[**class** PassThroughManager(PassThroughManagerMixin, models.Manager): 14](#_Toc8219447)

[**def** create\_pass\_through\_manager\_for\_queryset\_class(base, queryset\_cls): 14](#_Toc8219448)

[**class** \_PassThroughManager(base): 14](#_Toc8219449)

[**def** get\_queryset(self): 14](#_Toc8219450)

[**class** DataFrameQuerySet(QuerySet): 14](#_Toc8219451)

[def to\_pivot\_table(self, fieldnames=(), verbose=True, values=None, rows=None, cols=None, aggfunc='mean', fill\_value=None, margins=False, dropna=True, coerce\_float=True): 14](#_Toc8219452)

[**def** to\_timeseries(self, fieldnames=(), verbose=True, index=None, storage='wide', values=None, pivot\_columns=None, freq=None, coerce\_float=True, rs\_kwargs=None): 15](#_Toc8219453)

[**def** to\_dataframe(self, fieldnames=(), verbose=True, index=None, 16](#_Toc8219454)

[Utils.py 17](#_Toc8219455)

[**def** get\_model\_name(model): 17](#_Toc8219456)

[**def** replace\_from\_choices(choices): 17](#_Toc8219457)

[**def** get\_base\_cache\_key(model): 17](#_Toc8219458)

[**def** get\_cache\_key(obj): 17](#_Toc8219459)

[**def** invalidate(obj): 17](#_Toc8219460)

[**def** invalidate\_signal\_handler(sender, \*\*kwargs): 17](#_Toc8219461)

[**def** replace\_pk(model): 17](#_Toc8219462)

[**def** get\_cache\_key\_from\_pk(pk): 17](#_Toc8219463)

[**def** inner(pk\_series): 17](#_Toc8219464)

[**def** build\_update\_functions(fieldnames, fields): 18](#_Toc8219465)

[**def** update\_with\_verbose(df, fieldnames, fields): 18](#_Toc8219466)

[**def** get\_related\_model(field): 18](#_Toc8219467)

[DJANGO DATA WIZARD 18](#_Toc8219468)

[Data\_WIZARD.\_\_init\_\_.py 18](#_Toc8219469)

[**def** autodiscover(): 18](#_Toc8219470)

[**def** init\_backend(): 18](#_Toc8219471)

[**def** register(\*args, \*\*kwargs): 19](#_Toc8219472)

[**def** set\_loader(\*args, \*\*kwargs): 19](#_Toc8219473)

[Data\_WIZARD.Admin.py 19](#_Toc8219474)

[**class** FixedTabularInline(admin.TabularInline): 19](#_Toc8219475)

[**def** has\_add\_permission(self, request, obj): 19](#_Toc8219476)

[**class** RangeInline(admin.TabularInline): 19](#_Toc8219477)

[**class** RecordInline(FixedTabularInline): 19](#_Toc8219478)

[**class** RunLogInline(FixedTabularInline): 19](#_Toc8219479)

[**class** RunAdmin(admin.ModelAdmin): 19](#_Toc8219480)

[**class** IdentifierAdmin(admin.ModelAdmin): 20](#_Toc8219481)

[**def** start\_data\_wizard(modeladmin, request, queryset): 20](#_Toc8219482)

[Data\_WIZARD.Apps.py 20](#_Toc8219483)

[**class** WizardConfig(AppConfig): 20](#_Toc8219484)

[**def** ready(self): 20](#_Toc8219485)

[Data\_WIZARD.compat.py 20](#_Toc8219486)

[Data\_WIZARD.fields.py 21](#_Toc8219487)

[**class** DateTimeSplitField(serializers.Field): 21](#_Toc8219488)

[**def** process\_date\_FIXME(meta\_field, meta\_datatype, meta\_key, val, obj): 21](#_Toc8219489)

[**def** process\_date\_part(new\_val, old\_val, part): 21](#_Toc8219490)

[Data\_WIZARD.loaders.py 22](#_Toc8219491)

[**class** BaseLoader(object): 22](#_Toc8219492)

[**def** \_\_init\_\_(self, run): 22](#_Toc8219493)

[**def** get\_serializer\_name(self): 22](#_Toc8219494)

[**def** content\_object(self): 22](#_Toc8219495)

[**def** load\_io\_options(self): 22](#_Toc8219496)

[**class** FileLoader(BaseLoader): 22](#_Toc8219497)

[**def** file(self): 22](#_Toc8219498)

[**def** load\_io(self): 22](#_Toc8219499)

[**class** URLLoader(BaseLoader): 23](#_Toc8219500)

[**def** url(self): 23](#_Toc8219501)

[**def** load\_io(self): 23](#_Toc8219502)

[Data\_WIZARD.models.py 23](#_Toc8219503)

[**class** Run(models.Model): 23](#_Toc8219504)

[**def** \_\_str\_\_(self): 23](#_Toc8219505)

[**def** get\_absolute\_url(self): 23](#_Toc8219506)

[**def** save(self, \*args, \*\*kwargs): 23](#_Toc8219507)

[**def** load\_io(self): 23](#_Toc8219508)

[backend=None, user=None): 24](#_Toc8219509)

[**def** serializer\_label(self): 24](#_Toc8219510)

[**def** get\_serializer(self): 24](#_Toc8219511)

[**def** get\_serializer\_options(self): 24](#_Toc8219512)

[**def** already\_parsed(self): 24](#_Toc8219513)

[**def** add\_event(self, name): 24](#_Toc8219514)

[**def** last\_update(self): 24](#_Toc8219515)

[**class** Meta: 24](#_Toc8219516)

[**class** RunLog(models.Model): 24](#_Toc8219517)

[**def** \_\_str\_\_(self): 24](#_Toc8219518)

[**class** Meta: 24](#_Toc8219519)

[**class** Identifier(models.Model): 24](#_Toc8219520)

[**def** \_\_str\_\_(self): 25](#_Toc8219521)

[**def** type(self): 25](#_Toc8219522)

[**def** type\_label(self): 25](#_Toc8219523)

[**def** mapping\_label(self): 25](#_Toc8219524)

[**def** serializer\_label(self): 26](#_Toc8219525)

[**class** Range(models.Model): 26](#_Toc8219526)

[**def** save(self, \*args, \*\*kwargs): 26](#_Toc8219527)

[**def** \_\_str\_\_(self): 26](#_Toc8219528)

[**class** Meta: 27](#_Toc8219529)

[**class** Record(models.Model): 27](#_Toc8219530)

[**def** \_\_str\_\_(self): 27](#_Toc8219531)

[**class** Meta: 27](#_Toc8219532)

[Data\_WIZARD.registry.py 27](#_Toc8219533)

[**class** Registry(object): 27](#_Toc8219534)

[**def** get\_class\_name(self, serializer): 27](#_Toc8219535)

[**def** register(self, name, serializer=None): 27](#_Toc8219536)

[**def** create\_serializer(self, model): 28](#_Toc8219537)

[**def** get\_serializers(self): 28](#_Toc8219538)

[**def** get\_serializer\_name(self, name): 28](#_Toc8219539)

[**def** get\_serializer(self, name): 28](#_Toc8219540)

[**def** get\_serializer\_options(self, name): 28](#_Toc8219541)

[**def** get\_choices(self): 28](#_Toc8219542)

[**def** set\_loader(self, model, loader\_name): 28](#_Toc8219543)

[**def** get\_loader\_name(self, model): 28](#_Toc8219544)

[**def** get\_loader(self, loader\_name): 28](#_Toc8219545)

[Data\_WIZARD.rest.py 29](#_Toc8219546)

[**class** CurrentUserDefault(serializers.CurrentUserDefault): 29](#_Toc8219547)

[**def** \_\_call\_\_(self): 29](#_Toc8219548)

[**class** RunSerializer(ModelSerializer, wizard.RunSerializer): 29](#_Toc8219549)

[**class** Meta: 29](#_Toc8219550)

[**class** RecordSerializer(wizard.RecordSerializer): 29](#_Toc8219551)

[**class** RunViewSet(ModelViewSet, wizard.RunViewSet): 29](#_Toc8219552)

[**def** user\_filter(qs, request): 29](#_Toc8219553)

[Data\_WIZARD.serializers.py 29](#_Toc8219554)

[**class** ContentTypeIdField(serializers.RelatedField): 30](#_Toc8219555)

[**def** to\_internal\_value(self, data): 30](#_Toc8219556)

[**def** to\_representation(self, value): 30](#_Toc8219557)

[**class** RunSerializer(serializers.ModelSerializer): 30](#_Toc8219558)

[**def** get\_fields(self): 30](#_Toc8219559)

[**class** Meta: 30](#_Toc8219560)

[**class** RecordSerializer(serializers.ModelSerializer): 30](#_Toc8219561)

[**def** get\_row(self, instance): 30](#_Toc8219562)

[**def** get\_object\_label(self, instance): 30](#_Toc8219563)

[**def** get\_object\_url(self, instance): 30](#_Toc8219564)

[**class** Meta: 31](#_Toc8219565)

[Data\_WIZARD.settings.py 31](#_Toc8219566)

[**def** get\_setting(name): 31](#_Toc8219567)

[**def** import\_from\_string(path, setting\_name): 31](#_Toc8219568)

[**def** import\_setting(name): 31](#_Toc8219569)

[Data\_WIZARD.signals.py 31](#_Toc8219570)

[Data\_WIZARD.tasks.py 31](#_Toc8219571)

[**def** get\_ct(model): 32](#_Toc8219572)

[**def** ctid(ct): 32](#_Toc8219573)

[**def** metaname(cls): 32](#_Toc8219574)

[**def** get\_id(obj, field): 32](#_Toc8219575)

[**def** send\_progress(sender, run): 32](#_Toc8219576)

[**def** lookuprun(fn): 32](#_Toc8219577)

[**def** auto\_import(run, user): 32](#_Toc8219578)

[**def** get\_attribute\_field(field): 33](#_Toc8219579)

[**def** compute\_attr\_field(value\_field, attr\_name): 33](#_Toc8219580)

[**def** get\_choices(run): 34](#_Toc8219581)

[**def** load\_fields(serializer, group\_name, 34](#_Toc8219582)

[**def** get\_choice\_groups(run): 35](#_Toc8219583)

[**def** get\_choice\_ids(run): 35](#_Toc8219584)

[**def** read\_columns(run, user=None): 35](#_Toc8219585)

[**def** get\_columns(run): 35](#_Toc8219586)

[**def** get\_lookup\_columns(run): 36](#_Toc8219587)

[**def** load\_columns(run): 36](#_Toc8219588)

[**def** get\_range\_value(table, rng, scol, ecol): 36](#_Toc8219589)

[**def** parse\_columns(run): 37](#_Toc8219590)

[**def** parse\_column(run, name, \*\*kwargs): 37](#_Toc8219591)

[**def** update\_columns(run, user, post={}): 38](#_Toc8219592)

[**def** read\_row\_identifiers(run, user=None): 38](#_Toc8219593)

[**def** parse\_row\_identifiers(run): 38](#_Toc8219594)

[**def** load\_row\_identifiers(run): 39](#_Toc8219595)

[**def** update\_row\_identifiers(run, user, post={}): 40](#_Toc8219596)

[**def** import\_data(run, user): 40](#_Toc8219597)

[**def** do\_import(run, user): 41](#_Toc8219598)

[**def** \_do\_import(run, user): 41](#_Toc8219599)

[**def** import\_row(run, i, row, instance\_globals, matched): 42](#_Toc8219600)

[**def** save\_value(col, val, obj): 43](#_Toc8219601)

[**def** save\_attribute\_value(col, val, obj): 43](#_Toc8219602)

[Data\_WIZARD.test.py 43](#_Toc8219603)

[**class** WizardTestCase(APITransactionTestCase): 43](#_Toc8219604)

[**def** \_fixture\_teardown(self): 44](#_Toc8219605)

[**def** setUp(self): 44](#_Toc8219606)

[**def** mkurl(self, run, action): 44](#_Toc8219607)

[**def** get\_url(self, run, action, params={}): 44](#_Toc8219608)

[**def** post\_url(self, run, action, post): 44](#_Toc8219609)

[**def** wait(self, run, action): 44](#_Toc8219610)

[**def** create\_identifier(self, name, field, value=None, attr\_id=None): 44](#_Toc8219611)

[**def** upload\_file(self, filename, skip\_serializer=False): 44](#_Toc8219612)

[**def** download\_url(self, url, skip\_serializer=False): 45](#_Toc8219613)

[**def** set\_serializer(self, run): 45](#_Toc8219614)

[**def** check\_columns(self, run, expect\_columns, expect\_unknown): 45](#_Toc8219615)

[**def** update\_columns(self, run, mappings): 45](#_Toc8219616)

[**def** check\_row\_identifiers(self, run, expect\_identifiers, expect\_unknown): 46](#_Toc8219617)

[**def** update\_row\_identifiers(self, run, mappings): 46](#_Toc8219618)

[**def** start\_import(self, run, expect\_skipped): 46](#_Toc8219619)

[**def** auto\_import(self, run, expect\_input\_required=False): 47](#_Toc8219620)

[**def** assert\_status(self, run, expect\_count): 47](#_Toc8219621)

[**def** assert\_ranges(self, run, expect\_ranges): 47](#_Toc8219622)

[**def** assert\_records(self, run, expect\_records): 47](#_Toc8219623)

[**def** assert\_log(self, run, expect\_log): 47](#_Toc8219624)

[**def** assert\_urls(self, run, urltemplate): 48](#_Toc8219625)

[Data\_WIZARD.urls.py 48](#_Toc8219626)

[Data\_WIZARD.views.py 48](#_Toc8219627)

[**class** PageNumberPagination(PageNumberPagination): 48](#_Toc8219628)

[**class** RunViewSet(ModelViewSet): 48](#_Toc8219629)

[**def** backend(self): 48](#_Toc8219630)

[**def** template\_name(self): 48](#_Toc8219631)

[**def** get\_renderers(self): 48](#_Toc8219632)

[**def** status(self, request, \*args, \*\*kwargs): 49](#_Toc8219633)

[**def** run\_task(self, name, use\_async=False, post=None): 49](#_Toc8219634)

[**def** retrieve\_and\_run(self, task\_name, use\_async=False, post=None): 49](#_Toc8219635)

[**def** serializers(self, request, \*args, \*\*kwargs): 49](#_Toc8219636)

[**def** updateserializer(self, request, \*args, \*\*kwargs): 49](#_Toc8219637)

[**def** columns(self, request, \*args, \*\*kwargs): 49](#_Toc8219638)

[**def** updatecolumns(self, request, \*args, \*\*kwargs): 49](#_Toc8219639)

[**def** ids(self, request, \*args, \*\*kwargs): 50](#_Toc8219640)

[**def** updateids(self, request, \*args, \*\*kwargs): 50](#_Toc8219641)

[**def** data(self, request, \*args, \*\*kwargs): 50](#_Toc8219642)

[**def** auto(self, request, \*args, \*\*kwargs): 50](#_Toc8219643)

[**def** records(self, request, \*args, \*\*kwargs): 50](#_Toc8219644)

[Data\_WIZARD.Management.commands.runwizard.py 50](#_Toc8219645)

[**class** Command(BaseCommand): 50](#_Toc8219646)

[**def** add\_arguments(self, parser): 50](#_Toc8219647)

[**def** handle(self, \*args, \*\*options): 50](#_Toc8219648)

[Data\_WIZARD.Sources.admin.py 51](#_Toc8219649)

[Data\_WIZARD.Sources.apps.py 51](#_Toc8219650)

[**class** SourcesConfig(AppConfig): 51](#_Toc8219651)

[Data\_WIZARD.Sources.models.py 51](#_Toc8219652)

[**class** FileSource(models.Model): 51](#_Toc8219653)

[**def** \_\_str\_\_(self): 52](#_Toc8219654)

[**class** URLSource(models.Model): 52](#_Toc8219655)

[**def** \_\_str\_\_(self): 52](#_Toc8219656)

[Data\_WIZARD.Sources.rest.py 52](#_Toc8219657)

[Data\_WIZARD.Sources.wizard.py 52](#_Toc8219658)

[DJANGO WAREHOUSE 52](#_Toc8219659)

[Manage.py 52](#_Toc8219660)

[Warehouse.settings.py 52](#_Toc8219661)

[Warehouse.urls.py 54](#_Toc8219662)

[Warehouse.wsgi.py 54](#_Toc8219663)

[Warehouse.Reporting.admin.py 54](#_Toc8219664)

[**class** ReadOnlyAdmin(admin.ModelAdmin): 55](#_Toc8219665)

[**def** get\_readonly\_fields(self, request, obj=None): 55](#_Toc8219666)

[**class** ShapeInline(admin.TabularInline): 55](#_Toc8219667)

[**def** link(self, instance): 55](#_Toc8219668)

[**class** AssetAdmin(ReadOnlyAdmin): 55](#_Toc8219669)

[**def** storage\_size(self, obj): 55](#_Toc8219670)

[**def** all\_sites(self, obj): 55](#_Toc8219671)

[**class** ShapeAdmin(ReadOnlyAdmin): 55](#_Toc8219672)

[**class** AssetInline(admin.TabularInline): 55](#_Toc8219673)

[**def** link(self, instance): 55](#_Toc8219674)

[**class** SyncRunAdmin(ReadOnlyAdmin): 55](#_Toc8219675)

[**def** remaining(self, obj): 56](#_Toc8219676)

[Warehouse.Reporting.models.py 56](#_Toc8219677)

[**def** load\_json(raw): 56](#_Toc8219678)

[**def** dump\_json(obj): 56](#_Toc8219679)

[**def** GET(url, \*\*kwargs): 56](#_Toc8219680)

[**class** Site(models.Model): 56](#_Toc8219681)

[**def** \_\_unicode\_\_(self): 57](#_Toc8219682)

[**class** SyncRun(models.Model): 57](#_Toc8219683)

[**def** \_\_unicode\_\_(self): 57](#_Toc8219684)

[**class** DamAssetManager(models.Manager): 57](#_Toc8219685)

[**def** get\_queryset(self): 57](#_Toc8219686)

[**class** ReportableModelMixin(models.Model): 57](#_Toc8219687)

[**class** Meta: 57](#_Toc8219688)

[**class** Asset(ReportableModelMixin): 57](#_Toc8219689)

[**def** \_\_unicode\_\_(self): 57](#_Toc8219690)

[**def** update\_size(self): 57](#_Toc8219691)

[**class** Download(ReportableModelMixin): 57](#_Toc8219692)

[**class** Shape(ReportableModelMixin): 58](#_Toc8219693)

[**def** \_\_unicode\_\_(self): 58](#_Toc8219694)

[**def** get\_asset(url): 58](#_Toc8219695)

[**def** get\_shapes\_for\_asset(asset\_id): 58](#_Toc8219696)

[**def** get\_shape(url): 58](#_Toc8219697)

[**def** perform\_search(runas, filters = None): 58](#_Toc8219698)

[**def** get\_offsets(emitted, skip, precache): 58](#_Toc8219699)

[**def** asset\_iterator(zonza\_site, skip): 58](#_Toc8219700)

[Warehouse.Reporting.reports.py 59](#_Toc8219701)

[**class** CSVReport(object): 59](#_Toc8219702)

[**def** \_\_init\_\_(self, start=None, end=None, site=None): 59](#_Toc8219703)

[**def** write\_headings(self): 59](#_Toc8219704)

[**def** serialise(self): 60](#_Toc8219705)

[**class** UsageReport(CSVReport): 60](#_Toc8219706)

[**def** queryset(self): 60](#_Toc8219707)

[**def** serialise(self): 60](#_Toc8219708)

[**class** IngestShapeReport(CSVReport): 60](#_Toc8219709)

[**def** queryset(self): 61](#_Toc8219710)

[**def** serialise(self): 61](#_Toc8219711)

[**class** IngestReport(CSVReport): 61](#_Toc8219712)

[**def** queryset(self): 61](#_Toc8219713)

[**def** serialise(self): 61](#_Toc8219714)

[**class** DownloadReport(CSVReport): 61](#_Toc8219715)

[**def** queryset(self): 62](#_Toc8219716)

[**def** serialise(self): 62](#_Toc8219717)

[Warehouse.Reporting.tasks.py 62](#_Toc8219718)

[Warehouse.Reporting.urls.py 62](#_Toc8219719)

[Warehouse.Reporting.utils.py 63](#_Toc8219720)

[**def** filesizeformat(num): 63](#_Toc8219721)

[**def** bytes\_to\_gb(orig\_bytes): 63](#_Toc8219722)

[**def** get\_days\_elapsed(created, deleted, start, end): 63](#_Toc8219723)

[**def** get\_usage(shape\_row, start, end): 63](#_Toc8219724)

[Warehouse.Reporting.views.py 63](#_Toc8219725)

[**def** domain(request, domain): 63](#_Toc8219726)

[**def** download\_csv(domain): 64](#_Toc8219727)

[**def** dashboard(request): 64](#_Toc8219728)

Django Making Queries

# Making queries

Once you’ve created your [data models](https://docs.djangoproject.com/en/2.1/topics/db/models/), Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. This document explains how to use this API. Refer to the [data model reference](https://docs.djangoproject.com/en/2.1/ref/models/) for full details of all the various model lookup options. Models in this guide for Weblog application:

**from** **django.db** **import** models

**class** Blog(models.Model):

name = models.CharField(max\_length=100)

tagline = models.TextField()

**def** \_\_str\_\_(self):

**return** self.name

**class** Author(models.Model):

name = models.CharField(max\_length=200)

email = models.EmailField()

**def** \_\_str\_\_(self):

**return** self.name

**class** Entry(models.Model):

blog = models.ForeignKey(Blog, on\_delete=models.CASCADE)

headline = models.CharField(max\_length=255)

body\_text = models.TextField()

pub\_date = models.DateField()

mod\_date = models.DateField()

authors = models.ManyToManyField(Author)

n\_comments = models.IntegerField()

n\_pingbacks = models.IntegerField()

rating = models.IntegerField()

**def** \_\_str\_\_(self):

**return** self.headline

**Creating objects**

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table. To create an object, instantiate it using keyword arguments to the model class, then call [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save) to save it to the database. Assuming models live in a file mysite/blog/models.py:

**from** **blog.models** **import** Blog

b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')

b.save()

This performs an INSERT SQL statement behind the scenes. Django doesn’t hit the database until you explicitly call [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save). The [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save) method has no return value. See also: [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save) takes a number of advanced options not described here. See the documentation for [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save) for complete details. To create and save an object in a single step, use the [create()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.create) method.

**Saving changes to objects**

To save changes to an object that’s already in the database, use [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save). Given a Blog instance b5 that has already been saved to the database, this example changes its name and updates its record in the database:

b5.name = 'New name'

b5.save()

This performs an UPDATE SQL statement behind the scenes. Django doesn’t hit the database until you explicitly call [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save).

**Saving ForeignKey and ManyToManyField fields**

Updating a [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) field works the same way as saving a normal field – simply assign an object of the right type to the field in question. This example updates the blog attribute of an Entry instance entry, assuming instances of Entry and Blog already saved to database (so we can retrieve them):

**from** **blog.models** **import** Blog, Entry

entry = Entry.objects.get(pk=1)

cheese\_blog = Blog.objects.get(name="Cheddar Talk")

entry.blog = cheese\_blog

entry.save()

Updating a [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) works a little differently – use the [add()](https://docs.djangoproject.com/en/2.1/ref/models/relations/#django.db.models.fields.related.RelatedManager.add) method on the field to add a record to the relation. This example adds the Author instance joe to the entry object:

**from** **blog.models** **import** Author

joe = Author.objects.create(name="Joe")

entry.authors.add(joe)

To add multiple records to a [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) in one go, include multiple arguments in the call to [add()](https://docs.djangoproject.com/en/2.1/ref/models/relations/#django.db.models.fields.related.RelatedManager.add), like this:

john = Author.objects.create(name="John")

paul = Author.objects.create(name="Paul")

george = Author.objects.create(name="George")

ringo = Author.objects.create(name="Ringo")

entry.authors.add(john, paul, george, ringo)

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

**Retrieving objects**

To retrieve objects from your database, construct a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) via a [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) on your model class. A [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) represents a collection of objects from your database. It can have zero, one or many *filters*. Filters narrow down the query results based on the given parameters. In SQL terms, a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT. You get a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) by using your model’s [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager). Each model has at least one [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager), and it’s called [objects](https://docs.djangoproject.com/en/2.1/ref/models/class/#django.db.models.Model.objects) by default. Access it via the model class:

Blog.objects

<django.db.models.manager.Manager object at ...>

b = Blog(name='Foo', tagline='Bar')

b.objects

Traceback:

...

**AttributeError**: "Manager isn't accessible via Blog instances."

**Note**: Managers are accessible only via model classes, rather than from model instances, to enforce a separation between “table-level” operations and “record-level” operations. The [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) is the main source of QuerySets for a model. For example, Blog.objects.all() returns a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) that contains all Blog objects in the database.

**Retrieving all objects**

The simplest way to retrieve objects from a table is to get all of them. To do this, use the [all()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.all) method on a [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager):

all\_entries = Entry.objects.all()

The [all()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.all) method returns a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) of all the objects in the database.

**Retrieving specific objects with filters**

The [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) returned by [all()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.all) describes all objects in the database table. Usually, though, you’ll need to select only a subset of the complete set of objects. To create such a subset, you refine the initial [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet), adding filter conditions. The two most common ways to refine a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) are:

filter(\*\*kwargs)

Returns new [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) containing objects matching given lookup parameters.

exclude(\*\*kwargs)

Returns new [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) containing objects *not* matching given lookup params.

The lookup parameters (\*\*kwargs in the above function definitions) should be in the format described in [Field lookups](https://docs.djangoproject.com/en/2.1/topics/db/queries/#field-lookups) below. For example, to get a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) of blog entries from the year 2006, use [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) like so:

Entry.objects.filter(pub\_date\_\_year=2006)

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

Entry.objects.all().filter(pub\_date\_\_year=2006)

**Chaining filters**

The result of refining a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) is itself a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet), so it’s possible to chain refinements together. For example:

Entry.objects.filter(

... headline\_\_startswith='What'

... ).exclude(

... pub\_date\_\_gte=datetime.date.today()

... ).filter(

... pub\_date\_\_gte=datetime.date(2005, 1, 30)

... )

This takes the initial [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) containing all entries with a headline that starts with “What”, that were published between January 30, 2005, and the current day.

**Filtered QuerySets are unique**

Each time you refine a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet), you get a brand-new [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) that is in no way bound to the previous [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet). Each refinement creates a separate and distinct [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) that can be stored, used and reused. Example:

q1 = Entry.objects.filter(headline\_\_startswith="What")

q2 = q1.exclude(pub\_date\_\_gte=datetime.date.today())

q3 = q1.filter(pub\_date\_\_gte=datetime.date.today())

These three QuerySets are separate. The first is a base [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) containing all entries that contain a headline starting with “What”. The second is a subset of the first, with an additional criteria that excludes records whose pub\_date is today or in the future. The third is a subset of the first, with an additional criteria that selects only the records whose pub\_date is today or in the future. The initial [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) (q1) is unaffected by the refinement process.

**QuerySets are lazy**

QuerySets are lazy – the act of creating a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) doesn’t involve any database activity. You can stack filters together all day long, and Django won’t actually run the query until the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) is *evaluated*. Example:

q = Entry.objects.filter(headline\_\_startswith="What")

q = q.filter(pub\_date\_\_lte=datetime.date.today())

q = q.exclude(body\_text\_\_icontains="food")

**print**(q)

Though this looks like three database hits, in fact it hits the database only once, at the last line (print(q)). In general, the results of a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) aren’t fetched from the database until you “ask” for them. When you do, the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) is *evaluated* by accessing the database. For more details on exactly when evaluation takes place, see [When QuerySets are evaluated](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#when-querysets-are-evaluated).

**Retrieving a single object with get()**

[filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) will always give you a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet), even if only a single object matches the query - in this case, it will be a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) containing a single element. If you know there is only one object that matches your query, you can use the [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get) method on a [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) which returns the object directly:

one\_entry = Entry.objects.get(pk=1)

You can use any query expression with [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get), just like with [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) - again, see [Field lookups](https://docs.djangoproject.com/en/2.1/topics/db/queries/#field-lookups) below. Note that there is a difference between using [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get), and using [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) with a slice of [0]. If there are no results that match the query, [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get) will raise a DoesNotExist exception. This exception is an attribute of the model class that the query is being performed on - so in the code above, if there is no Entry object with a primary key of 1, Django will raise Entry.DoesNotExist. Django will complain if more than one item matches the [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get) query. In this case, it will raise [MultipleObjectsReturned](https://docs.djangoproject.com/en/2.1/ref/exceptions/#django.core.exceptions.MultipleObjectsReturned), which again is an attribute of the model class itself.

**Other QuerySet methods**

Most of the time you’ll use [all()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.all), [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get), [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) and [exclude()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exclude) when you need to look up objects from the database. However, that’s far from all there is; see the [QuerySet API Reference](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#queryset-api) for a complete list of all the various [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) methods.

**Limiting QuerySets**

Use a subset of Python’s array-slicing syntax to limit your [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) to certain results. This is equivalent to SQL’s LIMIT and OFFSET clauses. For example, this returns the first 5 objects (LIMIT 5):

Entry.objects.all()[:5]

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

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

Negative indexing (Entry.objects.all()[-1]) is not supported. Slicing a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) returns a new [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) – it doesn’t evaluate the query. An exception is if you use the “step” parameter of Python slice syntax. For example, this would actually execute the query in order to return a list of every *second* object of the first 10:

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

Further filtering or ordering of a sliced queryset is prohibited due to the ambiguous nature of how that might work. To retrieve a *single* object rather than a list (e.g. SELECT foo FROM bar LIMIT 1), use a simple index instead of a slice. For example, this returns the first Entry in the database, after ordering entries alphabetically by headline:

Entry.objects.order\_by('headline')[0]

This is roughly equivalent to:

Entry.objects.order\_by('headline')[0:1].get()

The first of these will raise IndexError while the second will raise DoesNotExist if no objects match the given criteria. See [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get) for details.

**Field lookups**

Field lookups are how you specify the meat of an SQL WHERE clause. They’re specified as keyword arguments to the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) methods [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter), [exclude()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exclude) and [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get). Basic lookups keyword arguments take the form field\_\_lookuptype=value. (That’s a double-underscore). For example:

Entry.objects.filter(pub\_date\_\_lte='2006-01-01')

translates (roughly) into the following SQL:

SELECT \* FROM blog\_entry WHERE pub\_date <= '2006-01-01';

How this is possible? Python has the ability to define functions that accept arbitrary name-value arguments whose names and values are evaluated at runtime. (See [Keyword Arguments](https://docs.python.org/3/tutorial/controlflow.html#tut-keywordargs) in the official Python tutorial). The field specified in a lookup has to be the name of a model field. Exception: in case of a [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) you can specify the field name suffixed with \_id. The value parameter is expected to contain the raw value of the foreign model’s primary key. Example:

Entry.objects.filter(blog\_id=4)

With an invalid keyword argument, a lookup function will raise TypeError. The database API supports about two dozen lookup types; a complete reference can be found in the [field lookup reference](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#field-lookups). To give you a taste of what’s available, here’s some of the more common lookups you’ll probably use:

[exact](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-exact)

An “exact” match. For example:

Entry.objects.get(headline\_\_exact="Cat bites dog")

Would generate SQL along these lines:

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

If you don’t provide a lookup type – that is, if your keyword argument doesn’t contain a double underscore – the lookup type is assumed to be exact. For example, the following two statements are equivalent:

Blog.objects.get(id\_\_exact=14) *# Explicit form*

Blog.objects.get(id=14) *# \_\_exact is implied*

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

[iexact](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-iexact)

A case-insensitive match. So, the query:

Blog.objects.get(name\_\_iexact="beatles blog")

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

[contains](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-contains)

Case-sensitive containment test. For example:

Entry.objects.get(headline\_\_contains='Lennon')

Roughly translates to this SQL:

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

Note this will match the headline 'Today Lennon honored' but not 'today lennon honored'. There’s also a case-insensitive version, [icontains](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-icontains).

[startswith](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-startswith), [endswith](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-endswith)

Starts-with and ends-with search, respectively. There are also case-insensitive versions called [istartswith](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-istartswith) and [iendswith](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-iendswith).

A complete reference can be found in the [field lookup reference](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#field-lookups).

**Lookups that span relationships**

Django offers a powerful and intuitive way to “follow” relationships in lookups, taking care of the SQL JOINs for you automatically, behind the scenes. To span a relationship, just use the field name of related fields across models, separated by double underscores, until you get to the field you want. This example retrieves all Entry objects with a Blog whose name is 'Beatles Blog':

Entry.objects.filter(blog\_\_name='Beatles Blog')

This spanning can be deep. It works backwards, too. To refer to a “reverse” relationship, just use the lowercase name of the model. This example retrieves all Blog objects with min one Entry whose headline contains 'Lennon':

Blog.objects.filter(entry\_\_headline\_\_contains='Lennon')

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

Blog.objects.filter(entry\_\_authors\_\_name='Lennon')

(if there was a related Author model), if there was no author associated with an entry, it would be treated as if there was also no name attached, rather than raising an error because of the missing author. The only case where it might be confusing is if you are using [isnull](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#std:fieldlookup-isnull). Thus:

Blog.objects.filter(entry\_\_authors\_\_name\_\_isnull=True)

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

Blog.objects.filter(entry\_\_authors\_\_isnull=False, entry\_\_authors\_\_name\_\_isnull=True)

**Spanning multi-valued relationships**

When you are filtering an object based on a [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) or a reverse [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey), there are two different sorts of filter. Consider the Blog/Entry relationship (Blog to Entry is a one-to-many relation). We might be interested in finding blogs that have an entry which has both *“Lennon”* in the headline and was published in 2008. Or we might want to find blogs that have an entry with *“Lennon”* in the headline as well as an entry that was published in 2008. Since there are multiple entries associated with a single Blog, both of these queries are possible and make sense in some situations. The same type of situation arises with a [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField). For example, if an Entry has a [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) called tags, we might want to find entries linked to tags called *“music”* and *“bands”* or we might want an entry that contains a tag with a name of *“music”* and a status of *“public”*. To handle both of these situations, Django has a consistent way of processing [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) calls. Everything inside a single [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) call is applied simultaneously to filter out items matching all those requirements. Successive [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) calls further restrict the set of objects, but for multi-valued relations, they apply to any object linked to the primary model, not necessarily those objects that were selected by an earlier [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) call. That may sound a bit confusing, so hopefully an example will clarify. To select all blogs that contain entries with both *“Lennon”* in the headline and that were published in 2008 (the same entry satisfying both conditions), we would write:

Blog.objects.filter(entry\_\_headline\_\_contains='Lennon', entry\_\_pub\_date\_\_year=2008

To select all blogs that contain an entry with *“Lennon”* in the headline **as well as** an entry that was published in 2008, we would write:

Blog.objects.filter(entry\_\_headline\_\_contains='Lennon').filter(entry\_\_pub\_date\_\_year=2008)

Suppose there is only one blog that had both entries containing *“Lennon”* and entries from 2008, but that none of the entries from 2008 contained *“Lennon”*. The first query would not return any blogs, but the second query would return that one blog. In the second example, the first filter restricts the queryset to all those blogs linked to entries with *“Lennon”* in the headline. The second filter restricts the set of blogs *further* to those that are also linked to entries that were published in 2008. The entries selected by the second filter may or may not be the same as the entries in the first filter. We are filtering the Blog items with each filter statement, not the Entry items. **Note**: The behavior of [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter) for queries that span multi-value relationships, as described above, is not implemented equivalently for [exclude()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exclude). Instead, the conditions in a single [exclude()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exclude) call will not necessarily refer to the same item. For example, the following query would exclude blogs that contain *both* entries with *“Lennon”* in the headline *and* entries published in 2008:

Blog.objects.exclude(

entry\_\_headline\_\_contains='Lennon',

entry\_\_pub\_date\_\_year=2008,

)

However, unlike the behavior when using [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter), this will not limit blogs based on entries that satisfy both conditions. In order to do that, i.e. to select all blogs that do not contain entries published with *“Lennon”* that were published in 2008, you need to make two queries:

Blog.objects.exclude(

entry\_\_in=Entry.objects.filter(

headline\_\_contains='Lennon',

pub\_date\_\_year=2008,

),

)

**Filters can reference fields on the model**

We have constructed filters that compare the value of a model field with a constant. But what if you want to compare the value of a model field with another field on the same model? Django provides [F expressions](https://docs.djangoproject.com/en/2.1/ref/models/expressions/#django.db.models.F) to allow such comparisons. Instances of F() act as a reference to a model field within a query. These references can then be used in query filters to compare the values of two different fields on the same model instance. For example, to find a list of all blog entries that have had more comments than pingbacks, we construct an F() object to reference the pingback count, and use that F() object in the query:

**from** **django.db.models** **import** F

Entry.objects.filter(n\_comments\_\_gt=F('n\_pingbacks'))

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

Entry.objects.filter(n\_comments\_\_gt=F('n\_pingbacks') \* 2)

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

Entry.objects.filter(rating\_\_lt=F('n\_comments') + F('n\_pingbacks'))

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

Entry.objects.filter(authors\_\_name=F('blog\_\_name'))

For date and date/time fields, you can add or subtract a [timedelta](https://docs.python.org/3/library/datetime.html#datetime.timedelta) object. The following would return all entries that were modified more than 3 days after they were published:

**from** **datetime** **import** timedelta

Entry.objects.filter(mod\_date\_\_gt=F('pub\_date') + timedelta(days=3))

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

F('somefield').bitand(16)

**The pk lookup shortcut**

For convenience, Django provides a pk lookup shortcut, which stands for “primary key”. In the example Blog model, the primary key is the id field, so these three statements are equivalent:

Blog.objects.get(id\_\_exact=14) *# Explicit form*

Blog.objects.get(id=14) *# \_\_exact is implied*

Blog.objects.get(pk=14) *# pk implies id\_\_exact*

The use of pk isn’t limited to \_\_exact queries – any query term can be combined with pk to perform a query on the primary key of a model:

*# Get blogs entries with id 1, 4 and 7*

Blog.objects.filter(pk\_\_in=[1,4,7])

*# Get all blog entries with id > 14*

Blog.objects.filter(pk\_\_gt=14)

pk lookups also work across joins. These three statements are equivalent:

Entry.objects.filter(blog\_\_id\_\_exact=3) *# Explicit form*

Entry.objects.filter(blog\_\_id=3) *# \_\_exact is implied*

Entry.objects.filter(blog\_\_pk=3) *# \_\_pk implies \_\_id\_\_exact*

**Escaping percent signs and underscores in LIKE statements**

The field lookups that equate to LIKE SQL statements (iexact, contains, icontains, startswith, istartswith, endswith and iendswith) will automatically escape the two special characters used in LIKE statements – the percent sign and the underscore. (In a LIKE statement, the percent sign signifies a multiple-character wildcard and the underscore signifies a single-character wildcard.) This means things should work intuitively, so the abstraction doesn’t leak. For example, to retrieve all the entries that contain a percent sign, just use the percent sign as any other character:

Entry.objects.filter(headline\_\_contains='%')

Django takes care of the quoting for you; the resulting SQL:

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

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

**Caching and QuerySets**

Each [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) contains a cache to minimize database access. Understanding how it works will allow you to write the most efficient code. In a newly created [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet), the cache is empty. The first time a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) is evaluated – and, hence, a database query happens – Django saves the query results in the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet)’s cache and returns the results that have been explicitly requested (e.g., the next element, if the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) is being iterated over). Subsequent evaluations of the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) reuse the cached results. Keep this caching behavior in mind, because it may bite you if you don’t use your [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet)s correctly. For example, the following will create two [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet)s, evaluate them, and throw them away:

**print**([e.headline **for** e **in** Entry.objects.all()])

**print**([e.pub\_date **for** e **in** Entry.objects.all()])

That means the same database query will be executed twice, effectively doubling your database load. Also, there’s a possibility the two lists may not include the same database records, because an Entry may have been added or deleted in the split second between the two requests. To avoid, save [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) and reuse it:

queryset = Entry.objects.all()

**print**([p.headline **for** p **in** queryset]) *# Evaluate the query set.*

**print**([p.pub\_date **for** p **in** queryset]) *# Re-use cache f/ evaluation.*

**When QuerySets are not cached**

Querysets do not always cache their results. When evaluating only *part* of the queryset, the cache is checked, but if it is not populated then the items returned by the subsequent query are not cached. Specifically, this means that [limiting the queryset](https://docs.djangoproject.com/en/2.1/topics/db/queries/#limiting-querysets) using an array slice or an index will not populate the cache. Repeatedly getting a certain index in a queryset object will query the database each time:

queryset = Entry.objects.all()

**print**(queryset[5]) *# Queries the database*

**print**(queryset[5]) *# Queries the database again*

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

queryset = Entry.objects.all()

[entry **for** entry **in** queryset] *# Queries the database*

**print**(queryset[5]) *# Uses cache*

**print**(queryset[5]) *# Uses cache*

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

[entry **for** entry **in** queryset]

bool(queryset)

entry **in** queryset

list(queryset)

Note: Simply printing the queryset will not populate the cache. This is because the call to \_\_repr\_\_() only returns a slice of the entire queryset.

**Complex lookups with Q objects**

Keyword argument queries – in [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter), etc. – are “AND”ed together. If you need to execute more complex queries (for example, queries with OR statements), you can use [Q objects](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.Q). A [Q object](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.Q) (django.db.models.Q) is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in “Field lookups” above. For example, this Q object encapsulates a single LIKE query:

**from** **django.db.models** **import** Q

Q(question\_\_startswith='What')

Q objects can be combined using the & and | operators. When an operator is used on two Q objects, it yields a new Q object. This statement yields a single Q object that represents the “OR” of two "question\_\_startswith" queries:

Q(question\_\_startswith='Who') | Q(question\_\_startswith='What')

This is equivalent to the following SQL WHERE clause:

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

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

Q(question\_\_startswith='Who') | ~Q(pub\_date\_\_year=2005)

Each lookup function that takes keyword-arguments (e.g. [filter()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.filter), [exclude()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exclude), [get()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.get)) can also be passed one or more Q objects as positional (not-named) arguments. If you provide multiple Q object arguments to a lookup function, the arguments will be “AND”ed together. For example:

Poll.objects.get(

Q(question\_\_startswith='Who'),

Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005, 5, 6))

)

… roughly translates into the SQL:

SELECT \* from polls WHERE question LIKE 'Who%'

AND (pub\_date = '2005-05-02' OR pub\_date = '2005-05-06')

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

Poll.objects.get(

Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005,5,6)),

question\_\_startswith='Who',

)

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

# INVALID QUERY

Poll.objects.get(

question\_\_startswith='Who',

Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005, 5, 6))

)

… would not be valid.

See also: The [OR lookups examples](https://github.com/django/django/blob/master/tests/or_lookups/tests.py) in the Django unit tests show some possible uses of Q.

**Comparing objects**

To compare two model instances, just use the standard Python comparison operator, the double equals sign: ==. Behind the scenes, that compares the primary key values of two models. Using the Entry example above, the following two statements are equivalent:

some\_entry == other\_entry

some\_entry.id == other\_entry.id

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

some\_obj == other\_obj

some\_obj.name == other\_obj.name

**Deleting objects**

The delete method, conveniently, is named [delete()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.delete). This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type. Example:

e.delete()

(1, {'weblog.Entry': 1})

You can also delete objects in bulk. Every [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) has a [delete()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.delete) method, which deletes all members of that [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet). For example, this deletes all Entry objects with a pub\_date year of 2005:

Entry.objects.filter(pub\_date\_\_year=2005).delete()

(5, {'webapp.Entry': 5})

Keep in mind that this will, whenever possible, be executed purely in SQL, and so the delete() methods of individual object instances will not necessarily be called during the process. If you’ve provided a custom delete() method on a model class and want to ensure that it is called, you will need to “manually” delete instances of that model (e.g., by iterating over a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) and calling delete() on each object individually) rather than using the bulk [delete()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.delete) method of a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet). When Django deletes an object, by default it emulates the behavior of the SQL constraint ON DELETE CASCADE – in other words, any objects which had foreign keys pointing at the object to be deleted will be deleted along with it. For example:

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

# This will delete the Blog and all of its Entry objects.

b.delete()

This cascade behavior is customizable via the [on\_delete](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey.on_delete) argument to the [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey). Note that [delete()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.delete) is the only [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) method that is not exposed on a [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) itself. This is a safety mechanism to prevent you from accidentally requesting Entry.objects.delete(), and deleting *all* the entries. If you *do* want to delete all the objects, then you have to explicitly request a complete query set:

Entry.objects.all().delete()

**Copying model instances**

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

blog = Blog(name='My blog', tagline='Blogging is easy')

blog.save() # blog.pk == 1

blog.pk = None

blog.save() # blog.pk == 2

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

class ThemeBlog(Blog):

theme = models.CharField(max\_length=200)

django\_blog = ThemeBlog(name='Django', tagline='Django is easy', theme='python')

django\_blog.save() # django\_blog.pk == 3

Due to how inheritance works, you have to set both pk and id to None:

django\_blog.pk = None

django\_blog.id = None

django\_blog.save() # django\_blog.pk == 4

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

entry = Entry.objects.all()[0] # some previous entry

old\_authors = entry.authors.all()

entry.pk = None

entry.save()

entry.authors.set(old\_authors)

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

detail = EntryDetail.objects.all()[0]

detail.pk = None

detail.entry = entry

detail.save()

**Updating multiple objects at once**

Sometimes you want to set a field to a particular value for all the objects in a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet). You can do this with the [update()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.update) method. For example:

# Update all the headlines with pub\_date in 2007.

Entry.objects.filter(pub\_date\_\_year=2007).update(headline='Everything is the same')

You can only set non-relation fields and [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) fields using this method. To update a non-relation field, provide the new value as a constant. To update [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) fields, set the new value to be the new model instance you want to point to. For example:

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

# Change every Entry so that it belongs to this Blog.

Entry.objects.all().update(blog=b)

The update() method is applied instantly and returns the number of rows matched by the query (which may not be equal to the number of rows updated if some rows already have the new value). The only restriction on the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) being updated is that it can only access one database table: the model’s main table. You can filter based on related fields, but you can only update columns in the model’s main table. Example:

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

# Update all the headlines belonging to this Blog.

Entry.objects.select\_related().filter(blog=b).update(headline='Everything is the same')

Be aware that the update() method is converted directly to an SQL statement. It is a bulk operation for direct updates. It doesn’t run any [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save) methods on your models, or emit the pre\_save or post\_save signals (which are a consequence of calling [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save)), or honor the [auto\_now](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.DateField.auto_now) field option. If you want to save every item in a [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) and make sure that the [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save) method is called on each instance, you don’t need any special function to handle that. Just loop over them and call [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save):

for item in my\_queryset:

item.save()

Calls to update can also use [F expressions](https://docs.djangoproject.com/en/2.1/ref/models/expressions/#django.db.models.F) to update one field based on the value of another field in the model. This is especially useful for incrementing counters based upon their current value. For example, to increment the pingback count for every entry in the blog:

Entry.objects.all().update(n\_pingbacks=F('n\_pingbacks') + 1)

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

# This will raise a FieldError

Entry.objects.update(headline=F('blog\_\_name'))

**Related objects**

When you define a relationship in a model (i.e., a [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey), [OneToOneField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.OneToOneField), or [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField)), instances of that model will have a convenient API to access the related object(s).

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

(Behind the scenes, this functionality is implemented by Python [descriptors](https://docs.python.org/howto/descriptor.html). This shouldn’t really matter to you, but we point it out here for the curious.)

Django also creates API accessors for the “other” side of the relationship – the link from the related model to the model that defines the relationship. For example, a Blog object b has access to a list of all related Entry objects via the entry\_set attribute: b.entry\_set.all(). All examples in this section use the sample Blog, Author and Entry models defined at the top of this page.

**One-to-many relationships**

**Forward**

If a model has a [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey), instances of that model will have access to the related (foreign) object via a simple attribute of the model. Example:

e = Entry.objects.get(id=2)

e.blog # Returns the related Blog object.

You can get and set via a foreign-key attribute. As you may expect, changes to the foreign key aren’t saved to the database until you call [save()](https://docs.djangoproject.com/en/2.1/ref/models/instances/#django.db.models.Model.save). Example:

e = Entry.objects.get(id=2)

e.blog = some\_blog

e.save()

If a [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) field has null=True set (i.e., it allows NULL values), you can assign None to remove the relation. Example:

e = Entry.objects.get(id=2)

e.blog = None

e.save() # "UPDATE blog\_entry SET blog\_id = NULL ...;"

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

e = Entry.objects.get(id=2)

print(e.blog) # Hits the database to retrieve the associated Blog.

print(e.blog) # Doesn't hit the database; uses cached version.

Note that the [select\_related()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.select_related) [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) method recursively prepopulates the cache of all one-to-many relationships ahead of time. Example:

e = Entry.objects.select\_related().get(id=2)

print(e.blog) # Doesn't hit the database; uses cached version.

print(e.blog) # Doesn't hit the database; uses cached version.

**Following relationships “backward”**

If a model has a [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey), instances of the foreign-key model will have access to a [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) that returns all instances of the first model. By default, this [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) is named FOO\_set, where FOO is the source model name, lowercased. This [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) returns QuerySets, which can be filtered and manipulated as described in the “Retrieving objects” section above. Example:

b = Blog.objects.get(id=1)

b.entry\_set.all() # Returns all Entry objects related to Blog.

# b.entry\_set is a Manager that returns QuerySets.

b.entry\_set.filter(headline\_\_contains='Lennon')

b.entry\_set.count()

You can override the FOO\_set name by setting the [related\_name](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey.related_name) parameter in the [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) definition. For example, if the Entry model was altered to blog = ForeignKey(Blog, on\_delete=models.CASCADE, related\_name='entries'), the above example code would look like this:

b = Blog.objects.get(id=1)

b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.

b.entries.filter(headline\_\_contains='Lennon')

b.entries.count()

**Using a custom reverse manager**

By default the [RelatedManager](https://docs.djangoproject.com/en/2.1/ref/models/relations/#django.db.models.fields.related.RelatedManager) used for reverse relations is a subclass of the [default manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#manager-names) for that model. If you would like to specify a different manager for a given query you can use the following syntax:

from django.db import models

class Entry(models.Model):

#...

objects = models.Manager() # Default Manager

entries = EntryManager() # Custom Manager

b = Blog.objects.get(id=1)

b.entry\_set(manager='entries').all()

If EntryManager performed default filtering in its get\_queryset() method, that filtering would apply to the all() call. Of course, specifying a custom reverse manager also enables you to call its custom methods:

b.entry\_set(manager='entries').is\_published()

**Additional methods to handle related objects**

In addition to the [QuerySet](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) methods defined in “Retrieving objects” above, the [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey) [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) has additional methods used to handle the set of related objects. A synopsis of each is below, and complete details can be found in the [related objects reference](https://docs.djangoproject.com/en/2.1/ref/models/relations/).

add(obj1, obj2, ...)

Adds the specified model objects to the related object set.

create(\*\*kwargs)

Creates a new object, saves it and puts it in the related object set. Returns the newly created object.

remove(obj1, obj2, ...)

Removes the specified model objects from the related object set.

clear()

Removes all objects from the related object set.

set(objs)

Replace the set of related objects.

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

b = Blog.objects.get(id=1)

b.entry\_set.set([e1, e2])

If the clear() method is available, any pre-existing objects will be removed from the entry\_set before all objects in the iterable (in this case, a list) are added to the set. If the clear() method is *not* available, all objects in the iterable will be added without removing any existing elements.

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

**Many-to-many relationships**

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

One difference is in the attribute naming: The model that defines the [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) uses the attribute name of that field itself, whereas the “reverse” model uses the lowercased model name of the original model, plus '\_set' (just like reverse one-to-many relationships). Example:

e = Entry.objects.get(id=3)

e.authors.all() # Returns all Author objects for this Entry.

e.authors.count()

e.authors.filter(name\_\_contains='John')

a = Author.objects.get(id=5)

a.entry\_set.all() # Returns all Entry objects for this Author.

Like [ForeignKey](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey), [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) can specify [related\_name](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField.related_name). In the above example, if the [ManyToManyField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ManyToManyField) in Entry had specified related\_name='entries', then each Author instance would have an entries attribute instead of entry\_set.

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

a = Author.objects.get(id=5)

a.entry\_set.set([e1, e2])

a.entry\_set.set([e1.pk, e2.pk])

**One-to-one relationships**

One-to-one relationships are very similar to many-to-one relationships. If you define a [OneToOneField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.OneToOneField) on your model, instances of that model will have access to the related object via a simple attribute of the model.

For example:

class EntryDetail(models.Model):

entry = models.OneToOneField(Entry, on\_delete=models.CASCADE)

details = models.TextField()

ed = EntryDetail.objects.get(id=2)

ed.entry # Returns the related Entry object.

The difference comes in “reverse” queries. The related model in a one-to-one relationship also has access to a [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) object, but that [Manager](https://docs.djangoproject.com/en/2.1/topics/db/managers/#django.db.models.Manager) represents a single object, rather than a collection of objects:

e = Entry.objects.get(id=2)

e.entrydetail # returns the related EntryDetail object

If no object has been assigned to this relationship, Django will raise a DoesNotExist exception. Instances can be assigned to the reverse relationship in the same way as you would assign the forward relationship:

e.entrydetail = ed

**How are the backward relationships possible?**

Other object-relational mappers require you to define relationships on both sides. The Django developers believe this is a violation of the DRY (Don’t Repeat Yourself) principle, so Django only requires you to define the relationship on one end. But how is this possible, given that a model class doesn’t know which other model classes are related to it until those other model classes are loaded? The answer lies in the [app registry](https://docs.djangoproject.com/en/2.1/ref/applications/#django.apps.apps). When Django starts, it imports each application listed in [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-INSTALLED_APPS), and then the models module inside each application. Whenever a new model class is created, Django adds backward-relationships to any related models. If the related models haven’t been imported yet, Django keeps tracks of the relationships and adds them when the related models eventually are imported. For this reason, it’s particularly important that all the models you’re using be defined in applications listed in [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-INSTALLED_APPS). Otherwise, backwards relations may not work properly.

**Queries over related objects**

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

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

Entry.objects.filter(blog=b) # Query using object instance

Entry.objects.filter(blog=b.id) # Query using id from instance

Entry.objects.filter(blog=5) # Query using id directly

**Falling back to raw SQL**

If you find yourself needing to write an SQL query that is too complex for Django’s database-mapper to handle, you can fall back on writing SQL by hand. Django has a couple of options for writing raw SQL queries; see [Performing raw SQL queries](https://docs.djangoproject.com/en/2.1/topics/db/sql/). Finally, it’s important to note that the Django database layer is merely an interface to your database. You can access your database via other tools, programming languages or database frameworks; there’s nothing Django-specific about your database.

Django QuerySet API

This document describes the details of the QuerySet API. It builds on the material presented in the [model](https://docs.djangoproject.com/en/2.1/topics/db/models/) and [database query](https://docs.djangoproject.com/en/2.1/topics/db/queries/) guides, so you’ll probably want to read and understand those documents before reading this one. Throughout this reference we’ll use the [example Weblog models](https://docs.djangoproject.com/en/2.1/topics/db/queries/#queryset-model-example) presented in the [database query guide](https://docs.djangoproject.com/en/2.1/topics/db/queries/).

# When QuerySets are evaluated

Internally, a QuerySet can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset. You can evaluate a QuerySet in the following ways:

* **Iteration.** A QuerySet is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database:

for e in Entry.objects.all():

print(e.headline)

Note: Don’t use this if all you want to do is determine if at least one result exists. It’s more efficient to use [exists()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exists).

* **Slicing.** As explained in [Limiting QuerySets](https://docs.djangoproject.com/en/2.1/topics/db/queries/#limiting-querysets), a QuerySet can be sliced, using Python’s array-slicing syntax. Slicing an unevaluated QuerySet usually returns another unevaluated QuerySet, but Django will execute the database query if you use the “step” parameter of slice syntax, and will return a list. Slicing a QuerySet that has been evaluated also returns a list. Also note that even though slicing an unevaluated QuerySet returns another unevaluated QuerySet, modifying it further (e.g., adding more filters, or modifying ordering) is not allowed, since that does not translate well into SQL and it would not have a clear meaning either.
* **Pickling/Caching.** See the following section for details of what is involved when [pickling QuerySets](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#pickling-querysets). The important thing for the purposes of this section is that the results are read from the database.
* **repr().** A QuerySet is evaluated when you call repr() on it. This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API interactively.
* **len().** A QuerySet is evaluated when you call len() on it. This, as you might expect, returns the length of the result list. Note: If you only need to determine the number of records in the set (and don’t need the actual objects), it’s much more efficient to handle a count at the database level using SQL’s SELECT COUNT(\*). Django provides a [count()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.count) method for this reason.
* **list().** Force evaluation of a QuerySet by calling list() on it. For example:

entry\_list = list(Entry.objects.all())

* **bool().** Testing a QuerySet in a boolean context, such as using bool(), or, and or an if statement, will cause the query to be executed. If there is at least one result, the QuerySet is True, otherwise False. For example:

if Entry.objects.filter(headline="Test"):

print("There is at least one Entry with the headline Test")

Note: If you only want to determine if at least one result exists (and don’t need the actual objects), it’s more efficient to use [exists()](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.exists).

**bulk\_create(objs, batch\_size=None)**

This method inserts the provided list of objects into the database in an efficient manner (generally only 1 query, no matter how many objects there are):

Entry.objects.bulk\_create([

... Entry(headline='This is a test'),

... Entry(headline='This is only a test'),

... ])

This has a number of caveats though:

* The model’s save() method will not be called, and the pre\_save and post\_save signals will not be sent.
* It does not work with child models in a multi-table inheritance scenario.
* If the model’s primary key is an [AutoField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.AutoField) it does not retrieve and set the primary key attribute, as save() does, unless the database backend supports it (currently PostgreSQL).
* It does not work with many-to-many relationships.
* It casts objs to a list, which fully evaluates objs if it’s a generator. The cast allows inspecting all objects so that any objects with a manually set primary key can be inserted first. If you want to insert objects in batches without evaluating the entire generator at once, you can use this technique as long as the objects don’t have any manually set primary keys:

**from** **itertools** **import** islice

batch\_size = 100

objs = (Entry(headline='Test **%s**' % i) **for** i **in** range(1000))

**while** True:

batch = list(islice(objs, batch\_size))

**if** **not** batch:

**break**

Entry.objects.bulk\_create(batch, batch\_size)

The batch\_size parameter controls how many objects are created in a single query. The default is to create all objects in one batch, except for SQLite where the default is such that at most 999 variables per query are used.

#### Example bulk\_create, pandas

**def** handle(self, \*args, \*\*options):

dtypes = {'review\_id': object, 'user\_id': object, 'business\_id': object, 'stars': float, 'date': object, 'text': object, 'useful': float, 'funny': float, 'cool': float}

ddf = dd.read\_csv('data/\*.csv', dtype=dtypes, parse\_dates=['date'])

ddf.dropna(subset=['date'])

DimDate.objects.bulk\_create([x[1][4] **for** x **in** ddf.iterrows()])

# Pickling QuerySets

If you [pickle](https://docs.python.org/3/library/pickle.html#module-pickle) a QuerySet, this will force all the results to be loaded into memory prior to pickling. Pickling is usually used as a precursor to caching and when the cached queryset is reloaded, you want the results to already be present and ready for use (reading from the database can take some time, defeating the purpose of caching). This means that when you unpickle a QuerySet, it contains the results at the moment it was pickled, rather than the results that are currently in the database. If you only want to pickle the necessary information to recreate the QuerySet from the database at a later time, pickle the query attribute of the QuerySet. You can then recreate the original QuerySet (without any results loaded) using some code like this:

import pickle

query = pickle.loads(s)# Assume 's' = pickled string.

qs = MyModel.objects.all()

qs.query = query # Restore the original 'query'.

The query attribute is an opaque object. It represents the internals of the query construction and is not part of the public API. However, it is safe (and fully supported) to pickle and unpickle the attribute’s contents as described here.

PANDAS-DJANGO MODEL

# Django ORM to insert data

**from** **myapp.models** **import** Agency

**class** Command(BaseCommand):

**def** handle(self, \*args, \*\*options):

*# Process data with Pandas*

agencies = pd.DataFrame({"name": ["Agency 1", "Agency 2", "Agency 3"]})

*# iterate over DataFrame and create your objects*

**for** agency **in** agencies.itertuples():

agency = Agency.objects.create(name=agency.name)

# import data With external script

**import** **os**, **sys**

**import** **django**

**import** **pandas** **as** **pd**

sys.path.append('../..') *# add path to project root dir*

os.environ["DJANGO\_SETTINGS\_MODULE"] = "myproject.settings"

*# for more sophisticated setups, if you need to change*

*# connection settings (e.g. when using django-environ):*

*# os.environ["DATABASE\_URL"] =*

*# "postgres://myuser:mypassword@localhost:54324/mydb"*

*# Connect to Django ORM*

django.setup()

*# process data*

**from** **myapp.models** **import** Agency

Agency.objects.create(name='MyAgency')

# Save Pandas DF as Django Model

**U**se bulk\_upload coupled with a comprehension to speed up loading. Initially, I started to convert the data frame to a Model object row by row and save it. While easy, it’s also arguably the slowest method to load records, because the save() call performs a commit.

**Manual commit**

**from** **django.db** **import** transaction

@transaction.commit\_manually

**def** save(df):

...

**for** item **in** df.to\_dict('records'):

entry = Entry(\*\*item)

entry.save()

transaction.commit()

**SQL**

Other solution would be to use raw SQL. Here, you need to replicate the table fields by tweaking the data frame.

**from** **django.conf** **import** settings

user = settings.DATABASES['default']['USER']

password = settings.DATABASES['default']['PASSWORD']

database\_name = settings.DATABASES['default']['NAME']

database\_url = 'postgresql://{user}:{password}@localhost:5432/{database\_name}'.format(

user=user,

password=password,

database\_name=database\_name,

)

engine = create\_engine(database\_url, echo=**False**)

df.to\_sql(model.\_meta.db\_table, con=engine)

Where:

* model.\_meta.db\_table is the table where we want to save
* database\_url is a sql\_alchemy database URL (above configured for PostgreSQL)

This option has a few drawbacks:

1. The URL needs to be manually created. If you have different DB engines for dev/test, you’ll need to cater for that
2. You’ll need to add a pk column with incremental values
3. The code needs to be changed for each migration. Ouch!

**Using bulk\_create()**

Recent versions of django added bulk\_create() where you can pass an array of model objects and they get created at once. The code would be something like:

Model.objects.bulk\_create(

Model(\*\*vals) **for** vals **in** df.to\_dict('records'))

The operation works like this:

1. Convert the dataframe in a list of dicts
2. Create a Model object for each item in the list
3. Perform the bulk\_create() on the list

This approach has some disadvantages too:

1. It creates objects (not updates) so you can end up with duplicates if not careful
2. Make sure the dict generated by a row can be used to create the model object
3. It’s slower than the SQL method above
4. You might end up with lots of memory allocated if the dataframe is very large

The above disadvantages are minor compared to the simplicity of the execution.

**Adapting DataFrame to Match**

My approach to making sure the dict generated from a dataframe’s row can be used to create a Model object is:

* Note the model’s fields which are not nullable
* Change the existing column names to match the field names
* Pay attention and convert timestamp values to datetime values:

df['timestamp'] = pd.to\_datetime(df['timestamp'], unit='s')

* Create any missing columns

# Insert 1,000 records into SQLite table using Django

check out django.db.transaction.commit\_manually.

<http://docs.djangoproject.com/en/dev/topics/db/transactions/#django-db-transaction-commit-manually>, So it would be something like:

**from** **django.db** **import** transaction

@transaction.commit\_manually

**def** viewfunc(request):

...

**for** item **in** items:

entry = Entry(a1=item.a1, a2=item.a2)

entry.save()

transaction.commit()

Which will only commit once, instead at each save(). In django 1.3 context managers were introduced: [**transaction.commit\_on\_success()**](https://docs.djangoproject.com/en/1.3/topics/db/transactions/#controlling-transaction-management-in-views) in a similar way:

**from** **django.db** **import** transaction

**def** viewfunc(request):

...

**with** transaction.commit\_on\_success():

**for** item **in** items:

entry = Entry(a1=item.a1, a2=item.a2)

entry.save()

In django 1.4, [bulk\_create](https://docs.djangoproject.com/en/dev/ref/models/querysets/#django.db.models.query.QuerySet.bulk_create) was added, allowing you to create lists of your model objects and then commit them all at once.

**NOTE** the save method will not be called when using bulk create.

Entry.objects.bulk\_create([

... Entry(headline="Django 1.0 Released"),

... Entry(headline="Django 1.1 Announced"),

... Entry(headline="Breaking: Django is awesome")

... ])

In django 1.6, [**transaction.atomic**](https://docs.djangoproject.com/en/1.6/topics/db/transactions/#django.db.transaction.atomic) was introduced, intended to replace now legacy functions commit\_on\_success and commit\_manually. From django [documentation on atomic](https://docs.djangoproject.com/en/1.6/topics/db/transactions/#django.db.transaction.atomic), atomic is usable both as a decorator:

**from** **django.db** **import** transaction

@transaction.atomic

**def** viewfunc(request):

*# This code executes inside a transaction.*

do\_stuff()

… and as a context manager:

**from** **django.db** **import** transaction

**def** viewfunc(request):

*# This code executes in autocommit mode (Django's default).*

do\_stuff()

**with** transaction.atomic():

*# This code executes inside a transaction.*

do\_more\_stuff()

DJANGO-PANDAS

# Io.py

**import** **pandas** **as** **pd**

**from** **.utils** **import** update\_with\_verbose, get\_related\_model

**import** **django**

## **def** to\_fields(qs, fieldnames):

**for** fieldname **in** fieldnames:

model = qs.model

**for** fieldname\_part **in** fieldname.split('\_\_'):

**try**:

field = model.\_meta.get\_field(fieldname\_part)

**except** django.db.models.fields.FieldDoesNotExist:

**try**:

rels = model.\_meta.get\_all\_related\_objects\_with\_model()

**except** **AttributeError**:

field = fieldname

**else**:

**for** relobj, \_ **in** rels:

**if** relobj.get\_accessor\_name() == fieldname\_part:

field = relobj.field

model = field.model

**break**

**else**:

model = get\_related\_model(field)

**yield** field

## def is\_values\_queryset(qs):

**if** django.VERSION < (1, 9): *# pragma: no cover*

**return** isinstance(qs, django.db.models.query.ValuesQuerySet)

**else**:

**return** qs.\_iterable\_class == django.db.models.query.ValuesIterable

## **def** read\_frame(qs, fieldnames=(), index\_col=None, coerce\_float=False, verbose=True, datetime\_index=False):

*"""*

*Returns a dataframe from a QuerySet*

*Optionally specify the field names/columns to utilize and*

*a field as the index*

*Parameters*

*----------*

*qs: The Django QuerySet.*

*fieldnames: The model field names to use in creating the frame.*

*You can span a relationship in the usual Django way*

*by using double underscores to specify a related field*

*in another model*

*You can span a relationship in the usual Django way*

*by using double underscores to specify a related field*

*in another model*

*index\_col: specify the field to use for the index. If the index*

*field is not in the field list it will be appended*

*coerce\_float : boolean, default False*

*Attempt to convert values to non-string, non-numeric data (like*

*decimal.Decimal) to floating point, useful for SQL result sets*

*verbose: boolean If this is ``True`` then populate the DataFrame with the*

*human readable versions of any foreign key fields else use*

*the primary keys values.*

*The human readable version of the foreign key field is*

*defined in the ``\_\_unicode\_\_`` or ``\_\_str\_\_``*

*methods of the related class definition*

*datetime\_index: specify whether index should be converted to a*

*DateTimeIndex.*

*"""*

**if** fieldnames:

fieldnames = pd.unique(fieldnames)

**if** index\_col **is** **not** None **and** index\_col **not** **in** fieldnames:

*# Add it to the field names if not already there*

fieldnames = tuple(fieldnames) + (index\_col,)

fields = to\_fields(qs, fieldnames)

**elif** is\_values\_queryset(qs):

**if** django.VERSION < (1, 9): *# pragma: no cover*

annotation\_field\_names = list(qs.query.annotation\_select)

**if** annotation\_field\_names **is** None:

annotation\_field\_names = []

extra\_field\_names = qs.extra\_names

**if** extra\_field\_names **is** None:

extra\_field\_names = []

select\_field\_names = qs.field\_names

**else**: *# pragma: no cover*

annotation\_field\_names = list(qs.query.annotation\_select)

extra\_field\_names = list(qs.query.extra\_select)

select\_field\_names = list(qs.query.values\_select)

fieldnames = select\_field\_names + annotation\_field\_names + \

extra\_field\_names

fields = [None **if** '\_\_' **in** f **else** qs.model.\_meta.get\_field(f)

**for** f **in** select\_field\_names] + \

[None] \* (len(annotation\_field\_names) + len(extra\_field\_names))

uniq\_fields = set()

fieldnames, fields = zip(

\*(f **for** f **in** zip(fieldnames, fields)

**if** f[0] **not** **in** uniq\_fields **and** **not** uniq\_fields.add(f[0])))

**else**:

fields = qs.model.\_meta.fields

fieldnames = [f.name **for** f **in** fields]

fieldnames += list(qs.query.annotation\_select.keys())

**if** is\_values\_queryset(qs):

recs = list(qs)

**else**:

recs = list(qs.values\_list(\*fieldnames))

df = pd.DataFrame.from\_records(recs, columns=fieldnames,

coerce\_float=coerce\_float)

**if** verbose:

update\_with\_verbose(df, fieldnames, fields)

**if** index\_col **is** **not** None:

df.set\_index(index\_col, inplace=True)

**if** datetime\_index:

df.index = pd.to\_datetime(df.index, errors="ignore")

**return** df

# Managers.py

**from** **django.db.models.query** **import** QuerySet

**from** **.io** **import** read\_frame

**import** **django**

**from** **django.db** **import** models

## **class** PassThroughManagerMixin(object):

*"""*

*A mixin that enables you to call custom QuerySet methods from your manager.*

*"""*

\_deny\_methods = ['\_\_getstate\_\_', '\_\_setstate\_\_', '\_\_getinitargs\_\_',

'\_\_getnewargs\_\_', '\_\_copy\_\_', '\_\_deepcopy\_\_', '\_db',

'\_\_slots\_\_']

### **def** \_\_init\_\_(self, queryset\_cls=None):

self.\_queryset\_cls = queryset\_cls

super(PassThroughManagerMixin, self).\_\_init\_\_()

### **def** \_\_getattr\_\_(self, name):

**if** name **in** self.\_deny\_methods:

**raise** **AttributeError**(name)

**return** getattr(self.get\_queryset(), name)

### **def** \_\_dir\_\_(self):

my\_values = frozenset(dir(type(self)))

my\_values |= frozenset(dir(self.get\_query\_set()))

**return** list(my\_values)

### **def** get\_queryset(self):

**try**:

qs = super(PassThroughManagerMixin, self).get\_queryset()

**except** **AttributeError**:

qs = super(PassThroughManagerMixin, self).get\_query\_set()

**if** self.\_queryset\_cls **is** **not** None:

qs = qs.\_clone(klass=self.\_queryset\_cls)

**return** qs

get\_query\_set = get\_queryset

@classmethod

### **def** for\_queryset\_class(cls, queryset\_cls):

**return** create\_pass\_through\_manager\_for\_queryset\_class(

cls, queryset\_cls)

## **class** PassThroughManager(PassThroughManagerMixin, models.Manager):

*"""*

*Inherit from this Manager to enable you to call any methods from your*

*custom QuerySet class from your manager. Simply define your QuerySet*

*class, and return an instance of it from your manager's `get\_queryset`*

*method.*

*Alternately, if you don't need any extra methods on your manager that*

*aren't on your QuerySet, then just pass your QuerySet class to the*

*``for\_queryset\_class`` class method.*

*class PostQuerySet(QuerySet):*

*def enabled(self):*

*return self.filter(disabled=False)*

*class Post(models.Model):*

*objects = PassThroughManager.for\_queryset\_class(PostQuerySet)()*

*"""*

**pass**

## **def** create\_pass\_through\_manager\_for\_queryset\_class(base, queryset\_cls):

### **class** \_PassThroughManager(base):

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

**return** super(\_PassThroughManager, self).\_\_init\_\_(\*args, \*\*kwargs)

### **def** get\_queryset(self):

qs = super(\_PassThroughManager, self).get\_queryset()

**return** qs.\_clone(klass=queryset\_cls)

get\_query\_set = get\_queryset

**return** \_PassThroughManager

## **class** DataFrameQuerySet(QuerySet):

### def to\_pivot\_table(self, fieldnames=(), verbose=True, values=None, rows=None, cols=None, aggfunc='mean', fill\_value=None, margins=False, dropna=True, coerce\_float=True):

*"""*

*A convenience method for creating a spread sheet style pivot table*

*as a DataFrame*

*Parameters*

*----------*

*fieldnames: The model field names(columns) to utilise in creating*

*the DataFrame. You can span a relationships in the usual*

*Django ORM way by using the foreign key field name*

*separated by double underscores and refer to a field*

*in a related model.*

*values: The field to use to calculate the values to aggregate.*

*rows: The list of field names to group on*

*Keys to group on the x-axis of the pivot table*

*cols: The list of column names or arrays to group on*

*Keys to group on the y-axis of the pivot table*

*aggfunc: How to arregate the values. By default this would be*

*``numpy.mean``. A list of aggregates functions can be passed*

*In this case the resulting pivot table will have*

*hierarchical columns whose top level are the function names*

*(inferred from the function objects themselves)*

*fill\_value: A scalar value to replace the missing values with*

*margins: Boolean, default False Add all row / columns*

*(e.g. for subtotal / grand totals)*

*dropna: Boolean, default True.*

*Do not include columns whose entries are all NaN*

*verbose: If this is ``True`` then populate the DataFrame with the*

*human readable versions for foreign key fields else use the*

*actual values set in the model*

*coerce\_float: Attempt to convert values to non-string, non-numeric*

*objects (like decimal.Decimal) to floating point.*

*"""*

df = self.to\_dataframe(fieldnames, verbose=verbose,

coerce\_float=coerce\_float)

**return** df.pivot\_table(values=values, fill\_value=fill\_value, index=rows,

columns=cols, aggfunc=aggfunc, margins=margins,

dropna=dropna)

### **def** to\_timeseries(self, fieldnames=(), verbose=True, index=None, storage='wide', values=None, pivot\_columns=None, freq=None, coerce\_float=True, rs\_kwargs=None):

*"""*

*A convenience method for creating a time series DataFrame i.e the*

*DataFrame index will be an instance of DateTime or PeriodIndex*

*Parameters*

*----------*

*fieldnames: The model field names(columns) to utilise in creating*

*the DataFrame. You can span a relationships in the usual*

*Django ORM way by using the foreign key field name*

*separated by double underscores and refer to a field*

*in a related model.*

*index: specify the field to use for the index. If the index*

*field is not in fieldnames it will be appended. This*

*is mandatory for timeseries.*

*storage: Specify if the queryset uses the*

*``wide`` format*

*date | col1| col2| col3|*

*-----------|------|-----|-----|*

*2001-01-01-| 100.5| 23.3| 2.2|*

*2001-02-01-| 106.3| 17.0| 4.6|*

*2001-03-01-| 111.7| 11.1| 0.7|*

*or the `long` format.*

*date |values| names|*

*-----------|------|------|*

*2001-01-01-| 100.5| col1|*

*2001-02-01-| 106.3| col1|*

*2001-03-01-| 111.7| col1|*

*2001-01-01-| 23.3| col2|*

*2001-02-01-| 17.0| col2|*

*2001-01-01-| 23.3| col2|*

*2001-02-01-| 2.2| col3|*

*2001-03-01-| 4.6| col3|*

*2001-03-01-| 0.7| col3|*

*pivot\_columns: Required once the you specify `long` format*

*storage. This could either be a list or string*

*identifying the field name or combination of field.*

*If the pivot\_column is a single column then the*

*unique values in this column become a new columns in*

*the DataFrame If the pivot column is a list the values*

*in these columns are concatenated (using the '-'*

*as a separator) and these values are used for the new*

*timeseries columns*

*values: Also required if you utilize the `long` storage the*

*values column name is use for populating new frame values*

*freq: The offset string or object representing a target conversion*

*rs\_kwargs: A dictonary of keyword arguments based on the*

*``pandas.DataFrame.resample`` method*

*verbose: If this is ``True`` then populate the DataFrame with the*

*human readable versions of any foreign key fields else use*

*the primary keys values else use the actual values set*

*in the model.*

*coerce\_float: Attempt to convert values to non-string, non-numeric*

*objects (like decimal.Decimal) to floating point.*

*"""*

**assert** index **is** **not** None, 'You must supply an index field'

**assert** storage **in** ('wide', 'long'), 'storage must be wide or long'

**if** rs\_kwargs **is** None:

rs\_kwargs = {}

**if** storage == 'wide':

df = self.to\_dataframe(fieldnames, verbose=verbose, index=index,

coerce\_float=coerce\_float, datetime\_index=True)

**else**:

df = self.to\_dataframe(fieldnames, verbose=verbose,

coerce\_float=coerce\_float, datetime\_index=True)

**assert** values **is** **not** None, 'You must specify a values field'

**assert** pivot\_columns **is** **not** None, 'You must specify pivot\_columns'

**if** isinstance(pivot\_columns, (tuple, list)):

df['combined\_keys'] = ''

**for** c **in** pivot\_columns:

df['combined\_keys'] += df[c].str.upper() + '.'

df['combined\_keys'] += values.lower()

df = df.pivot(index=index,

columns='combined\_keys',

values=values)

**else**:

df = df.pivot(index=index,

columns=pivot\_columns,

values=values)

**if** freq **is** **not** None:

df = df.resample(freq, \*\*rs\_kwargs)

**return** df

### **def** to\_dataframe(self, fieldnames=(), verbose=True, index=None,

coerce\_float=False, datetime\_index=False):

*"""*

*Returns a DataFrame from the queryset*

*Paramaters*

*-----------*

*fieldnames: The model field names(columns) to utilise in creating*

*the DataFrame. You can span a relationships in the usual*

*Django ORM way by using the foreign key field name*

*separated by double underscores and refer to a field*

*in a related model.*

*index: specify the field to use for the index. If the index*

*field is not in fieldnames it will be appended. This*

*is mandatory for timeseries.*

*verbose: If this is ``True`` then populate the DataFrame with the*

*human readable versions for foreign key fields else*

*use the actual values set in the model*

*coerce\_float: Attempt to convert values to non-string, non-numeric*

*objects (like decimal.Decimal) to floating point.*

*datetime\_index: specify whether index should be converted to a*

*DateTimeIndex.*

*"""*

**return** read\_frame(self, fieldnames=fieldnames, verbose=verbose,

index\_col=index, coerce\_float=coerce\_float,

datetime\_index=datetime\_index)

DataFrameManager = models.Manager.from\_queryset(DataFrameQuerySet)

# Utils.py

*# coding: utf-8*

**from** **django.core.cache** **import** cache

**from** **django.utils.encoding** **import** force\_text

**from** **django.db.models** **import** Field

## **def** get\_model\_name(model):

*"""*

*Returns the name of the model*

*"""*

**return** model.\_meta.model\_name

## **def** replace\_from\_choices(choices):

**def** inner(values):

**return** [choices.get(v, v) **for** v **in** values]

**return** inner

## **def** get\_base\_cache\_key(model):

**return** 'pandas\_**%s**\_**%s**\_**%%**s\_rendering' % (

model.\_meta.app\_label, get\_model\_name(model))

## **def** get\_cache\_key(obj):

**return** get\_base\_cache\_key(obj.\_meta.model) % obj.pk

## **def** invalidate(obj):

cache.delete(get\_cache\_key(obj))

## **def** invalidate\_signal\_handler(sender, \*\*kwargs):

invalidate(kwargs['instance'])

## **def** replace\_pk(model):

base\_cache\_key = get\_base\_cache\_key(model)

### **def** get\_cache\_key\_from\_pk(pk):

**return** None **if** pk **is** None **else** base\_cache\_key % pk

### **def** inner(pk\_series):

pk\_series = pk\_series.where(pk\_series.notnull(), None)

cache\_keys = pk\_series.apply(

get\_cache\_key\_from\_pk, convert\_dtype=False)

unique\_cache\_keys = list(filter(None, cache\_keys.unique()))

**if** **not** unique\_cache\_keys:

**return** pk\_series

out\_dict = cache.get\_many(unique\_cache\_keys)

**if** len(out\_dict) < len(unique\_cache\_keys):

out\_dict = dict([(base\_cache\_key % obj.pk, force\_text(obj))

**for** obj **in** model.objects.filter(

pk\_\_in=list(filter(None, pk\_series.unique())))])

cache.set\_many(out\_dict)

**return** list(map(out\_dict.get, cache\_keys))

**return** inner

## **def** build\_update\_functions(fieldnames, fields):

**for** fieldname, field **in** zip(fieldnames, fields):

**if** **not** isinstance(field, Field):

**yield** fieldname, None

**else**:

**if** field **and** field.choices:

choices = dict([(k, force\_text(v))

**for** k, v **in** field.flatchoices])

**yield** fieldname, replace\_from\_choices(choices)

**elif** field **and** field.get\_internal\_type() == 'ForeignKey':

**yield** fieldname, replace\_pk(get\_related\_model(field))

## **def** update\_with\_verbose(df, fieldnames, fields):

**for** fieldname, function **in** build\_update\_functions(fieldnames, fields):

**if** function **is** **not** None:

df[fieldname] = function(df[fieldname])

## **def** get\_related\_model(field):

*"""Gets the related model from a related field"""*

model = None

**if** hasattr(field, 'related\_model') **and** field.related\_model: *# pragma: no cover*

model = field.related\_model

*# Django<1.8 doesn't have the related\_model API, so we need to use rel,*

*# which was removed in Django 2.0*

**elif** hasattr(field, 'rel') **and** field.rel: *# pragma: no cover*

model = field.rel.to

**return** model

DJANGO DATA WIZARD

# Data\_WIZARD.\_\_init\_\_.py

**from** **django.utils.module\_loading** **import** autodiscover\_modules

**from** **.registry** **import** registry

\_\_all\_\_ = (

"autodiscover",

"backend",

"registry",

"register",

"set\_loader",

"default\_app\_config",

)

backend = None

## **def** autodiscover():

autodiscover\_modules('wizard', register\_to=None)

## **def** init\_backend():

**global** backend

**from** **.backends** **import** create\_backend

backend = create\_backend()

## **def** register(\*args, \*\*kwargs):

registry.register(\*args, \*\*kwargs)

## **def** set\_loader(\*args, \*\*kwargs):

registry.set\_loader(\*args, \*\*kwargs)

default\_app\_config = 'data\_wizard.apps.WizardConfig'

# Data\_WIZARD.Admin.py

**from** **django.contrib** **import** admin

**from** **django.contrib** **import** messages

**from** **django.contrib.contenttypes.models** **import** ContentType

**from** **.models** **import** Run, RunLog, Identifier, Range, Record

**from** **.compat** **import** reverse

**from** **django.http** **import** HttpResponseRedirect

## **class** FixedTabularInline(admin.TabularInline):

can\_delete = False

extra = 0

### **def** has\_add\_permission(self, request, obj):

**return** False

## **class** RangeInline(admin.TabularInline):

model = Range

fields = [

'identifier',

'type',

'header\_col',

'start\_col',

'end\_col',

'header\_row',

'start\_row',

'end\_row',

'count'

]

extra = 0

## **class** RecordInline(FixedTabularInline):

model = Record

fields = readonly\_fields = [

'row',

'success',

'content\_type',

'content\_object',

'fail\_reason'

]

## **class** RunLogInline(FixedTabularInline):

model = RunLog

readonly\_fields = ['event', 'date']

@admin.register(Run)

## **class** RunAdmin(admin.ModelAdmin):

list\_display = [

'\_\_str\_\_', 'serializer\_label', 'record\_count', 'last\_update'

]

inlines = [RangeInline, RecordInline, RunLogInline]

@admin.register(Identifier)

## **class** IdentifierAdmin(admin.ModelAdmin):

list\_display = [

'serializer\_label',

'type\_label',

'name',

'mapping\_label',

'resolved',

]

list\_display\_links = ['name', 'mapping\_label']

list\_filter = ['serializer']

## **def** start\_data\_wizard(modeladmin, request, queryset):

**if** queryset.count() != 1:

modeladmin.message\_user(

request,

'Select a single row to start data wizard.',

level=messages.ERROR,

)

**return**

instance = queryset.first()

**if** isinstance(instance, Run):

run = instance

**else**:

ct = ContentType.objects.get\_for\_model(queryset.model)

run = Run.objects.create(

user=request.user,

content\_type=ct,

object\_id=instance.pk,

)

**return** HttpResponseRedirect(

reverse('data\_wizard:run-serializers', kwargs={'pk': run.pk})

)

start\_data\_wizard.short\_description = "Import via data wizard"

admin.site.add\_action(start\_data\_wizard, 'data\_wizard')

# Data\_WIZARD.Apps.py

**from** **django.apps** **import** AppConfig

**from** **django.core.exceptions** **import** ImproperlyConfigured

**from** **.compat** **import** reverse

**from** **django.conf** **import** settings

## **class** WizardConfig(AppConfig):

name = 'data\_wizard'

verbose\_name = 'Data Wizard'

### **def** ready(self):

self.module.autodiscover()

self.module.init\_backend()

**if** 'wq.db.rest' **in** settings.INSTALLED\_APPS:

**return**

*# FIXME: Drop this check in 2.0*

**try**:

base\_url = reverse('data\_wizard:run-list')

**except** **Exception**:

**pass**

**else**:

**if** base\_url == '/':

**raise** ImproperlyConfigured(

"data\_wizard.urls at /, add 'datawizard/' prefix"

)

# Data\_WIZARD.compat.py

*# FIXME: Drop this support in 2.0*

**try**:

**from** **django.urls** **import** reverse, NoReverseMatch

**except** **ImportError**:

*# Django 1.8*

**from** **django.core.urlresolvers** **import** reverse, NoReverseMatch

\_\_all\_\_ = ['reverse', 'NoReverseMatch']

# Data\_WIZARD.fields.py

**from** **rest\_framework** **import** serializers

**from** **django.utils.six** **import** string\_types

**import** **datetime**

DATE\_FIELDS = {

'DateTimeField': datetime.datetime,

'DateField': datetime.date,

}

## **class** DateTimeSplitField(serializers.Field):

*# FIXME*

**pass**

## **def** process\_date\_FIXME(meta\_field, meta\_datatype, meta\_key, val, obj):

*# A meta\_field value of '[field].[part]' indicates a value is split across*

*# multiple columns. For example, a spreadsheet could contain two columns*

*# (date and time) that would be merged into a single "observed" field on a*

*# custom Event class. There would then be two MetaColumns values, with*

*# names of "observed.date" and "observed.time" respectively.*

**if** '.' **in** meta\_field:

meta\_field, part = meta\_field.split('.')

**else**:

part = None

*# Automatically parse date values as such*

**if** (meta\_datatype **in** DATE\_FIELDS **and** isinstance(val, string\_types) **and**

part != 'time'):

**from** **dateutil.parser** **import** parse

val = parse(val)

**if** meta\_datatype == 'DateField':

val = val.date()

*# If field is already set by an earlier column, this value might be the*

*# second half of a date/time pair.*

**if** obj[meta\_key].get(meta\_field, None) **is** **not** None:

**if** **not** part:

**raise** **Exception**(

'Multiple columns found for **%s**' % meta\_field

)

**if** part **not** **in** ('date', 'time'):

**raise** **Exception**(

'Unexpected multi-column field name: **%s**.**%s**!' % (

meta\_field, part

)

)

other\_val = obj[meta\_key][meta\_field]

val = process\_date\_part(val, other\_val, part)

## **def** process\_date\_part(new\_val, old\_val, part):

*"""*

*Combine separate date & time columns into a single value.*

*"""*

**if** part == 'date':

date, time = new\_val, old\_val

**else**:

date, time = old\_val, new\_val

*# Date should already be a valid date (see parse in save\_metadata\_value)*

**if** **not** isinstance(date, datetime.date):

**raise** **Exception**("Expected date but got **%s**!" % date)

*# Try some extra hacks to convert time values*

**if** **not** isinstance(time, datetime.time):

**if** (isinstance(time, float) **and**

time >= 100 **and** time <= 2400):

*# "Numeric" time (hour \* 100 + minutes)*

time = str(time)

**elif** isinstance(time, string\_types) **and** ":" **in** time:

*# Take out semicolon for isdigit() code below*

time = time.replace(":", "")

*# FIXME: what about seconds?*

**if** time.isdigit() **and** len(time) **in** (3, 4):

**if** len(time) == 3:

*# 300 -> time(3, 0)*

time = datetime.time(

int(time[0]),

int(time[1:])

)

**else**:

*# 1200 -> time(12, 0)*

time = datetime.time(

int(time[0:2]),

int(time[2:])

)

**else**:

*# Meh, it was worth a shot*

**raise** **Exception**("Expected time but got **%s**!" % time)

**return** datetime.datetime.combine(date, time)

# Data\_WIZARD.loaders.py

## **class** BaseLoader(object):

default\_serializer = None

### **def** \_\_init\_\_(self, run):

self.run = run

### **def** get\_serializer\_name(self):

**return** self.default\_serializer

@property

### **def** content\_object(self):

obj = self.run.content\_object

**if** **not** obj:

**raise** **Exception**("Could not find {} with pk={}".format(

self.run.content\_type,

self.run.object\_id,

))

**return** obj

### **def** load\_io\_options(self):

**return** {

key: val

**for** key, val **in** self.run.get\_serializer\_options().items()

**if** key **in** self.valid\_options

}

## **class** FileLoader(BaseLoader):

file\_attr = 'file'

valid\_options = {'header\_row', 'start\_row'}

@property

### **def** file(self):

**return** getattr(self.content\_object, self.file\_attr)

### **def** load\_io(self):

**from** **wq.io** **import** load\_file

options = self.load\_io\_options()

**return** load\_file(self.file.path, options=options)

## **class** URLLoader(BaseLoader):

url\_attr = 'url'

valid\_options = {'header\_row', 'start\_row'}

@property

### **def** url(self):

**return** getattr(self.content\_object, self.url\_attr)

### **def** load\_io(self):

**from** **wq.io** **import** load\_url

options = self.load\_io\_options()

**return** load\_url(self.url, options=options)

# Data\_WIZARD.models.py

**from** **django.db** **import** models

**from** **django.contrib.contenttypes.models** **import** ContentType

**from** **django.contrib.contenttypes.fields** **import** GenericForeignKey

**from** **django.conf** **import** settings

**import** **data\_wizard**

**from** **data\_wizard** **import** registry

**from** **.compat** **import** reverse

## **class** Run(models.Model):

user = models.ForeignKey(

settings.AUTH\_USER\_MODEL, on\_delete=models.PROTECT,

)

record\_count = models.IntegerField(null=True, blank=True)

loader = models.CharField(max\_length=255, null=True, blank=True)

serializer = models.CharField(max\_length=255, null=True, blank=True)

content\_type = models.ForeignKey(

ContentType, null=True, blank=True, on\_delete=models.PROTECT,

)

object\_id = models.PositiveIntegerField(null=True, blank=True)

content\_object = GenericForeignKey()

### **def** \_\_str\_\_(self):

**return** "Run for **%s**" % self.content\_object

### **def** get\_absolute\_url(self):

**return** reverse('data\_wizard:run-detail', kwargs={'pk': self.pk})

### **def** save(self, \*args, \*\*kwargs):

**if** **not** self.loader:

self.loader = registry.get\_loader\_name(

type(self.content\_object)

)

**if** self.loader **and** **not** self.serializer:

**try**:

Loader = registry.get\_loader(self.loader)

**except** **ImportError**:

**pass**

**else**:

self.serializer = Loader(self).get\_serializer\_name()

is\_new = **not** self.id

super(Run, self).save(\*args, \*\*kwargs)

**if** is\_new:

self.add\_event('created')

### **def** load\_io(self):

**if** **not** hasattr(self, '\_io\_data'):

Loader = registry.get\_loader(self.loader)

loader = Loader(self)

self.\_io\_data = loader.load\_io()

**return** self.\_io\_data

**def** run\_task(self, name, use\_async=False, post=None,

### backend=None, user=None):

**if** **not** backend:

backend = data\_wizard.backend

**if** **not** user:

user = self.user

**return** backend.run(

name, self.pk, user.pk, use\_async, post,

)

@property

### **def** serializer\_label(self):

**if** self.serializer:

**return** registry.get\_serializer\_name(self.serializer)

serializer\_label.fget.short\_description = 'serializer'

### **def** get\_serializer(self):

**if** self.serializer:

**return** registry.get\_serializer(self.serializer)

**else**:

**raise** **Exception**("No serializer specified!")

### **def** get\_serializer\_options(self):

**if** self.serializer:

**return** registry.get\_serializer\_options(self.serializer)

**else**:

**raise** **Exception**("No serializer specified!")

### **def** already\_parsed(self):

**return** self.range\_set.count()

### **def** add\_event(self, name):

self.log.create(

event=name

)

@property

### **def** last\_update(self):

last = self.log.last()

**if** last:

**return** last.date

### **class** Meta:

ordering = ('-pk',)

## **class** RunLog(models.Model):

run = models.ForeignKey(Run, related\_name='log', on\_delete=models.CASCADE)

event = models.CharField(max\_length=100)

date = models.DateTimeField(auto\_now\_add=True)

### **def** \_\_str\_\_(self):

**return** self.event

### **class** Meta:

ordering = ('date',)

## **class** Identifier(models.Model):

serializer = models.CharField(max\_length=255)

name = models.CharField(

max\_length=255,

verbose\_name='spreadsheet value',

)

value = models.CharField(

max\_length=255, null=True, blank=True,

verbose\_name='mapped value',

)

field = models.CharField(

max\_length=255, null=True, blank=True,

verbose\_name='serializer field',

)

attr\_field = models.CharField(

max\_length=255, null=True, blank=True,

verbose\_name='EAV attribute field',

)

attr\_id = models.PositiveIntegerField(

null=True, blank=True,

verbose\_name='EAV attribute id',

)

resolved = models.BooleanField(default=False)

### **def** \_\_str\_\_(self):

**if** self.mapping\_label:

**return** '{name} -> {mapping}'.format(

name=self.name,

mapping=self.mapping\_label

)

**else**:

**return** "{type}: {name}".format(

type=self.type\_label,

name=self.name

)

@property

### **def** type(self):

**if** self.resolved:

**if** self.attr\_id **is** **not** None:

**return** 'attribute'

**elif** self.value **is** **not** None:

**return** 'instance'

**elif** self.field:

**return** 'meta'

**else**:

**if** self.field:

**return** 'unresolved'

**else**:

**return** 'unknown'

@property

### **def** type\_label(self):

**if** self.type == 'attribute':

**return** 'EAV Column'

**elif** self.type == 'meta':

**return** 'Column/Header'

**elif** self.type == 'instance':

**return** 'FK Value'

**else**:

**return** self.type.title()

type\_label.fget.short\_description = 'Type'

@property

### **def** mapping\_label(self):

**if** self.type == 'meta':

**if** self.field == '\_\_ignore\_\_':

**return** '(ignored)'

**else**:

**return** self.field

**elif** self.type == 'attribute':

**if** '[]' **in** self.field:

prefix, field\_name = self.field.split('[]')

field\_name = field\_name.strip('[]')

**else**:

prefix = ""

field\_name = self.field

**if** self.attr\_field **and** '[]' **in** self.attr\_field:

attr\_prefix, attr\_field\_name = self.attr\_field.split('[]')

*# assert attr\_prefix == prefix*

attr\_field\_name = attr\_field\_name.strip('[]')

**else**:

attr\_field\_name = 'attr'

**return** "{prefix}.{field} ({attr\_field}={attr\_id})".format(

prefix=prefix,

field=field\_name,

attr\_field=attr\_field\_name,

attr\_id=self.attr\_id,

)

**elif** self.type == 'instance':

**return** "{field}={value}".format(

field=self.field,

value=self.value,

)

mapping\_label.fget.short\_description = 'Mapped To'

@property

### **def** serializer\_label(self):

**if** self.serializer:

**return** registry.get\_serializer\_name(self.serializer)

serializer\_label.fget.short\_description = 'Serializer'

## **class** Range(models.Model):

RANGE\_TYPES = (

('list', 'Data Column'),

('value', 'Header metadata'),

('data', 'Cell value'),

)

run = models.ForeignKey(Run, on\_delete=models.CASCADE)

identifier = models.ForeignKey(Identifier, on\_delete=models.PROTECT)

type = models.CharField(max\_length=10, choices=RANGE\_TYPES)

header\_col = models.IntegerField()

start\_col = models.IntegerField()

end\_col = models.IntegerField(null=True, blank=True)

header\_row = models.IntegerField()

start\_row = models.IntegerField()

end\_row = models.IntegerField(null=True, blank=True)

count = models.IntegerField(null=True, blank=True)

### **def** save(self, \*args, \*\*kwargs):

**if** self.header\_col **is** None:

self.header\_col = self.start\_col

**if** self.header\_row **is** None:

self.header\_row = self.start\_row

super(Range, self).save()

### **def** \_\_str\_\_(self):

**if** self.start\_row == self.end\_row:

row = "Row **%s**" % self.start\_row

**elif** self.end\_row **is** **not** None:

row = "Rows **%s**-**%s**" % (self.start\_row, self.end\_row)

**else**:

row = "Row **%s** onward" % (self.start\_row,)

**if** self.start\_col == self.end\_col:

col = "Column **%s**" % self.start\_col

**elif** self.end\_col **is** **not** None:

col = "Column **%s**-**%s**" % (self.start\_col, self.end\_col)

**else**:

row = "Column **%s** onward" % (self.start\_col,)

header = ""

**if** self.type == "list" **and** self.header\_row != self.start\_row - 1:

header = " (header starts in Row **%s**)" % self.header\_row

**elif** self.type == "value" **and** self.header\_col != self.start\_col - 1:

header = " (header starts in Column **%s**)" % self.header\_col

**return** "{type} '{ident}' at {row}, {col}{head}".format(

type=self.get\_type\_display(),

ident=self.identifier,

row=row,

col=col,

head=header

)

### **class** Meta:

ordering = ('run\_id', '-type', 'start\_row', 'start\_col', 'pk')

## **class** Record(models.Model):

run = models.ForeignKey(Run, on\_delete=models.CASCADE)

content\_type = models.ForeignKey(

ContentType, null=True, blank=True, on\_delete=models.PROTECT

)

object\_id = models.PositiveIntegerField(null=True, blank=True)

content\_object = GenericForeignKey()

row = models.PositiveIntegerField()

success = models.BooleanField(default=True)

fail\_reason = models.TextField(null=True, blank=True)

### **def** \_\_str\_\_(self):

**if** self.success:

**return** "Imported '{obj}' at row {row}".format(

obj=self.content\_object,

row=self.row,

)

**else**:

**return** "Failed at row {row}".format(

row=self.row,

)

### **class** Meta:

ordering = ('run\_id', 'row')

# Data\_WIZARD.registry.py

**from** **django.core.exceptions** **import** ImproperlyConfigured

**from** **collections** **import** OrderedDict

**from** **.settings** **import** get\_setting, import\_from\_string

**from** **django.db** **import** models

## **class** Registry(object):

\_serializers = OrderedDict()

\_serializer\_names = {}

\_loaders = {}

\_loader\_classes = {}

### **def** get\_class\_name(self, serializer):

**return** "**%s**.**%s**" % (serializer.\_\_module\_\_, serializer.\_\_name\_\_)

### **def** register(self, name, serializer=None):

**if** isinstance(name, str):

**assert** serializer

**elif** isinstance(name, type) **and** issubclass(name, models.Model):

model = name

name = model.\_meta.verbose\_name.title()

**if** **not** serializer:

serializer = self.create\_serializer(model)

**else**:

**raise** **Exception**("Unexpected registration")

class\_name = self.get\_class\_name(serializer)

**if** name **in** self.\_serializers:

other\_class = self.get\_class\_name(self.\_serializers[name])

**raise** ImproperlyConfigured(

"Could not register serializer **%s**: "

"the name '**%s**' was already registered for **%s**"

% (self.get\_class\_name(serializer), name, other\_class)

)

**if** class\_name **in** self.\_serializer\_names:

other\_name = self.\_serializer\_names[class\_name]

**raise** ImproperlyConfigured(

"**%s** was already registered as **%s**"

% (class\_name, other\_name)

)

self.\_serializers[name] = serializer

self.\_serializer\_names[class\_name] = name

### **def** create\_serializer(self, model):

**from** **natural\_keys** **import** NaturalKeyModelSerializer

serializer = NaturalKeyModelSerializer.for\_model(

model,

include\_fields="\_\_all\_\_",

)

serializer.\_\_qualname\_\_ = serializer.\_\_name\_\_ = '{}Serializer'.format(

model.\_\_name\_\_

)

serializer.\_\_module\_\_ = 'data\_wizard.registry'

**return** serializer

### **def** get\_serializers(self):

serializers = []

**for** name, serializer **in** self.\_serializers.items():

serializers.append({

'name': name,

'serializer': serializer,

'class\_name': self.get\_class\_name(serializer),

'options': self.get\_serializer\_options(name),

})

**return** serializers

### **def** get\_serializer\_name(self, name):

**return** self.\_serializer\_names.get(name, name)

### **def** get\_serializer(self, name):

name = self.get\_serializer\_name(name)

**if** name **not** **in** self.\_serializers:

**raise** ImproperlyConfigured(

"**%s** is not a registered serializer!" % name

)

**return** self.\_serializers[name]

### **def** get\_serializer\_options(self, name):

serializer = self.get\_serializer(name)

meta = getattr(serializer, 'Meta', None)

**if** **not** meta:

**return** {}

options = getattr(meta, 'data\_wizard', {})

**return** options

### **def** get\_choices(self):

**return** [

(s['class\_name'], s['name'])

**for** s **in** self.get\_serializers()

]

### **def** set\_loader(self, model, loader\_name):

self.\_loaders[model] = loader\_name

### **def** get\_loader\_name(self, model):

**return** self.\_loaders.get(model, get\_setting('LOADER'))

### **def** get\_loader(self, loader\_name):

reg = self.\_loader\_classes

**if** loader\_name **in** reg:

Loader = reg[loader\_name]

**else**:

Loader = import\_from\_string(loader\_name, 'LOADER')

reg[loader\_name] = Loader

**return** Loader

registry = Registry()

# Data\_WIZARD.rest.py

**from** **rest\_framework** **import** serializers

**from** **wq.db** **import** rest

**from** **wq.db.rest.views** **import** ModelViewSet

**from** **wq.db.rest.serializers** **import** ModelSerializer

**from** **.models** **import** Run

**from** **data\_wizard** **import** views **as** wizard

*# wq.db-compatible serializers*

## **class** CurrentUserDefault(serializers.CurrentUserDefault):

### **def** \_\_call\_\_(self):

user = super(CurrentUserDefault, self).\_\_call\_\_()

**return** user.pk

## **class** RunSerializer(ModelSerializer, wizard.RunSerializer):

user\_id = serializers.HiddenField(default=CurrentUserDefault())

### **class** Meta:

exclude = ['content\_type']

## **class** RecordSerializer(wizard.RecordSerializer):

**def** get\_object\_url(self, instance):

obj = instance.content\_object

conf = rest.router.get\_model\_config(type(obj))

**if** **not** conf:

**return** None

urlbase = conf['url']

objid = getattr(obj, conf.get('lookup', 'pk'))

**return** "**%s**/**%s**" % (urlbase, objid)

## **class** RunViewSet(ModelViewSet, wizard.RunViewSet):

record\_serializer\_class = RecordSerializer

*# wq.db router registration*

## **def** user\_filter(qs, request):

**if** request.user.is\_authenticated:

**return** qs.filter(user=request.user)

**else**:

**return** qs.none()

rest.router.register\_model(

Run,

serializer=RunSerializer,

viewset=RunViewSet,

url='datawizard',

modes=[],

server\_modes=[

'list', 'detail',

'serializers', 'columns', 'ids', 'data', 'auto', 'records',

],

fields="\_\_all\_\_",

filter=user\_filter,

cache='none',

)

# Data\_WIZARD.serializers.py

**from** **rest\_framework** **import** serializers

**from** **django.contrib.contenttypes.models** **import** ContentType

**from** **.compat** **import** reverse, NoReverseMatch

**from** **.models** **import** Run, Record

**from** **data\_wizard** **import** registry

*# c.f SlugRelatedField*

## **class** ContentTypeIdField(serializers.RelatedField):

default\_error\_messages = {

'does\_not\_exist': 'Content Type {app\_label}.{model} does not exist.',

'invalid': 'Invalid value',

}

### **def** to\_internal\_value(self, data):

**try**:

app\_label, model = data.split('.')

**except** **ValueError**:

self.fail('invalid')

**try**:

**return** self.get\_queryset().get(

app\_label=app\_label,

model=model,

)

**except** ContentType.DoesNotExist:

self.fail('does\_not\_exist', app\_label=app\_label, model=model)

### **def** to\_representation(self, value):

**return** '**%s**.**%s**' % (value.app\_label, value.model)

## **class** RunSerializer(serializers.ModelSerializer):

user = serializers.HiddenField(default=serializers.CurrentUserDefault())

content\_type\_id = ContentTypeIdField(

source="content\_type",

queryset=ContentType.objects.all()

)

label = serializers.ReadOnlyField(source='\_\_str\_\_')

object\_label = serializers.StringRelatedField(

source='content\_object', read\_only=True

)

serializer\_label = serializers.ReadOnlyField()

last\_update = serializers.ReadOnlyField()

### **def** get\_fields(self):

fields = super(RunSerializer, self).get\_fields()

fields['serializer'] = serializers.ChoiceField(

choices=registry.get\_choices(),

required=False,

)

**return** fields

### **class** Meta:

model = Run

exclude = ['content\_type']

## **class** RecordSerializer(serializers.ModelSerializer):

row = serializers.SerializerMethodField()

success = serializers.ReadOnlyField()

fail\_reason = serializers.ReadOnlyField()

object\_label = serializers.SerializerMethodField()

object\_url = serializers.SerializerMethodField()

### **def** get\_row(self, instance):

**return** instance.row + 1

### **def** get\_object\_label(self, instance):

**return** str(instance.content\_object)

### **def** get\_object\_url(self, instance):

**if** **not** instance.content\_object:

**return** None

obj = instance.content\_object

**if** hasattr(obj, 'get\_absolute\_url'):

object\_url = obj.get\_absolute\_url()

**else**:

**try**:

object\_url = reverse(

'admin:{app}\_{model}\_change'.format(

app=obj.\_meta.app\_label,

model=obj.\_meta.model\_name,

),

args=[obj.pk],

)

**except** NoReverseMatch:

object\_url = None

**return** object\_url

### **class** Meta:

model = Record

fields = "\_\_all\_\_"

# Data\_WIZARD.settings.py

**from** **rest\_framework.settings** **import** import\_from\_string **as** drf\_import

**from** **django.conf** **import** settings

DEFAULTS = {

'BACKEND': 'data\_wizard.backends.threading',

'LOADER': 'data\_wizard.loaders.FileLoader',

'PERMISSION': 'rest\_framework.permissions.IsAdminUser',

}

## **def** get\_setting(name):

*# FIXME: Drop this in 2.0*

**if** getattr(settings, 'CELERY\_RESULT\_BACKEND', None):

DEFAULTS['BACKEND'] = 'data\_wizard.backends.celery'

wizard\_settings = getattr(settings, 'DATA\_WIZARD', {})

**return** wizard\_settings.get(name, DEFAULTS[name])

## **def** import\_from\_string(path, setting\_name):

**try**:

obj = drf\_import(path, setting\_name)

**except** **ImportError** **as** e:

msg = e.args[0].replace("API", "Data Wizard")

**raise** **ImportError**(msg)

**else**:

**return** obj

## **def** import\_setting(name):

path = get\_setting(name)

**return** import\_from\_string(path, name)

# Data\_WIZARD.signals.py

**from** **django.dispatch** **import** Signal

import\_complete = Signal(providing\_args=['run', 'status'])

new\_metadata = Signal(providing\_args=['run', 'identifier'])

progress = Signal(providing\_args=['run', 'status', 'meta'])

# Data\_WIZARD.tasks.py

**from** **xlrd** **import** colname

**from** **collections** **import** OrderedDict

**from** **.models** **import** Run, Identifier

**from** **.signals** **import** progress, import\_complete, new\_metadata

**from** **functools** **import** wraps

**from** **django.db** **import** transaction

**from** **django.contrib.contenttypes.models** **import** ContentType

**from** **django.contrib.auth** **import** get\_user\_model

**from** **rest\_framework** **import** serializers

**from** **natural\_keys** **import** NaturalKeySerializer

**from** **html\_json\_forms** **import** parse\_json\_form

**import** **json**

**import** **logging**

**try**:

**import** **reversion**

**except** **ImportError**:

reversion = None

User = get\_user\_model()

PRIORITY = {

'instance': 1,

'attribute': 2,

'meta': 3,

'unresolved': 4,

'unknown': 5,

}

## **def** get\_ct(model):

**return** ContentType.objects.get\_for\_model(model)

## **def** ctid(ct):

**return** '**%s**.**%s**' % (ct.app\_label, ct.model)

## **def** metaname(cls):

**return** ctid(get\_ct(cls)) + '\_meta'

## **def** get\_id(obj, field):

**if** isinstance(field, NaturalKeySerializer):

data = list(type(field)(obj).data.values())

**return** data[0]

**else**:

**return** field.to\_representation(obj)

## **def** send\_progress(sender, run):

**def** send(state, meta):

progress.send(

sender=sender,

run=run,

state=state,

meta=meta

)

**return** send

## **def** lookuprun(fn):

@wraps(fn)

**def** wrapped(run, user=None, \*\*kwargs):

**if** **not** isinstance(run, Run):

run = Run.objects.get(pk=run)

**if** user **and** **not** isinstance(user, User):

user = User.objects.get(pk=user)

**return** fn(run, user, \*\*kwargs)

**return** wrapped

@lookuprun

## **def** auto\_import(run, user):

*"""*

*Walk through all the steps necessary to interpret and import data from an*

*IO. Meant to be called asynchronously. Automatically suspends import if*

*any additional input is needed from the user.*

*"""*

send = send\_progress(auto\_import, run)

run.add\_event('auto\_import')

**if** **not** run.serializer:

result = {

'action': 'serializers',

'message': 'Input Needed'

}

send('SUCCESS', result)

**return** result

*# Preload IO to catch any load errors early*

status = {

'message': "Loading Data...",

'stage': 'meta',

'current': 1,

'total': 5,

}

send('PROGRESS', status)

run.load\_io()

*# Parse columns*

status.update(

message="Parsing Columns...",

current=2,

)

send('PROGRESS', status)

result = read\_columns(run, user)

**if** result['unknown\_count']:

result['action'] = "columns"

result['message'] = "Input Needed"

send('SUCCESS', result)

**return** result

*# Parse row identifiers*

status.update(

message="Parsing Identifiers...",

current=3,

)

send('PROGRESS', status)

result = read\_row\_identifiers(run, user)

**if** result['unknown\_count']:

result['action'] = "ids"

result['message'] = "Input Needed"

send('SUCCESS', result)

**return** result

status.update(

message="Importing Data...",

current=4,

)

send('PROGRESS', status)

*# The rest is the same as import\_data*

**return** do\_import(run, user)

## **def** get\_attribute\_field(field):

**for** cname, cfield **in** field.child.get\_fields().items():

**if** isinstance(cfield, serializers.RelatedField):

**return** cname, cfield

## **def** compute\_attr\_field(value\_field, attr\_name):

parts = value\_field.split('[')

parts[-1] = attr\_name + ']'

**return** '['.join(parts)

## **def** get\_choices(run):

**def** make\_list(choices):

**return** [{

'id': row.pk,

'label': str(row),

} **for** row **in** choices]

Serializer = run.get\_serializer()

field\_choices = set()

### **def** load\_fields(serializer, group\_name,

label\_prefix="", name\_prefix="",

attribute\_name=None, attribute\_choices=None):

fields = serializer.get\_fields().items()

**if** len(fields) == 1 **and** isinstance(serializer, NaturalKeySerializer):

is\_natkey\_lookup = True

**else**:

is\_natkey\_lookup = False

**for** name, field **in** fields:

**if** field.read\_only:

**continue**

**if** name\_prefix:

qualname = name\_prefix + ('[**%s**]' % name)

**else**:

qualname = name

label = (field.label **or** name).title()

**if** label\_prefix:

quallabel = label\_prefix + " " + label

**else**:

quallabel = label

**if** isinstance(field, NaturalKeySerializer):

load\_fields(

field, group\_name,

label\_prefix=quallabel, name\_prefix=qualname

)

**elif** isinstance(field, serializers.ListSerializer):

attr\_name, attr\_field = get\_attribute\_field(field)

**if** **not** attr\_field:

**raise** **Exception**("No attribute field found!")

choices = make\_list(attr\_field.get\_queryset())

load\_fields(

field.child,

group\_name=quallabel,

label\_prefix="",

name\_prefix=qualname + '[]',

attribute\_name=attr\_name,

attribute\_choices=choices,

)

**elif** attribute\_choices:

**if** isinstance(field, serializers.RelatedField):

**continue**

**for** choice **in** attribute\_choices:

field\_choices.add((

group\_name,

'**%s**;**%s**=**%s**' % (qualname, attribute\_name, choice['id']),

'**%s** for **%s**' % (

label, choice['label']

),

False,

field,

))

**elif** isinstance(field, serializers.ModelSerializer):

load\_fields(

field,

group\_name=quallabel,

label\_prefix="",

name\_prefix=qualname,

)

**else**:

**if** is\_natkey\_lookup:

is\_lookup = True

lookup\_field = serializer

**else**:

is\_lookup = isinstance(field, serializers.RelatedField)

lookup\_field = field

field\_choices.add(

(group\_name, qualname, quallabel, is\_lookup, lookup\_field)

)

**if** hasattr(Serializer, 'Meta') **and** hasattr(Serializer.Meta, 'model'):

root\_label = Serializer.Meta.model.\_meta.verbose\_name.title()

**else**:

root\_label = run.serializer\_label

load\_fields(Serializer(), root\_label)

field\_choices.add(

('Other', '\_\_ignore\_\_', 'Ignore this Column', False, None)

)

field\_choices = sorted(field\_choices, key=**lambda** d: d[1])

choices = [{

'id': name,

'label': label,

'is\_lookup': is\_lookup,

'group': group\_name,

'field': field,

} **for** group\_name, name, label, is\_lookup, field **in** field\_choices]

**return** choices

## **def** get\_choice\_groups(run):

choices = get\_choices(run)

groups = OrderedDict()

**for** choice **in** choices:

groups.setdefault(choice['group'], [])

groups[choice['group']].append({

'id': choice['id'],

'label': choice['label'],

})

**return** [{

'name': group,

'choices': group\_choices

} **for** group, group\_choices **in** groups.items()]

## **def** get\_choice\_ids(run):

**return** [choice['id'] **for** choice **in** get\_choices(run)]

@lookuprun

## **def** read\_columns(run, user=None):

matched = get\_columns(run)

unknown\_count = 0

**for** info **in** matched:

**if** info['type'] == 'unknown':

unknown\_count += 1

*# Add some useful context items for client*

info['unknown'] = True

info['types'] = get\_choice\_groups(run)

**assert**(info['type'] != 'unresolved')

**return** {

'columns': matched,

'unknown\_count': unknown\_count,

}

*# FIXME: These functions might make more sense as methods on Run*

## **def** get\_columns(run):

**if** run.already\_parsed():

**return** load\_columns(run)

**else**:

**return** parse\_columns(run)

## **def** get\_lookup\_columns(run):

cols = []

choices = {

choice['id']: choice

**for** choice **in** get\_choices(run)

**if** choice['is\_lookup']

}

**for** col **in** get\_columns(run):

**if** 'colnum' **not** **in** col **or** col['type'] != 'meta':

**continue**

**if** col['field\_name'] **not** **in** choices:

**continue**

col = col.copy()

info = choices[col['field\_name']]

**if** isinstance(info['field'], NaturalKeySerializer):

*# FIXME: how to override this?*

queryset = info['field'].Meta.model.objects.all()

**else**:

queryset = info['field'].get\_queryset()

col['serializer\_field'] = info['field']

col['queryset'] = queryset

cols.append(col)

**return** cols

## **def** load\_columns(run):

table = run.load\_io()

cols = list(table.field\_map.keys())

matched = []

**for** rng **in** run.range\_set.exclude(type='data'):

ident = rng.identifier

info = {

'match': str(ident),

'mapping': ident.mapping\_label,

'rel\_id': rng.pk,

'type': ident.type,

}

**if** ident.type == 'meta':

info['field\_name'] = rng.identifier.field

**elif** ident.type == 'attribute':

info['field\_name'] = rng.identifier.field

info['attr\_id'] = rng.identifier.attr\_id

info['attr\_field'] = rng.identifier.attr\_field

**else**:

info['value'] = ident.name

**if** rng.type == 'list':

col = rng.start\_col

info['name'] = cols[col].replace('**\n**', ' - ')

info['column'] = colname(col)

info['colnum'] = col

**elif** rng.type == 'value':

info['name'] = get\_range\_value(

table, rng, rng.header\_col, rng.start\_col - 1

)

info['meta\_value'] = get\_range\_value(

table, rng, rng.start\_col, rng.end\_col

)

info['colnum'] = rng.start\_col

info['rownum'] = rng.start\_row

matched.append(info)

matched.sort(key=**lambda** info: info.get('colnum', -1))

**return** matched

## **def** get\_range\_value(table, rng, scol, ecol):

**if** rng.start\_row == rng.end\_row **and** scol == ecol:

**return** table.extra\_data.get(rng.start\_row, {}).get(scol)

val = ""

**for** r **in** range(rng.start\_row, rng.end\_row + 1):

**for** c **in** range(scol, ecol + 1):

val += str(table.extra\_data.get(r, {}).get(c, ""))

**return** val

## **def** parse\_columns(run):

run.add\_event('parse\_columns')

table = run.load\_io()

**if** table.tabular:

**for** r **in** table.extra\_data:

row = table.extra\_data[r]

**for** c **in** row:

**if** c + 1 **in** row **and** c - 1 **not** **in** row:

parse\_column(

run,

row[c],

type='value',

start\_row=r,

end\_row=r,

header\_col=c,

start\_col=c + 1,

end\_col=c + 1,

)

**for** i, name **in** enumerate(table.field\_map.keys()):

**if** table.tabular:

header\_row = table.header\_row

start\_row = table.start\_row

**else**:

header\_row = -1

start\_row = 0

name = table.clean\_field\_name(name)

parse\_column(

run,

name=name,

type='list',

header\_row=header\_row,

start\_row=start\_row,

end\_row=start\_row + len(table) - 1,

start\_col=i,

end\_col=i,

)

**return** load\_columns(run)

## **def** parse\_column(run, name, \*\*kwargs):

matches = list(Identifier.objects.filter(

serializer=run.serializer,

name\_\_iexact=name,

))

**if** len(matches) > 0:

matches.sort(

key=**lambda** ident: PRIORITY.get(ident.type, 0)

)

ident = matches[0]

**else**:

**if** name **in** get\_choice\_ids(run):

field = name

**else**:

field = None

ident = Identifier.objects.create(

serializer=run.serializer,

name=name,

field=field,

resolved=(field **is** **not** None),

)

run.range\_set.create(

identifier=ident,

\*\*kwargs

)

@lookuprun

## **def** update\_columns(run, user, post={}):

run.add\_event('update\_columns')

matched = get\_columns(run)

**for** col **in** matched:

**if** col['type'] != 'unknown':

**continue**

val = post.get('rel\_**%s**' % col['rel\_id'], None)

**if** **not** val:

**continue**

ident = run.range\_set.get(pk=col['rel\_id']).identifier

**assert**(ident.field **is** None)

**if** val **not** **in** get\_choice\_ids(run):

**continue**

**if** ';' **in** val:

field, attr\_info = val.split(';')

attr\_name, attr\_id = attr\_info.split('=')

attr\_field = compute\_attr\_field(field, attr\_name)

**else**:

field = val

attr\_id = None

attr\_field = None

ident.field = field

ident.attr\_id = attr\_id

ident.attr\_field = attr\_field

ident.resolved = True

ident.save()

new\_metadata.send(

sender=update\_columns,

run=run,

identifier=ident,

)

**return** read\_columns(run)

@lookuprun

## **def** read\_row\_identifiers(run, user=None):

**if** run.range\_set.filter(type='data').exists():

**return** load\_row\_identifiers(run)

**else**:

**return** parse\_row\_identifiers(run)

## **def** parse\_row\_identifiers(run):

run.add\_event('parse\_row\_identifiers')

lookup\_cols = get\_lookup\_columns(run)

lookup\_fields = OrderedDict()

**for** col **in** lookup\_cols:

field\_name = col['field\_name']

lookup\_fields.setdefault(field\_name, {

'cols': [],

'ids': OrderedDict(),

'start\_col': 1e10,

'end\_col': -1,

})

info = lookup\_fields[field\_name]

info['cols'].append(col)

info['start\_col'] = min(info['start\_col'], col['colnum'])

info['end\_col'] = max(info['end\_col'], col['colnum'])

**if** 'meta\_value' **in** col:

info['is\_meta\_value'] = True

info['ids'] = {

col['meta\_value']: {

'count': 1,

'start\_row': col['rownum'],

'end\_row': col['rownum'],

}

}

**assert**(info['start\_col'] < 1e10)

**assert**(info['end\_col'] > -1)

table = run.load\_io()

**for** i, row **in** enumerate(table):

**for** field\_name, info **in** lookup\_fields.items():

**if** 'is\_meta\_value' **in** info:

**continue**

names = [str(row[col['colnum']]) **for** col **in** info['cols']]

name = " ".join(names)

info['ids'].setdefault(name, {

'count': 0,

'start\_row': 1e10,

'end\_row': -1,

})

idinfo = info['ids'][name]

idinfo['count'] += 1

rownum = i

**if** table.tabular:

rownum += table.start\_row

idinfo['start\_row'] = min(idinfo['start\_row'], rownum)

idinfo['end\_row'] = max(idinfo['end\_row'], rownum)

**assert**(idinfo['start\_row'] < 1e10)

**assert**(idinfo['end\_row'] > -1)

**for** field\_name, info **in** lookup\_fields.items():

**for** name, idinfo **in** info['ids'].items():

ident = Identifier.objects.filter(

serializer=run.serializer,

field=field\_name,

name\_\_iexact=name,

).first()

**if** **not** ident:

ident = Identifier.objects.create(

serializer=run.serializer,

field=field\_name,

name=name,

)

run.range\_set.create(

type='data',

identifier=ident,

start\_col=info['start\_col'],

end\_col=info['end\_col'],

start\_row=idinfo['start\_row'],

end\_row=idinfo['end\_row'],

count=idinfo['count'],

)

**return** load\_row\_identifiers(run)

## **def** load\_row\_identifiers(run):

ids = {}

lookup\_cols = get\_lookup\_columns(run)

**for** rng **in** run.range\_set.filter(type='data'):

ident = rng.identifier

info = None

**for** col **in** lookup\_cols:

**if** col['field\_name'] == ident.field:

info = col

**if** **not** info:

**continue**

model = info['queryset'].model

ids.setdefault(model, {})

ids[model][ident] = rng.count, info

unknown\_ids = 0

idgroups = []

**for** model **in** ids:

mtype = get\_ct(model)

idinfo = {

'type\_id': ctid(mtype),

'type\_label': mtype.name.title(),

'ids': []

}

**for** ident, (count, col) **in** ids[model].items():

info = {

'value': ident.name,

'count': count,

}

**if** ident.resolved:

info['match'] = ident.value **or** ident.name

**else**:

**assert**(ident.type == 'unresolved')

unknown\_ids += 1

field = col['serializer\_field']

info['ident\_id'] = ident.pk

info['unknown'] = True

info['choices'] = [{

'id': get\_id(choice, field),

'label': str(choice),

} **for** choice **in** col['queryset']]

**if** isinstance(field, NaturalKeySerializer):

info['choices'].insert(0, {

'id': 'new',

'label': "New **%s**" % idinfo['type\_label'],

})

idinfo['ids'].append(info)

idinfo['ids'].sort(key=**lambda** info: info['value'])

idgroups.append(idinfo)

**return** {

'unknown\_count': unknown\_ids,

'types': idgroups,

}

@lookuprun

## **def** update\_row\_identifiers(run, user, post={}):

run.add\_event('update\_row\_identifiers')

unknown = run.range\_set.filter(

type='data',

identifier\_\_resolved=False,

)

**for** rng **in** unknown:

ident = rng.identifier

ident\_id = post.get('ident\_**%s**\_id' % ident.pk, None)

**if** **not** ident\_id:

**continue**

**if** ident\_id == 'new':

ident.value = ident.name

**else**:

ident.value = ident\_id

ident.resolved = True

ident.save()

new\_metadata.send(

sender=update\_row\_identifiers,

run=run,

identifier=ident,

)

**return** read\_row\_identifiers(run, user)

@lookuprun

## **def** import\_data(run, user):

*"""*

*Import all parseable data from the dataset instance's IO class.*

*"""*

result = do\_import(run, user)

**return** result

## **def** do\_import(run, user):

**if** reversion:

**with** reversion.create\_revision():

reversion.set\_user(user)

reversion.set\_comment('Imported via **%s**' % run)

result = \_do\_import(run, user)

**else**:

result = \_do\_import(run, user)

**return** result

## **def** \_do\_import(run, user):

send = send\_progress(import\_data, run)

run.add\_event('do\_import')

*# (Re-)Load data and column information*

table = run.load\_io()

matched = get\_columns(run)

*# Set global defaults for metadata values*

**if** **not** user.is\_authenticated:

user = None

run\_globals = {

*# Metadata fields*

}

*# Set any global defaults defined within data themselves (usually as extra*

*# cells above the headers in a spreadsheet)*

**for** col **in** matched:

**if** 'meta\_value' **in** col:

save\_value(col, col['meta\_value'], run\_globals)

**elif** 'attr\_id' **in** col **and** **not** col.get('attr\_field'):

*# FIXME: Drop this in 2.0*

Serializer = run.get\_serializer()

basename = col['field\_name'].split('[')[0]

field = Serializer().get\_fields().get(basename)

**if** field:

attr\_name, attr\_field = get\_attribute\_field(field)

col['attr\_field'] = '**%s**[][**%s**]' % (

basename, attr\_name

)

*# Loop through table rows and add each record*

rows = len(table)

skipped = []

**if** table.tabular:

**def** rownum(i):

**return** i + table.start\_row

**else**:

**def** rownum(i):

**return** i

**for** i, row **in** enumerate(table):

*# Update state (for status() on view)*

send('PROGRESS', {

'message': "Importing Data...",

'stage': 'data',

'current': i,

'total': rows,

'skipped': skipped

})

*# Create report, capturing any errors*

obj, error = import\_row(run, i, row, run\_globals, matched)

**if** error:

success = False

fail\_reason = error

skipped.append({'row': rownum(i) + 1, 'reason': fail\_reason})

**else**:

success = True

fail\_reason = None

*# Record relationship between data source and resulting report (or*

*# skipped record), including specific cell range.*

run.record\_set.create(

row=rownum(i),

content\_object=obj,

success=success,

fail\_reason=fail\_reason

)

*# Send completion signal (in case any server handlers are registered)*

status = {

'current': i + 1,

'total': rows,

'skipped': skipped

}

run.add\_event('import\_complete')

run.record\_count = run.record\_set.filter(success=True).count()

run.save()

send('SUCCESS', status)

import\_complete.send(sender=import\_data, run=run, status=status)

**return** status

## **def** import\_row(run, i, row, instance\_globals, matched):

*"""*

*Create actual report instance from parsed values.*

*"""*

*# Copy global values to record hash*

record = {

key: instance\_globals[key]

**for** key **in** instance\_globals

}

**for** col **in** matched:

**if** 'colnum' **in** col **and** 'meta\_value' **not** **in** col:

val = row[col['colnum']]

save\_value(col, val, record)

seen = set()

**for** col **in** matched:

field\_name = col['field\_name']

**if** col['type'] == 'meta' **and** field\_name **not** **in** seen:

seen.add(field\_name)

ident = Identifier.objects.filter(

serializer=run.serializer,

name\_\_iexact=str(record[field\_name]),

).first()

**if** ident **and** ident.value:

record[field\_name] = ident.value

record.pop('\_attr\_index', None)

Serializer = run.get\_serializer()

**try**:

serializer = Serializer(

data=parse\_json\_form(record),

context={

'data\_wizard': {

'run': run,

}

},

)

**if** serializer.is\_valid():

**with** transaction.atomic():

obj = serializer.save()

error = None

**else**:

obj = None

error = json.dumps(serializer.errors)

**except** **Exception** **as** e:

logging.warning(

"{run}: Error In Row {row}".format(

run=run,

row=i,

)

)

logging.exception(e)

obj = None

error = repr(e)

**return** obj, error

## **def** save\_value(col, val, obj):

*"""*

*For each cell in each row, use parsed col(umn) information to determine how*

*to apply the cell val(ue) to the obj(ect hash).*

*"""*

*# In some spreadsheets (i.e. "horizontal" tables), multiple columns*

*# indicate attribute names and each row contains result values. In others*

*# (i.e. "vertical" tables), each row lists both the attribute name and the*

*# value.*

**if** col['type'] == "attribute":

*# Attribute value in a "horizontal" table*

save\_attribute\_value(col, val, obj)

**elif** col['type'] == "meta":

*# Metadata value in either a "horizontal" or "vertical" table*

set\_value(obj, col['field\_name'], val)

## **def** save\_attribute\_value(col, val, obj):

*"""*

*This column was identified as an EAV attribute; update nested array with*

*the cell value from this row.*

*"""*

**if** 'attr\_field' **not** **in** col:

**raise** **Exception**("Unexpected EAV value!")

**if** '\_attr\_index' **not** **in** obj:

obj['\_attr\_index'] = {

col['attr\_id']: 0

}

**else**:

obj['\_attr\_index'].setdefault(

col['attr\_id'], (max(obj['\_attr\_index'].values()) **or** 0) + 1

)

index = obj['\_attr\_index'][col['attr\_id']]

value\_field = col['field\_name'].replace('[]', '[**%s**]' % index)

attr\_field = col['attr\_field'].replace('[]', '[**%s**]' % index)

set\_value(obj, value\_field, val)

obj[attr\_field] = col['attr\_id']

**def** set\_value(obj, field\_name, val):

**if** field\_name **in** obj:

val = "**%s** **%s**" % (obj[field\_name], val)

obj[field\_name] = val

# Data\_WIZARD.test.py

**from** **\_\_future\_\_** **import** print\_function *# FIXME: Drop this in 2.0*

**from** **rest\_framework.test** **import** APITransactionTestCase

**from** **rest\_framework** **import** status

**import** **os**

**from** **time** **import** sleep

**from** **django.contrib.contenttypes.models** **import** ContentType

**from** **django.contrib.auth.models** **import** User

**from** **django.core.files** **import** File

**from** **data\_wizard.models** **import** Run, Identifier

**from** **django.conf** **import** settings

## **class** WizardTestCase(APITransactionTestCase):

serializer\_name = None

attr\_field = None

with\_wqdb = True

file\_url = None

file\_model = None

file\_content\_type = None

### **def** \_fixture\_teardown(self):

*# \_fixture\_teardown truncates related tables including contenttypes*

*# (even though that table is populated before the test runs)*

content\_types = list(ContentType.objects.all())

super(WizardTestCase, self).\_fixture\_teardown()

ContentType.objects.bulk\_create(content\_types)

### **def** setUp(self):

self.user = User.objects.create(

username='testuser',

is\_staff=True,

is\_superuser=True,

)

self.client.force\_authenticate(user=self.user)

### **def** mkurl(self, run, action):

**if** self.with\_wqdb:

template = "/datawizard/**%s**/**%s**.json"

**else**:

template = "/datawizard/**%s**/**%s**/?format=json"

**return** template % (run.pk, action)

### **def** get\_url(self, run, action, params={}):

**return** self.client.get(self.mkurl(run, action), params)

### **def** post\_url(self, run, action, post):

**return** self.client.post(self.mkurl(run, action), post)

### **def** wait(self, run, action):

**print**()

response = self.post\_url(run, action, None)

self.assertIn("task\_id", response.data)

status\_params = {'task': response.data['task\_id']}

done = False

**while** **not** done:

sleep(1)

response = self.get\_url(run, 'status', status\_params)

res = response.data

**if** res.get('status', None) **in** ("PENDING", "PROGRESS"):

**print**(res)

**else**:

done = True

**return** res

### **def** create\_identifier(self, name, field, value=None, attr\_id=None):

*"""*

*0. Preregister any necessary identifiers*

*"""*

**if** attr\_id:

attr\_field = self.attr\_field

**else**:

attr\_field = None

Identifier.objects.create(

serializer=self.serializer\_name,

name=name,

field=field,

value=value,

attr\_id=attr\_id,

attr\_field=attr\_field,

resolved=True,

)

### **def** upload\_file(self, filename, skip\_serializer=False):

*"""*

*1. Upload spreadsheet file*

*"""*

filename = os.path.join(settings.MEDIA\_ROOT, filename)

**with** open(filename, 'rb') **as** f:

**if** self.with\_wqdb:

response = self.client.post(self.file\_url, {'file': f})

self.assertEqual(response.status\_code, status.HTTP\_201\_CREATED)

file\_id = response.data['id']

**else**:

file\_id = self.file\_model.objects.create(file=File(f)).pk

post = {

'content\_type\_id': self.file\_content\_type,

'object\_id': file\_id,

}

**if** **not** skip\_serializer:

post['serializer'] = self.serializer\_name

response = self.client.post('/datawizard/?format=json', post)

self.assertEqual(

response.status\_code, status.HTTP\_201\_CREATED, response.data

)

run = Run.objects.get(pk=response.data['id'])

**return** run

### **def** download\_url(self, url, skip\_serializer=False):

*"""*

*1. Download data from url*

*"""*

url\_id = self.url\_model.objects.create(url=url).pk

post = {

'content\_type\_id': self.url\_content\_type,

'object\_id': url\_id,

}

**if** **not** skip\_serializer:

post['serializer'] = self.serializer\_name

response = self.client.post('/datawizard/?format=json', post)

self.assertEqual(

response.status\_code, status.HTTP\_201\_CREATED, response.data

)

run = Run.objects.get(pk=response.data['id'])

**return** run

### **def** set\_serializer(self, run):

*"""*

*1b. Set serializer class*

*"""*

response = self.get\_url(run, 'serializers')

found = False

**for** choice **in** response.data.get('serializer\_choices'):

**if** choice['name'] == self.serializer\_name:

found = True

self.assertTrue(found)

response = self.post\_url(run, 'updateserializer', {

'serializer': self.serializer\_name,

})

self.assertEqual(response.data.get('serializer'), self.serializer\_name)

### **def** check\_columns(self, run, expect\_columns, expect\_unknown):

*"""*

*2. Start import process by verifying columns*

*"""*

response = self.get\_url(run, 'columns')

self.assertIn('result', response.data)

self.assertIn('columns', response.data['result'])

self.assertEqual(

len(response.data['result']['columns']), expect\_columns

)

self.assertEqual(

response.data['result'].get('unknown\_count', 0), expect\_unknown

)

### **def** update\_columns(self, run, mappings):

*"""*

*3. Inspect unmatched columns and select choices*

*"""*

response = self.get\_url(run, 'columns')

post = {}

**for** col **in** response.data['result']['columns']:

**if** **not** col.get('unknown', False):

**continue**

self.assertIn('types', col)

type\_choices = {

tc['name']: tc['choices'] **for** tc **in** col['types']

}

**for** type\_name, mapping **in** mappings.items():

self.assertIn(type\_name, type\_choices)

*# "Choose" options from dropdown menu choices*

col\_id = mapping.get(col['name'])

**if** col\_id **is** None:

**continue**

found = False

**for** choice **in** type\_choices[type\_name]:

**if** choice['id'] == col\_id:

found = True

self.assertTrue(

found,

col\_id + " not found in choices: **%s**" %

type\_choices[type\_name]

)

post["rel\_**%s**" % col['rel\_id']] = col\_id

response = self.post\_url(run, 'updatecolumns', post)

unknown = response.data['result']['unknown\_count']

self.assertFalse(unknown, "**%s** unknown columns remain" % unknown)

### **def** check\_row\_identifiers(self, run, expect\_identifiers, expect\_unknown):

*"""*

*4. Verify identifier (foreign key) values*

*"""*

response = self.get\_url(run, 'ids')

self.assertIn('result', response.data)

self.assertIn('types', response.data['result'])

all\_ids = sum([

len(group['ids'])

**for** group **in** response.data['result']['types']

])

self.assertEqual(expect\_identifiers, all\_ids)

self.assertEqual(

expect\_unknown, response.data['result'].get('unknown\_count', 0)

)

### **def** update\_row\_identifiers(self, run, mappings):

*"""*

*5. Inspect unmatched identifiers and select choices*

*"""*

response = self.get\_url(run, 'ids')

type\_ids = {

t['type\_id']: t['ids']

**for** t **in** response.data['result']['types']

}

post = {}

**for** typeid, mapping **in** mappings.items():

self.assertIn(typeid, type\_ids)

**for** idinfo **in** type\_ids[typeid]:

**if** idinfo['value'] **in** mapping:

post[

'ident\_**%s**\_id' % idinfo['ident\_id']

] = mapping[idinfo['value']]

*# 7. Post selected options, verify that all identifiers are now known*

response = self.post\_url(run, 'updateids', post)

unknown = response.data['result']['unknown\_count']

self.assertFalse(unknown, "**%s** unknown identifiers remain" % unknown)

### **def** start\_import(self, run, expect\_skipped):

*"""*

*6. Start data import process, wait for completion*

*"""*

res = self.wait(run, 'data')

**for** key **in** ('status', 'total', 'current', 'skipped'):

self.assertIn(key, res)

self.assertEqual('SUCCESS', res['status'])

*# FIXME: Drop this in 2.0*

**for** skipped **in** res['skipped']:

skipped['reason'] = skipped['reason'].replace(

'YYYY[-MM[-DD]]', 'YYYY-MM-DD'

)

self.assertEqual(expect\_skipped, res['skipped'])

### **def** auto\_import(self, run, expect\_input\_required=False):

*"""*

*Test the auto import (steps 2-6)*

*"""*

res = self.wait(run, 'auto')

self.assertEqual(res['status'], "SUCCESS")

**if** expect\_input\_required:

self.assertIn('message', res)

**return** res

self.assertNotIn('message', res, res.get('message'))

**for** key **in** ('status', 'total', 'current', 'skipped'):

self.assertIn(key, res)

**return** res

### **def** assert\_status(self, run, expect\_count):

*"""*

*7. Verify record count, loader and serializer*

*"""*

run = Run.objects.get(pk=run.pk)

self.assertEqual(expect\_count, run.record\_count)

self.assertTrue(run.loader)

self.assertEqual(self.serializer\_name, run.serializer)

### **def** assert\_ranges(self, run, expect\_ranges):

*"""*

*8. Verify column and identifier ranges*

*"""*

ranges = [

str(rng)

**for** rng **in** run.range\_set.all()

]

self.assertEqual(expect\_ranges, ranges)

### **def** assert\_records(self, run, expect\_records):

*"""*

*9. Verify column and identifier ranges*

*"""*

**def** make\_str(record):

text = str(record)

**if** text.startswith('Failed'):

text += ": " + record.fail\_reason

*# FIXME: Drop PY2/DRF<3.9 support in 2.0*

text = (

text.replace('[u"', '["') *# Python 2.7*

.replace("YYYY[-MM[-DD]]", "YYYY-MM-DD") *# DRF < 3.9*

.replace(',)', ')') *# Python < 3.7*

)

**return** text

records = [

make\_str(record)

**for** record **in** run.record\_set.all()

]

self.assertEqual(expect\_records, records)

### **def** assert\_log(self, run, expect\_log):

*"""*

*10. Verify expected process was followed*

*"""*

steps = [log.event **for** log **in** run.log.all()]

self.assertEqual(expect\_log, steps)

### **def** assert\_urls(self, run, urltemplate):

**if** **not** self.with\_wqdb:

**return**

records = self.get\_url(run, 'records').data['records']

**for** row, record **in** zip(records, run.record\_set.all()):

**if** record.success:

self.assertEqual(

urltemplate % record.object\_id,

row.get('object\_url'),

)

# Data\_WIZARD.urls.py

**from** **rest\_framework** **import** routers

**from** **.views** **import** RunViewSet

router = routers.SimpleRouter()

router.register(r'', RunViewSet)

app\_name = 'data\_wizard'

urlpatterns = router.urls

# Data\_WIZARD.views.py

**from** **rest\_framework.response** **import** Response

**from** **rest\_framework.decorators** **import** detail\_route

**from** **rest\_framework.viewsets** **import** ModelViewSet

**from** **rest\_framework.pagination** **import** PageNumberPagination

**from** **rest\_framework** **import** renderers

**from** **data\_wizard** **import** registry

**import** **data\_wizard**

**from** **.serializers** **import** RunSerializer, RecordSerializer

**from** **.models** **import** Run

**from** **.settings** **import** import\_setting

## **class** PageNumberPagination(PageNumberPagination):

page\_size = 50

## **class** RunViewSet(ModelViewSet):

serializer\_class = RunSerializer

pagination\_class = PageNumberPagination

renderer\_classes = [

renderers.TemplateHTMLRenderer,

renderers.JSONRenderer,

renderers.BrowsableAPIRenderer,

]

permission\_classes = [

import\_setting('PERMISSION'),

]

record\_serializer\_class = RecordSerializer

queryset = Run.objects.all()

@property

### **def** backend(self):

**return** data\_wizard.backend

@property

### **def** template\_name(self):

**if** self.action == 'retrieve':

template = 'detail'

**else**:

template = self.action

**return** 'data\_wizard/run\_{}.html'.format(template)

### **def** get\_renderers(self):

**if** self.action == 'status':

**return** [renderers.JSONRenderer()]

**else**:

**return** super(RunViewSet, self).get\_renderers()

@detail\_route()

### **def** status(self, request, \*args, \*\*kwargs):

task\_id = request.GET.get('task', None)

result = self.backend.get\_async\_status(task\_id)

status = result.get('status', 'UNKNOWN')

action = result.get('action', None)

**if** **not** action **and** status == 'SUCCESS':

action = 'records'

**if** action:

url = '/datawizard/{pk}/{action}'.format(

pk=self.get\_object().pk,

action=action,

)

result['location'] = url

**elif** status == 'FAILURE' **and** **not** result.get('error'):

result['error'] = "Unknown Error"

result['status'] = status

**return** Response(result)

### **def** run\_task(self, name, use\_async=False, post=None):

run = self.get\_object()

**return** run.run\_task(

name,

use\_async=use\_async,

post=post,

backend=self.backend,

user=self.request.user,

)

### **def** retrieve\_and\_run(self, task\_name, use\_async=False, post=None):

response = self.retrieve(self.request, \*\*self.kwargs)

result = self.run\_task(task\_name, use\_async, post)

response.data.update(result)

**return** response

@detail\_route()

### **def** serializers(self, request, \*args, \*\*kwargs):

response = self.retrieve(request, \*\*self.kwargs)

response.data['serializer\_choices'] = [

{

'name': s['class\_name'],

'label': s['name'],

} **for** s **in** registry.get\_serializers()

**if** s['options'].get('show\_in\_list', True)

]

**return** response

@detail\_route(methods=['post'])

### **def** updateserializer(self, request, \*args, \*\*kwargs):

run = self.get\_object()

self.action = 'serializers'

name = request.POST.get('serializer', None)

**if** name **and** registry.get\_serializer(name):

run.serializer = name

run.save()

run.add\_event('update\_serializer')

**return** self.serializers(request)

@detail\_route()

### **def** columns(self, request, \*args, \*\*kwargs):

**return** self.retrieve\_and\_run('read\_columns')

@detail\_route(methods=['post'])

### **def** updatecolumns(self, request, \*args, \*\*kwargs):

response = self.retrieve\_and\_run('read\_columns')

self.action = 'columns'

result = self.run\_task('update\_columns', post=request.POST)

response.data.update(result)

**return** response

@detail\_route()

### **def** ids(self, request, \*args, \*\*kwargs):

**return** self.retrieve\_and\_run('read\_row\_identifiers')

@detail\_route(methods=['post'])

### **def** updateids(self, request, \*args, \*\*kwargs):

response = self.retrieve\_and\_run('read\_row\_identifiers')

self.action = 'ids'

result = self.run\_task('update\_row\_identifiers', post=request.POST)

response.data.update(result)

**return** response

@detail\_route(methods=['post'])

### **def** data(self, request, \*args, \*\*kwargs):

**return** self.retrieve\_and\_run('import\_data', use\_async=True)

@detail\_route(methods=['post', 'get'])

### **def** auto(self, request, \*args, \*\*kwargs):

**if** request.method == 'GET':

self.action = 'retrieve'

**return** self.retrieve(request, \*\*kwargs)

**return** self.retrieve\_and\_run('auto\_import', use\_async=True)

@detail\_route()

### **def** records(self, request, \*args, \*\*kwargs):

response = self.retrieve(self.request, \*\*kwargs)

response.data['records'] = self.record\_serializer\_class(

self.get\_object().record\_set.all(),

many=True

).data

**return** response

# Data\_WIZARD.Management.commands.runwizard.py

**from** **django.core.management.base** **import** BaseCommand, CommandError

**from** **data\_wizard.models** **import** Run

**from** **django.contrib.contenttypes.models** **import** ContentType

**from** **data\_wizard.serializers** **import** ContentTypeIdField

**from** **data\_wizard.backends** **import** immediate

**from** **rest\_framework.exceptions** **import** ValidationError

**from** **django.contrib.auth** **import** get\_user\_model

**import** **getpass**

User = get\_user\_model()

## **class** Command(BaseCommand):

### **def** add\_arguments(self, parser):

parser.add\_argument('contenttype\_id')

parser.add\_argument('object\_id')

parser.add\_argument('--serializer')

parser.add\_argument('--loader')

parser.add\_argument('--username')

parser.add\_argument('--quiet', action="store\_true")

### **def** handle(self, \*args, \*\*options):

ctid = options['contenttype\_id']

**try**:

ct = ContentTypeIdField(

queryset=ContentType.objects.all()

).to\_internal\_value(ctid)

**except** ValidationError **as** e:

**raise** CommandError(e)

objid = options['object\_id']

**try**:

ct.get\_object\_for\_this\_type(pk=objid)

**except** ct.model\_class().DoesNotExist:

**raise** CommandError(

"Could not find {} with pk={}".format(ct, objid)

)

username = options['username'] **or** getpass.getuser()

**try**:

user = User.objects.get(\*\*{

User.USERNAME\_FIELD: username

})

**except** User.DoesNotExist:

**raise** CommandError("No such user '{}'".format(username))

run = Run.objects.create(

user=user,

content\_type=ct,

object\_id=objid,

serializer=options['serializer'],

loader=options['loader'],

)

**if** **not** run.serializer:

**raise** CommandError("No serializer specified.")

**try**:

result = run.run\_task(

'auto\_import',

use\_async=False,

backend=immediate.Backend(),

)

**except** **Exception** **as** e:

**raise** CommandError(e)

**if** 'error' **in** result:

**raise** CommandError(result['error'])

**elif** 'result' **in** result:

result = result['result']

**if** 'action' **in** result:

*# TODO: Interactive CLI for resolving input?*

**raise** CommandError(

"{message} for {unknown\_count} {action}".format(

message=result.get('message', 'Input Needed'),

unknown\_count=result.get('unknown\_count', '?'),

action=result['action'],

)

)

**if** **not** options['quiet']:

self.stdout.write(

"{total} row imported ({skipped} skipped).".format(

total=result['total'],

skipped=len(result['skipped']),

)

)

# Data\_WIZARD.Sources.admin.py

**from** **django.contrib** **import** admin

**from** **.models** **import** FileSource, URLSource

admin.site.register(FileSource)

admin.site.register(URLSource)

# Data\_WIZARD.Sources.apps.py

**from** **django.apps** **import** AppConfig

## **class** SourcesConfig(AppConfig):

name = 'data\_wizard.sources'

verbose\_name = 'Data Wizard - Sources'

# Data\_WIZARD.Sources.models.py

**from** **django.db** **import** models

## **class** FileSource(models.Model):

name = models.CharField(max\_length=255, null=True, blank=True)

file = models.FileField(upload\_to='datawizard/')

date = models.DateTimeField(auto\_now\_add=True)

### **def** \_\_str\_\_(self):

**return** self.name **or** self.file.name

## **class** URLSource(models.Model):

name = models.CharField(max\_length=255, null=True, blank=True)

url = models.URLField()

date = models.DateTimeField(auto\_now\_add=True)

### **def** \_\_str\_\_(self):

**return** self.name **or** self.url

# Data\_WIZARD.Sources.rest.py

**from** **wq.db** **import** rest

**from** **.models** **import** FileSource, URLSource

rest.router.register\_model(FileSource, fields="\_\_all\_\_")

rest.router.register\_model(URLSource, fields="\_\_all\_\_")

# Data\_WIZARD.Sources.wizard.py

**import** **data\_wizard**

**from** **.models** **import** FileSource, URLSource

data\_wizard.set\_loader(FileSource, 'data\_wizard.loaders.FileLoader')

data\_wizard.set\_loader(URLSource, 'data\_wizard.loaders.URLLoader')

DJANGO WAREHOUSE

# Manage.py

*#!/usr/bin/env python*

**import** **os**, **sys**

**if** \_\_name\_\_ == "\_\_main\_\_":

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "warehouse.settings")

**from** **django.core.management** **import** execute\_from\_command\_line

execute\_from\_command\_line(sys.argv)

# Warehouse.settings.py

**import** **os**

**import** **sys**

DEBUG = True

TEMPLATE\_DEBUG = DEBUG

PROJECT\_ROOT = os.path.dirname(\_\_file\_\_)

sys.path.insert(0, PROJECT\_ROOT)

ADMINS = (

('Steven Challis', 'steve@stevechallis.com'),

)

MANAGERS = ADMINS

TEST\_RUNNER = 'django.test.runner.DiscoverRunner'

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': 'warehouse',

'USER': os.environ.get('APPSETTING\_DB\_USER'),

'PASSWORD': os.environ.get('APPSETTING\_DB\_PASSWORD'),

'HOST': os.environ.get('APPSETTING\_DB\_HOST'),

'PORT': '',

}

}

ALLOWED\_HOSTS = ['\*']

TIME\_ZONE = 'Europe/London'

LANGUAGE\_CODE = 'en-gb'

SITE\_ID = 1

USE\_I18N = True

USE\_L10N = False

DATETIME\_FORMAT = 'N j, Y, P (T)'

MEDIA\_ROOT = ''

MEDIA\_URL = ''

**if** os.environ.get('APPSETTING\_LOCAL\_STATIC') == 'true':

STATIC\_ROOT = 'static/'

STATIC\_URL = '/static/'

**else**:

STATIC\_ROOT = 'D:/home/site/wwwroot/static'

STATIC\_URL = 'https://portalvhds52l58tfthh6wl.blob.core.windows.net/staticfiles/'

STATICFILES\_STORAGE = 'azure\_storage.storage.AzureStorage'

ADMIN\_MEDIA\_PREFIX = '/static/admin/'

STATICFILES\_DIRS = ()

STATICFILES\_FINDERS = (

'django.contrib.staticfiles.finders.FileSystemFinder',

'django.contrib.staticfiles.finders.AppDirectoriesFinder',

)

SECRET\_KEY = '((@b3f+s!o8^r$wb$p8@+&8k@&y\*3o7sy3dp4xk+45=$e2996x'

TEMPLATE\_LOADERS = (

'django.template.loaders.filesystem.Loader',

'django.template.loaders.app\_directories.Loader',

)

MIDDLEWARE\_CLASSES = (

'django.middleware.common.CommonMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

)

ROOT\_URLCONF = 'warehouse.urls'

TEMPLATE\_DIRS = ()

INSTALLED\_APPS = (

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.sites',

'django.contrib.messages',

'django.contrib.staticfiles',

'django.contrib.admin',

'debug\_toolbar',

'reporting',

)

*#INTERNAL\_IPS = ('127.0.0.1', )*

LOGGING = {

'version': 1,

'disable\_existing\_loggers': False,

'formatters':{

'verbose': {

'format': '[**%(levelname)s** **%(threadName)s** **%(asctime)s** **%(module)s**] **%(message)s**'

},

'simple': {

'format': '**%(levelname)s** **%(message)s**'

},

},

'handlers': {

'console': {

'level': 'DEBUG',

'class': 'logging.StreamHandler',

'formatter': 'verbose',

'stream': sys.stdout

},

},

'loggers': {

'reporting': {

'handlers': ['console'],

'level': 'DEBUG',

'propagate': True,

},

}

}

BORK\_URL = 'http://api.zonza.tv:8080/v0/'

BORK\_AUTH = {

'Bork-Token': os.environ.get('APPSETTING\_BORK\_TOKEN'),

'Bork-Username': os.environ.get('APPSETTING\_BORK\_USERNAME'),

}

AZURE\_STORAGE = {

'ACCOUNT\_NAME': os.environ.get('APPSETTING\_STORAGE\_ACCOUNT\_NAME'),

'ACCOUNT\_KEY': os.environ.get('APPSETTING\_STORAGE\_ACCOUNT\_KEY'),

'CONTAINER': 'staticfiles',

'STATIC\_CONTAINER': 'static',

'CDN\_HOST': None,

'USE\_SSL': False,

}

# Warehouse.urls.py

**from** **django.conf.urls** **import** patterns, include, url

**from** **django.contrib** **import** admin

admin.autodiscover()

**import** **debug\_toolbar**

**from** **reporting** **import** urls **as** reporting\_urls

admin.site.site\_header = 'ZONZA Reporting'

urlpatterns = patterns('',

url(r'^admin/', include(admin.site.urls)),

url(r'^', include(reporting\_urls)),

)

# Warehouse.wsgi.py

*"""*

*WSGI config for warehouse project.*

*It exposes the WSGI callable as a module-level variable named ``application``.*

*For more information on this file, see*

*https://docs.djangoproject.com/en/1.7/howto/deployment/wsgi/*

*"""*

**import** **os**

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "warehouse.settings")

**from** **django.core.wsgi** **import** get\_wsgi\_application

application = get\_wsgi\_application()

# Warehouse.Reporting.admin.py

**from** **django.contrib** **import** admin

**from** **django.contrib.auth.models** **import** User, Group

**from** **django.core.urlresolvers** **import** reverse

**from** **django.utils.safestring** **import** mark\_safe

**from** **django.db.models** **import** Sum

**from** **reporting** **import** models

**from** **django.contrib** **import** admin

**from** **django.contrib.admin.util** **import** flatten\_fieldsets

## **class** ReadOnlyAdmin(admin.ModelAdmin):

### **def** get\_readonly\_fields(self, request, obj=None):

**if** self.declared\_fieldsets:

**return** flatten\_fieldsets(self.declared\_fieldsets)

**else**:

**return** list(set(

[field.name **for** field **in** self.opts.local\_fields] +

[field.name **for** field **in** self.opts.local\_many\_to\_many]

))

## **class** ShapeInline(admin.TabularInline):

model = models.Shape

fields = ('link', 'shapetag', 'size')

readonly\_fields = ('link', 'shapetag', 'size')

extra = 0

### **def** link(self, instance):

url = reverse("admin:reporting\_shape\_change", args = (instance.id,))

**return** mark\_safe("<a href='**%s**'>**%s**</a>" % (url, instance.vs\_id))

## **class** AssetAdmin(ReadOnlyAdmin):

actions = None

inlines = ShapeInline,

fields = ('vs\_id', 'username', 'filename', 'created', 'deleted',

'raw\_data', 'last\_synced', 'last\_sync', 'id', 'storage\_size')

list\_display = ('vs\_id', 'username', 'last\_sync')

*#WARNING: Both of the below execute SQL for \*each row\* and are slow*

### **def** storage\_size(self, obj):

total\_bytes = obj.shape\_set.aggregate(Sum('size'))['size\_\_sum'] **or** 0

**return** '{:.2f}'.format(float(total\_bytes)/1000\*\*3)

storage\_size.short\_description = 'Storage Size (GB)'

### **def** all\_sites(self, obj):

**return** ", ".join([site.domain **for** site **in** obj.sites.all()])

## **class** ShapeAdmin(ReadOnlyAdmin):

fields = ('vs\_id', 'deleted', 'timestamp', 'last\_synced', 'last\_sync',

'asset', 'version', 'raw\_data', 'shapetag', 'size')

list\_display = ('vs\_id', 'shapetag', 'size', 'version', 'asset',

'last\_sync')

actions = None

## **class** AssetInline(admin.TabularInline):

model = models.Asset

fields = ('vs\_id', 'username', 'sites')

readonly\_fields = ('vs\_id', 'username', 'sites')

extra = 0

### **def** link(self, instance):

url = reverse("admin:reporting\_asset\_change", args = (instance.id,))

**return** mark\_safe("<a href='**%s**'>**%s**</a>" % (url, instance.vs\_id))

## **class** SyncRunAdmin(ReadOnlyAdmin):

list\_display = ('sync\_uuid', 'start\_time', 'end\_time', 'completed',

'remaining')

actions = None

*#inlines = AssetInline,*

### **def** remaining(self, obj):

**return** obj.asset\_set.count()

admin.site.register(models.Asset, AssetAdmin)

admin.site.register(models.Shape, ShapeAdmin)

admin.site.register(models.Download)

admin.site.register(models.SyncRun, SyncRunAdmin)

admin.site.register(models.Site)

*#admin.site.unregister(User)*

admin.site.unregister(Group)

*#admin.site.unregister(Site)*

# Warehouse.Reporting.models.py

**import** **os**

**import** **time**

**import** **logging**

**import** **requests**

**import** **jsonfield**

**import** **json**

**from** **retrying** **import** retry

**from** **django.db** **import** models

**from** **django.conf** **import** settings

**from** **django.core.exceptions** **import** ImproperlyConfigured

**from** **django.contrib.sites.models** **import** \_simple\_domain\_name\_validator

**from** **dateutil** **import** parser

log = logging.getLogger(\_\_name\_\_)

PER\_PAGE = 100

PAGE\_TOKEN = '\_\_page'

LIMIT\_TOKEN = '\_\_page\_size'

## **def** load\_json(raw):

*"""for debugging"""*

**try**:

**return** json.loads(raw)

**except** **ValueError**:

log.error('Unable to load JSON: **%r**' % raw)

**raise**

## **def** dump\_json(obj):

**try**:

**return** json.dumps(obj)

**except** **ValueError**:

log.error('Unable to dump JSON: **%r**' % obj)

**raise**

@retry(stop\_max\_attempt\_number=3, wait\_exponential\_multiplier=1000, wait\_exponential\_max=10000)

## **def** GET(url, \*\*kwargs):

result = requests.get(url, \*\*kwargs)

log.debug('HTTP Request performed to: {0} [status: {1}]'.format(url, result.status\_code))

**if** int(result.status\_code) == 500:

log.debug('warning status {0}'.format(result.status\_code))

**raise** **Exception**('Unable to retrieve data')

**return** result

## **class** Site(models.Model):

domain = models.CharField('domain name', max\_length=100, unique=True,

validators=[\_simple\_domain\_name\_validator])

*# Sales person?*

*# Account person?*

*# Company?*

*# Contract terms?*

### **def** \_\_unicode\_\_(self):

**return** self.domain

## **class** SyncRun(models.Model):

*"""Track each sync"""*

start\_time = models.DateTimeField(auto\_now=True)

end\_time = models.DateTimeField(blank=True, null=True)

sync\_uuid = models.CharField(max\_length=32)

completed = models.BooleanField(default=False)

site = models.ForeignKey('reporting.Site')

### **def** \_\_unicode\_\_(self):

**return** self.sync\_uuid

## **class** DamAssetManager(models.Manager):

### **def** get\_queryset(self):

**return** BorkAssetQuerySet(model=self.model, using=None)

## **class** ReportableModelMixin(models.Model):

last\_synced = models.DateTimeField(auto\_now=True)

last\_sync = models.ForeignKey('reporting.SyncRun')

### **class** Meta:

abstract = True

## **class** Asset(ReportableModelMixin):

*"""A local cache record of an asset in Vidispine"""*

objects = models.Manager()

vidispine\_objects = DamAssetManager()

deleted = models.DateTimeField(blank=True, null=True)

vs\_id = models.CharField(max\_length=10, unique=True)

filename = models.CharField(max\_length=255, blank=True, null=True)

username = models.CharField(max\_length=255)

created = models.DateTimeField()

sites = models.ManyToManyField('reporting.Site')

raw\_data = jsonfield.JSONField()

### **def** \_\_unicode\_\_(self):

**return** u'{} - {} ({})'.format(self.vs\_id, self.filename, self.username)

### **def** update\_size(self):

original\_shape\_qs = self.shape\_set.filter(shapetag='original')

**if** self.size == 0 **and** original\_shape\_qs **and** original\_shape\_qs[0].size:

self.size = original\_shape\_qs[0].size

self.save()

## **class** Download(ReportableModelMixin):

*"""A record of each shape download"""*

item = models.ForeignKey('reporting.Asset')

when = models.DateTimeField(auto\_now\_add=True)

username = models.CharField(max\_length=255)

shape = models.ForeignKey('reporting.Shape')

## **class** Shape(ReportableModelMixin):

*"""A record of a Vidispine shape"""*

deleted = models.DateTimeField(blank=True, null=True)

asset = models.ForeignKey('reporting.Asset')

vs\_id = models.CharField(max\_length=10, unique=True)

shapetag = models.CharField(max\_length=255)

timestamp = models.DateTimeField(blank=True, null=True)

size = models.BigIntegerField()

version = models.IntegerField()

raw\_data = jsonfield.JSONField()

### **def** \_\_unicode\_\_(self):

**return** u'{} - {} (version {})'.format(self.vs\_id, self.shapetag, self.version)

## **def** get\_asset(url):

*"""Retrieve full information for specific asset"""*

headers = {'content-type': 'application/json'}

headers.update(settings.BORK\_AUTH)

response = GET(url, headers=headers)

json\_response = load\_json(response.content)

**return** json\_response

## **def** get\_shapes\_for\_asset(asset\_id):

*"""Retrieve individual transcodes"""*

headers = {'content-type': 'application/json'}

headers.update(settings.BORK\_AUTH)

response = GET('{}item/{}/asset'.format(settings.BORK\_URL, asset\_id), headers=headers)

json\_response = load\_json(response.content)

**return** json\_response.get('assets')

## **def** get\_shape(url):

headers = {'content-type': 'application/json'}

headers.update(settings.BORK\_AUTH)

response = GET('{}'.format(url), headers=headers)

json\_response = load\_json(response.content)

**return** json\_response

## **def** perform\_search(runas, filters = None):

*"""Query Vidispine for assets"""*

log.debug('ZONZA API search request {0}'.format(filters))

headers = {'content-type': 'application/json'}

headers.update(settings.BORK\_AUTH)

response = GET('{}item'.format(settings.BORK\_URL), params=filters, headers=headers)

json\_response = load\_json(response.content)

**return** json\_response

## **def** get\_offsets(emitted, skip, precache):

*"""Compute vidispine search offsets from our iterator state"""*

start\_at = skip + emitted

asset = start\_at % precache

page = start\_at / precache + 1

**return** (asset, page)

## **def** asset\_iterator(zonza\_site, skip):

count = 0

consumed = 0

emitted = 0

existing = 0

msg = 'Synced {0}/{1} ({2} already existed)'

skip = skip **or** 0

sync\_message = **lambda** : msg.format(emitted, skip, count, existing)

per\_page = PER\_PAGE

assets\_skipped = skip/per\_page \* per\_page

filters = {

'zonza\_site': zonza\_site,

LIMIT\_TOKEN: per\_page,

PAGE\_TOKEN: 1

}

**while** True:

asset\_offset, page = get\_offsets(emitted, skip, per\_page)

filters[PAGE\_TOKEN] = page

**try**:

filters = filters

result = perform\_search(runas=None, filters=filters)

**except** **Exception** **as** exc:

error = 'Error when searching for DamAsset: {0}'.format(exc)

**raise**

**if** **not** result.get('item'):

**raise** **StopIteration**

delay = int(getattr(settings, 'SYNC\_CALL\_DELAY', 0))

time.sleep(delay)

count = int(result.get('hits')) - skip

**for** num, asset **in** enumerate(result.get('item')):

consumed += 1

**if** num >= asset\_offset:

asset\_id = asset['id']

vidi\_ids = [{'id': asset\_id}]

**try**:

emitted += 1

**yield** (asset, count) *# single-threaded*

*#yield asset*

**except** **Exception** **as** exc:

error = 'Error when creating DamAsset: {0}'.format(exc)

**raise**

hits = int(result.get('hits'))

**if** skip:

hits = hits - assets\_skipped

**if** consumed >= hits:

**raise** **StopIteration**

# Warehouse.Reporting.reports.py

**import** **csv**

**from** **datetime** **import** datetime, timedelta

**from** **tempfile** **import** NamedTemporaryFile

**from** **collections** **import** OrderedDict

**from** **django.core.exceptions** **import** ImproperlyConfigured

**from** **django.utils.translation** **import** ugettext\_lazy **as** \_

**from** **django.db.models** **import** F, Max

**from** **django.contrib.sites.models** **import** Site

**from** **reporting** **import** models, utils

## **class** CSVReport(object):

*"""An abstract csv serialisable report"""*

columns = {}

extra\_columns = {}

### **def** \_\_init\_\_(self, start=None, end=None, site=None):

self.file = NamedTemporaryFile(delete=False)

self.writer = csv.writer(self.file, dialect=csv.excel)

**if** **not** end:

end = datetime.now()

**if** **not** start:

start = end - timedelta(100)

self.start = start

self.end = end

self.site = site **and** Site.objects.get(name=site) **or** None

### **def** write\_headings(self):

self.writer.writerow(self.columns.keys() + self.extra\_columns.keys())

### **def** serialise(self):

*"""Write report data to `self.file` before returning it*

*:returns:*

*A temporary file handle containing CSV data*

*"""*

**raise** **NotImplementedError**

## **class** UsageReport(CSVReport):

*"""Report each shape for every item ingested in the period*

*Also computes a usage metric for the time each shape has been on*

*the system (within the period)*

*"""*

columns = OrderedDict([

(\_('Created'), 'item\_\_created'),

(\_('Item ID'), 'item\_\_vs\_id'),

(\_('Shape ID'), 'vs\_id'),

(\_('Shape Tag'), 'shapetag'),

(\_('Shape Size'), 'size'),

(\_('Version'), 'version'),

(\_('Filename'), 'item\_\_filename'),

(\_('Username'), 'item\_\_username'),

(\_('Sites'), 'item\_\_sites\_\_name'),

(\_('Deleted On'), 'deleted'),

])

extra\_columns = OrderedDict([

(\_('Usage (GB days, to the nearest KB)'), utils.get\_usage),

])

### **def** queryset(self):

shapes = models.Shape.objects.filter(item\_\_created\_\_range=(self.start, self.end))

**if** self.site:

**return** shapes.filter(item\_\_sites=self.site)

**return** shapes

### **def** serialise(self):

self.write\_headings()

**for** row **in** self.queryset().values(\*self.columns.values()):

orig\_row = row.copy()

row['size'] = utils.filesizeformat(row['size'])

new\_row = [row[c] **for** c **in** self.columns.values()]

*# Add extra computed columns*

**for** key, func **in** self.extra\_columns.items():

new\_row.append(func(orig\_row, self.start, self.end))

self.writer.writerow(new\_row)

**print** self.file.name

self.file.flush()

self.file.close()

**return** self.file

## **class** IngestShapeReport(CSVReport):

*"""Report all shapes ingested*

*NOTE: This will miss ingests that have no shapes*

*"""*

columns = OrderedDict([

(\_('Ingested'), 'item\_\_created'),

(\_('User'), 'item\_\_username'),

(\_('Item ID'), 'item\_\_vs\_id'),

(\_('Shape ID'), 'vs\_id'),

(\_('Version'), 'version'),

(\_('Sites'), 'item\_\_sites\_\_name'),

(\_('Filename'), 'item\_\_filename'),

(\_('Size'), 'item\_\_size'),

])

### **def** queryset(self):

items = models.Shape.objects.filter(item\_\_created\_\_range=(self.start, self.end))

**if** self.site:

**return** items.filter(item\_\_sites=self.site)

**return** items

### **def** serialise(self):

self.write\_headings()

**for** row **in** self.queryset().values(\*self.columns.values()):

new\_row = [row[c] **for** c **in** self.columns.values()]

self.writer.writerow(new\_row)

**print** self.file.name

self.file.flush()

self.file.close()

**return** self.file

## **class** IngestReport(CSVReport):

*"""Report all items ingested, excluding versions*

*"""*

columns = OrderedDict([

(\_('Ingested'), 'created'),

(\_('User'), 'username'),

(\_('Item ID'), 'vs\_id'),

(\_('Versions'), 'versions'),

(\_('Sites'), 'sites\_\_name'),

(\_('Filename'), 'filename'),

(\_('Size'), 'size'),

])

### **def** queryset(self):

items = models.Asset.objects.filter(created\_\_range=(self.start, self.end))

items = items.order\_by('created')

items = items.annotate(versions=Max('shape\_\_version'))

**if** self.site:

**return** items.filter(sites=self.site)

**return** items

### **def** serialise(self):

self.write\_headings()

**for** row **in** self.queryset().values(\*self.columns.values()):

row['size'] = utils.filesizeformat(row['size'])

new\_row = [row[c] **for** c **in** self.columns.values()]

self.writer.writerow(new\_row)

**print** self.file.name

self.file.flush()

self.file.close()

**return** self.file

## **class** DownloadReport(CSVReport):

columns = OrderedDict([

(\_('When'), 'when'),

(\_('User'), 'username'),

(\_('Item ID'), 'item\_\_vs\_id'),

(\_('Shape ID'), 'shape\_\_vs\_id'),

(\_('Shape Tag'), 'shape\_\_shapetag'),

(\_('Version'), 'shape\_\_version'),

(\_('Sites'), 'item\_\_sites\_\_name'),

(\_('Size'), 'shape\_\_size'),

(\_('Filename'), 'item\_\_filename'),

])

### **def** queryset(self):

downloads = models.Download.objects.filter(when\_\_range=(self.start, self.end))

**if** self.site:

**return** downloads.filter(item\_\_sites=self.site)

**return** downloads

### **def** serialise(self):

self.write\_headings()

**for** row **in** self.queryset().values(\*self.columns.values()):

row['shape\_\_size'] = utils.filesizeformat(row['shape\_\_size'])

new\_row = [row[c] **for** c **in** self.columns.values()]

self.writer.writerow(new\_row)

**print** self.file.name

self.file.flush()

self.file.close()

**return** self.file

# Warehouse.Reporting.tasks.py

**import** **logging**

*#from celery.task import PeriodicTask*

*#from celery.schedules import crontab*

*#from celery.registry import tasks*

**from** **django.utils** **import** timezone

**from** **reporting.models** **import** Asset

log = logging.getLogger(\_\_name\_\_)

PERCENTILES = 100

*#class SyncReportDataTask(PeriodicTask):*

*#"""Called asynchronously by CeleryBeat Scheduler"""*

*#name = "reporting.tasks.SyncReportDataTask"*

*#routing\_key = "celery"*

*#run\_every = crontab(hour="4")*

*#def run(self, hours=24, \*\*kwargs):*

*#log.debug('SyncReportDataTask: Started at {0}'.format(*

*#timezone.now().isoformat()))*

*#objects = Asset.vidispine\_objects.all()*

*#count = objects.count()*

*#done = 0*

*#for item in objects:*

*#done += 1*

*#self.update\_progress(done, count)*

*#def update\_progress(self, done, count):*

*#percentiles = [int(count \* percentile / PERCENTILES)*

*#for percentile in range(0, PERCENTILES + 1)]*

*#if done in percentiles:*

*#log.debug('SyncReportDataTask: {0}% ({1}/{2})'.format(*

*#int(PERCENTILES \* done / count), done, count))*

*#tasks.register(SyncReportDataTask*

# Warehouse.Reporting.urls.py

**from** **django.conf.urls** **import** patterns, include, url

**from** **django.contrib** **import** admin

**from** **reporting** **import** views

admin.autodiscover()

urlpatterns = patterns('',

url(r'^$', views.dashboard, name='dashboard'),

url(r'^download/$', views.download\_csv, name='download\_csv'),

url(r'^domain/(?P<domain>.+)/', views.domain, name='domain'),

)

# Warehouse.Reporting.utils.py

## **def** filesizeformat(num):

**for** size **in** ['bytes', 'KB', 'MB', 'GB']:

**if** num < 1024.0:

**return** "**%3.2f%s**" % (num, size)

num /= 1024.0

**return** "**%3.2f%s**" % (num, 'TB')

## **def** bytes\_to\_gb(orig\_bytes):

**return** "**%3.6f**" % (orig\_bytes / float(1024 \* 1024 \* 1024))

## **def** get\_days\_elapsed(created, deleted, start, end):

**if** start > end:

**raise** ImproperlyConfigured("Start date must be becore the end date")

actual\_start = start

actual\_end = end

*# Camp reduce period if created or deleted within*

**if** created > start:

actual\_start = created

**if** deleted **and** deleted < end:

actual\_end = end

**return** (actual\_end - actual\_start).days

## **def** get\_usage(shape\_row, start, end):

days = get\_days\_elapsed(shape\_row['item\_\_created'],

shape\_row['deleted'],

start, end)

**return** bytes\_to\_gb(shape\_row['size'] \* days)

# Warehouse.Reporting.views.py

**from** **cStringIO** **import** StringIO

**import** **json**

**from** **django.http** **import** HttpResponse

**from** **django.contrib.admin.views.decorators** **import** staff\_member\_required

**from** **django.contrib** **import** messages

**from** **django.shortcuts** **import** render, redirect

**from** **django.template.context** **import** RequestContext

**from** **django.db** **import** connection

**from** **django.db.models** **import** Count, Sum, Max, F

**from** **django.contrib** **import** admin

**from** **django.core.urlresolvers** **import** reverse

**from** **reporting.models** **import** Asset, SyncRun, Site

## **def** domain(request, domain):

sync\_runs = SyncRun.objects.filter(site\_\_domain\_\_contains=domain).order\_by('-start\_time')

**try**:

last\_sync = sync\_runs.filter(completed=True)[0]

**except** **IndexError**:

last\_sync = None

all\_sites = Site.objects.filter(domain=domain)

size\_by\_site = all\_sites.values('domain') \

.annotate(

transcodes=Count('asset\_\_shape'),

size=Sum('asset\_\_shape\_\_size'),

count=Count('asset', distinct=True),

uploaders=Count('asset\_\_username', distinct=True)) \

.order\_by('-size')

top\_uploaders = all\_sites.values('domain', 'asset\_\_username') \

.annotate(count=Count('asset')).order\_by('-count')[:20]

params = {

'domain': domain,

'site\_header': admin.site.site\_header + " for " + domain,

'last\_sync': last\_sync,

'last\_syncs': sync\_runs[:5],

'size\_by\_site': size\_by\_site,

'top\_uploaders': top\_uploaders,

}

**return** render(request, 'reporting/domain.html', params,

context\_instance=RequestContext(request))

## **def** download\_csv(domain):

csv\_file = StringIO()

*# Postgres generates CSV for us, we just serve it directly to the user*

raw\_sql = """COPY

(SELECT

id,

raw\_data::json->'metadata'->>'zonza\_site' as "Sites",

raw\_data::json->'metadata'->>'trials\_category' as "Category"

FROM reporting\_asset)

TO STDOUT CSV HEADER;"""

cursor = connection.cursor()

cursor.copy\_expert(raw\_sql, csv\_file)

filename = 'zonza-asset-report.csv'

response = HttpResponse(csv\_file.getvalue(), content\_type='text/csv')

response['Content-Disposition'] = 'attachment; filename={}'.format(filename)

**return** response

*#@staff\_member\_required*

## **def** dashboard(request):

*"""Give a snapshot of the status of the reporting app"""*

sync\_runs = SyncRun.objects.all().order\_by('-start\_time')

**try**:

last\_sync = sync\_runs.filter(completed=True)[0]

**except** **IndexError**:

last\_sync = None

all\_sites = Site.objects.all()

*#num\_assets\_by\_site = all\_sites.values('domain') \*

*#.annotate(count=Count('asset')).order\_by('-count')*

*#num\_uploaders\_by\_site = all\_sites.values('domain') \*

*#.annotate(uploaders=Count('asset\_\_username', distinct=True))*

*# TODO: add days since last since per site*

size\_by\_site = all\_sites.values('domain') \

.annotate(

transcodes=Count('asset\_\_shape'),

size=Sum('asset\_\_shape\_\_size'),

count=Count('asset', distinct=True),

uploaders=Count('asset\_\_username', distinct=True)) \

.order\_by('-count')[:10]

sizes = all\_sites.aggregate(

count=Count('asset', distinct=True),

transcodes=Count('asset\_\_shape'),

size=Sum('asset\_\_shape\_\_size'),

uploaders=Count('asset\_\_username', distinct=True))

top\_uploaders = all\_sites.values('domain', 'asset\_\_username') \

.annotate(count=Count('asset')).order\_by('-count')[:10]

x = [x **for** x **in** all\_sites.values('domain')

.annotate(size=Count('asset', distinct=True))

.annotate(total=Sum('asset\_\_shape\_\_size'))

.order\_by('-size') *# TODO: Figure out why sorting by total is buggy*

.values\_list('domain', 'size')]

graph\_assets\_data = x[:5]

graph\_assets\_data = [[x,y,reverse('reporting.views.domain', args=(x,))]

**for** x, y **in** graph\_assets\_data]

*# [["teamhills.zonza.tv", null], ["deluxe.zonza.tv", null], ["trials.zonza.tv", 243497942962], ["grey.zonza.tv", 135888998176], ["230pas.zonza.tv", 48393166596], ["zonzacompany.zonza.tv", 5456789214], ["trg-deluxe.zonza.tv", 2953964488], ["gmi-deluxe.zonza.tv", 2182796439]]'*

params = {

'site\_header': admin.site.site\_header,

'last\_sync': last\_sync,

'last\_syncs': sync\_runs[:5],

'sizes': sizes,

'sizes\_json': json.dumps(sizes),

'size\_by\_site': size\_by\_site,

'top\_uploaders': top\_uploaders,

'graph\_assets\_data': json.dumps(graph\_assets\_data)

}

**return** render(request, 'reporting/dashboard.html', params,

context\_instance=RequestContext(request))