Django Sorcery Documentation

Release 0.12.0

Serkan Hosca

Aug 06, 2022

Contents

GitHub: https://github.com/shosca/django-sorcery

SQLAlchemy is an excellent orm. And Django is a great framework, until you decide not to use Django ORM. This library provides utilities, helpers and configurations to ease the pain of using SQLAlchemy with Django. It aims to provide a similar development experience to building a Django application with Django ORM, except with SQLAlchemy.

CHAPTER 1

Installation

```
pip install django-sorcery
```

CHAPTER 2

Quick Start

Lets start by creating a site:

```
$ django-admin startproject mysite
```

And lets create an app:

```
$ cd mysite
$ python manage.py startapp polls
```

This will create a polls app with standard django app layout:

```
$ tree
.
├── manage.py
├── polls
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
└── mysite
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py

3 directories, 12 files
```

And lets add our `polls` app and `django_sorcery` in `INSTALLED_APPS` in `mysite/settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_sorcery',
    'polls.apps.PollsConfig',
]
```

Now we're going to make a twist and start building our app with `sqlalchemy`. Lets define our models in `polls/models.py`:

```
from django_sorcery.db import databases


db = databases.get("default")


class Question(db.Model):
    pk = db.Column(db.Integer(), autoincrement=True, primary_key=True)
    question_text = db.Column(db.String(length=200))
    pub_date = db.Column(db.DateTime())


class Choice(db.Model):
    pk = db.Column(db.Integer(), autoincrement=True, primary_key=True)
    choice_text = db.Column(db.String(length=200))
    votes = db.Column(db.Integer(), default=0)

    question = db.ManyToOne(Question, backref=db.backref("choices", cascade="all,␣
↪delete-orphan"))
```

Now that we have some models, lets create a migration using `alembic` integration:

```
$ python manage.py sorcery revision -m "Add question and poll models" polls
  Generating ./polls/migrations/3983fc419e10_add_question_and_poll_models.py ... done
```

Let's take a look at the generated migration file `./polls/migrations/3983fc419e10_add_question_and_poll_models.py`:

```
"""
Add question and poll models

Revision ID: 3983fc419e10
Revises:
Create Date: 2019-04-16 20:57:48.154179
"""

from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision = '3983fc419e10'
down_revision = None
```

(continues on next page)

(continued from previous page)

```
branch_labels = None
depends_on = None


def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('question',
    sa.Column('pk', sa.Integer(), autoincrement=True, nullable=False),
    sa.Column('question_text', sa.String(length=200), nullable=True),
    sa.Column('pub_date', sa.DateTime(), nullable=True),
    sa.PrimaryKeyConstraint('pk')
    )
    op.create_table('choice',
    sa.Column('pk', sa.Integer(), autoincrement=True, nullable=False),
    sa.Column('choice_text', sa.String(length=200), nullable=True),
    sa.Column('votes', sa.Integer(), nullable=True),
    sa.Column('question_pk', sa.Integer(), nullable=True),
    sa.ForeignKeyConstraint(['question_pk'], ['question.pk'], ),
    sa.PrimaryKeyConstraint('pk')
    )
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_table('choice')
    op.drop_table('question')
    # ### end Alembic commands ###
```

Let's take a look at generated sql:

```
$ python manage.py sorcery upgrade --sql polls

CREATE TABLE alembic_version_polls (
    version_num VARCHAR(32) NOT NULL,
    CONSTRAINT alembic_version_polls_pkc PRIMARY KEY (version_num)
);

-- Running upgrade  -> d7d86e07cc8e

CREATE TABLE question (
    pk INTEGER NOT NULL,
    question_text VARCHAR(200),
    pub_date DATETIME,
    PRIMARY KEY (pk)
);

CREATE TABLE choice (
    pk INTEGER NOT NULL,
    choice_text VARCHAR(200),
    votes INTEGER,
    question_pk INTEGER,
    PRIMARY KEY (pk),
    FOREIGN KEY(question_pk) REFERENCES question (pk)
);

INSERT INTO alembic_version_polls (version_num) VALUES ('d7d86e07cc8e');
```

Let's bring our db up to date:

```
$ python manage.py sorcery upgrade
Running migrations for polls on database default
```

Right now, we have enough to hop in django shell:

```
$ python manage.py shell

>>> from polls.models import Choice, Question, db  # Import the model classes and the db

# we have no choices or questions in db yet
>>> Choice.query.all()
[]
>>> Question.query.all()
[]

# Lets create a new question
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())
>>> q
Question(pk=None, pub_date=datetime.datetime(2018, 5, 19, 0, 54, 20, 778186, tzinfo=<UTC>), question_text="What's new?")

# lets save our question, we need to add our question to the db
>>> db.add(q)

# at this point the question is in pending state
>>> db.new
IdentitySet([Question(pk=None, pub_date=datetime.datetime(2018, 5, 19, 0, 54, 20, 778186, tzinfo=<UTC>), question_text="What's new?")])

# lets flush to the database
>>> db.flush()

# at this point our question is in persistent state and will receive a primary key
>>> q.pk
1

# lets change the question text
>>> q.question_text = "What's up?"
>>> db.flush()

# Question.objects and Question.query are both query properties that return a query object bound to db
>>> Question.objects
<django_sorcery.db.query.Query at 0x7feb1c7899e8>
>>> Question.query
<django_sorcery.db.query.Query at 0x7feb1c9377f0>

# and lets see all the questions
>>> Question.objects.all()
[Question(pk=1, pub_date=datetime.datetime(2018, 5, 19, 0, 54, 20, 778186, tzinfo=<UTC>), question_text="What's up?")]

>>> exit()
```

Let's add a couple of views in `polls/views.py`, starting with a list view:

```
from django.shortcuts import render
from django.template import loader
from django.http import HttpResponseRedirect
from django.urls import reverse

from django_sorcery.shortcuts import get_object_or_404

from .models import Question, Choice, db

def index(request):
   latest_question_list = Question.objects.order_by(Question.pub_date.desc())[:5]
   context = {'latest_question_list': latest_question_list}
   return render(request, 'polls/index.html', context)


def detail(request, question_id):
   question = get_object_or_404(Question, pk=question_id)
   return render(request, 'polls/detail.html', {'question': question})


def results(request, question_id):
   question = get_object_or_404(Question, pk=question_id)
   return render(request, 'polls/results.html', {'question': question})


def vote(request, question_id):
   question = get_object_or_404(Question, pk=question_id)

   selected_choice = Choice.query.filter(
      Choice.question == question,
      Choice.pk == request.POST['choice'],
   ).one_or_none()

   if not selected_choice:
      return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
      })

   selected_choice.votes += 1
   db.flush()
   return HttpResponseRedirect(reverse('polls:results', args=(question.pk,)))
```

and register the view in `polls/urls.py`:

```
from django.urls import path

from . import views


app_name = 'polls'
urlpatterns = [
   path('', views.index, name='index'),
   path('<int:question_id>/', views.detail, name='detail'),
   path('<int:question_id>/results', views.results, name='results'),
   path('<int:question_id>/vote', views.vote, name='vote'),
]
```

and register the `SQLAlchemyMiddleware` to provide unit-of-work per request pattern:

```
MIDDLEWARE = [
    'django_sorcery.db.middleware.SQLAlchemyMiddleware',
    # ...
]
```

and add some templates:

`polls/templates/polls/index.html`:

```
{% if latest_question_list %}
<ul>
{% for question in latest_question_list %}
<li><a href="{% url 'polls:detail' question.pk %}">{{ question.question_text }}</a></
↪li>
{% endfor %}
</ul>
{% else %}
<p>No polls are available.</p>
{% endif %}
```

`polls/templates/polls/detail.html`:

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.pk %}" method="post">
{% csrf_token %}
{% for choice in question.choices %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{␣
↪choice.pk }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

`polls/templates/polls/results.html`:

```
<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choices %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}
↪</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' question.pk %}">Vote again?</a>
```

This is all fine but we can do one better using generic views. Lets adjust our views in `polls/views.py`:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from django.urls import reverse

from django_sorcery.shortcuts import get_object_or_404
```

(continues on next page)

(continued from previous page)

```
from django_sorcery import views

from .models import Question, Choice, db


class IndexView(views.ListView):
   template_name = 'polls/index.html'
   context_object_name = 'latest_question_list'

   def get_queryset(self):
      return Question.objects.order_by(Question.pub_date.desc())[:5]


class DetailView(views.DetailView):
   model = Question
   session = db
   template_name = 'polls/detail.html'


class ResultsView(DetailView):
   template_name = 'polls/results.html'


def vote(request, question_id):
   question = get_object_or_404(Question, pk=question_id)

   selected_choice = Choice.query.filter(
      Choice.question == question,
      Choice.pk == request.POST['choice'],
   ).one_or_none()

   if not selected_choice:
      return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
      })

   selected_choice.votes += 1
   db.flush()
   return HttpResponseRedirect(reverse('polls:results', args=(question.pk,)))
```

and adjust the `polls/urls.py` like:

```
from django.urls import path

from . import views


app_name = 'polls'
urlpatterns = [
   path('', views.IndexView.as_view(), name='index'),
   path('<int:pk>/', views.DetailView.as_view(), name='detail'),
   path('<int:pk>/results', views.ResultsView.as_view(), name='results'),
   path('<int:question_id>/vote', views.vote, name='vote'),
]
```

The default values for `template_name` and `context_object_name` are similar to django's generic views. If

we handn't defined those the default for template names would've been `polls/question_detail.html` and `polls/question_list.html` for the detail and list template names, and `question` and `question_list` for context names for detail and list views.

This is all fine but we can even do one better using a viewset. Lets adjust our views in `polls/views.py`:

```
from django.http import HttpResponseRedirect
from django.urls import reverse, reverse_lazy

from django_sorcery.routers import action
from django_sorcery.viewsets import ModelViewSet

from .models import Question, Choice, db


class PollsViewSet(ModelViewSet):
    model = Question
    fields = "__all__"
    destroy_success_url = reverse_lazy("polls:question-list")

    def get_success_url(self):
        return reverse("polls:question-detail", kwargs={"pk": self.object.pk})

    @action(detail=True)
    def results(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    @action(detail=True, methods=["POST"])
    def vote(self, request, *args, **kwargs):
        self.object = self.get_object()

        selected_choice = Choice.query.filter(
            Choice.question == self.object, Choice.pk == request.POST.get("choice")
        ).one_or_none()

        if not selected_choice:
            context = self.get_detail_context_data(object=self.object)
            context["error_message"] = "You didn't select a choice."
            self.action = "retrieve"
            return self.render_to_response(context)

        selected_choice.votes += 1
        db.flush()
        return HttpResponseRedirect(reverse("polls:question-results", args=(self.object.
↪pk,)))
```

And adjusting our `polls/urls.py` like:

```
from django.urls import path, include

from django_sorcery.routers import SimpleRouter

from . import views

router = SimpleRouter()
router.register("", views.PollsViewSet)

app_name = "polls"
urlpatterns = [path("", include(router.urls))]
```

With these changes we'll have the following urls:

```
$ ./manage.py run show_urls
/polls/      polls.views.PollsViewSet        polls:question-list
/polls/<pk>/ polls.views.PollsViewSet        polls:question-detail
/polls/<pk>/delete/  polls.views.PollsViewSet        polls:question-destroy
/polls/<pk>/edit/    polls.views.PollsViewSet        polls:question-edit
/polls/<pk>/results/ polls.views.PollsViewSet        polls:question-results
/polls/<pk>/vote/    polls.views.PollsViewSet        polls:question-vote
/polls/new/  polls.views.PollsViewSet        polls:question-new
```

This will map the following operations to following actions on the viewset:

Method	Path	Action	Route Name
GET	/polls/	list	question-list
POST	/polls/	create	question-list
GET	/polls/new/	new	question-new
GET	/polls/1/	retrieve	question-detail
POST	/polls/1/	update	question-detail
PUT	/polls/1/	update	question-detail
PATCH	/polls/1/	update	question-detail
DELETE	/polls/1/	destroy	question-detail
GET	/polls/1/edit/	edit	question-edit
GET	/polls/1/delete/	confirm_destoy	question-delete
POST	/polls/1/delete/	destroy	question-delete

Now, lets add an inline formset to be able to add choices to questions, adjust `polls/views.py`:

```
from django.http import HttpResponseRedirect
from django.urls import reverse, reverse_lazy

from django_sorcery.routers import action
from django_sorcery.viewsets import ModelViewSet
from django_sorcery.formsets import inlineformset_factory

from .models import Question, Choice, db


ChoiceFormSet = inlineformset_factory(relation=Question.choices, fields=(Choice.
↪choice_text.key,), session=db)


class PollsViewSet(ModelViewSet):
    model = Question
    fields = (Question.question_text.key, Question.pub_date.key)
    destroy_success_url = reverse_lazy("polls:question-list")

    def get_success_url(self):
        return reverse("polls:question-detail", kwargs={"pk": self.object.pk})

    def get_form_context_data(self, **kwargs):
        kwargs["choice_formset"] = self.get_choice_formset()
        return super().get_form_context_data(**kwargs)

    def get_choice_formset(self, instance=None):
        if not hasattr(self, "_choice_formset"):
```

(continues on next page)

(continued from previous page)

```
            instance = instance or self.object
            self._choice_formset = ChoiceFormSet(
               instance=instance, data=self.request.POST if self.request.POST else␣
↪None
            )

      return self._choice_formset

   def process_form(self, form):
      if form.is_valid() and self.get_choice_formset(instance=form.instance).is_
↪valid():
            return self.form_valid(form)

      return form.invalid(self, form)

   def form_valid(self, form):
      self.object = form.save()
      self.object.choices = self.get_choice_formset().save()
      db.flush()
      return HttpResponseRedirect(self.get_success_url())

   @action(detail=True)
   def results(self, request, *args, **kwargs):
      return self.retrieve(request, *args, **kwargs)

   @action(detail=True, methods=["POST"])
   def vote(self, request, *args, **kwargs):
      self.object = self.get_object()

      selected_choice = Choice.query.filter(
            Choice.question == self.object, Choice.pk == request.POST.get("choice")
      ).one_or_none()

      if not selected_choice:
            context = self.get_detail_context_data(object=self.object)
            context["error_message"] = "You didn't select a choice."
            self.action = "retrieve"
            return self.render_to_response(context)

      selected_choice.votes += 1
      db.flush()
      return HttpResponseRedirect(reverse("polls:question-results", args=(self.object.
↪pk,)))
```

And add `choice_formset` in the `polls/templates/question_edit.html` and `polls/templates/question_edit.html`

```
<form ... >
   ...
   {{ choice_formset }}
   ...
</form >
```

2.1 Changelog

2.1.1 0.12.0 (2022-03-25)

- Django 4.0 support (#170) [Diego Guerrero, Serkan Hosca]
- [pre-commit.ci] pre-commit autoupdate (#166) [pre-commit-ci[bot], pre- commit-ci[bot]]
- [pre-commit.ci] pre-commit autoupdate (#165) [pre-commit-ci[bot], pre- commit-ci[bot]]
- [pre-commit.ci] pre-commit autoupdate (#164) [pre-commit-ci[bot], pre- commit-ci[bot]]
- [pre-commit.ci] pre-commit autoupdate (#163) [pre-commit-ci[bot], pre- commit-ci[bot]]

2.1.2 0.11.3 (2021-11-30)

- [pre-commit.ci] pre-commit autoupdate (#162) [Serkan Hosca, pre- commit-ci[bot], pre-commit-ci[bot]]

2.1.3 0.11.2 (2021-04-08)

- Django 3.2 support (#159) [Serkan Hosca]

2.1.4 0.11.1 (2021-03-20)

- Attempt django like order by only when all criterion are strings (#158) [Serkan Hosca]

2.1.5 0.11.0 (2021-03-20)

- Add sqlalchemy 1.4 support (#157) [Serkan Hosca]
- Github actions build badge (#155) [Serkan Hosca]
- Adding github actions (#154) [Serkan Hosca]
- Fix build link. [Serkan Hosca]
- Add dj3.1 to matrix (#153) [Serkan Hosca]
- Pre-commit imporanize pyupgrade and docformat (#152) [Serkan Hosca]
- Add django3 to build matrix (#151) [Serkan Hosca]

2.1.6 0.10.2 (2019-11-07)

- Adding check-manifest in pre-commit (#150) [Miroslav Shubernetskiy]

2.1.7 0.10.1 (2019-08-31)

- Drop py2 from setup.py. [Serkan Hosca]

2.1.8 0.10.0 (2019-08-31)

- Drop py2 support (#149) [Serkan Hosca]
- Make model meta look like django meta (#148) [Serkan Hosca]
- Simplify column meta info (#147) [Serkan Hosca]
- Fix ModelChoiceField.get_object and make ModelForm inherit BaseModelForm (#146) [Serkan Hosca]

2.1.9 0.9.4 (2019-07-14)

- Fix middleware response return (#143) [Serkan Hosca]
- Use python/black (#141) [Serkan Hosca]

2.1.10 0.9.3 (2019-06-27)

- Defining test matrix in tox and switchint to tox-travis (#140) [Miroslav Shubernetskiy]
- Increase build matrix (#139) [Serkan Hosca]
- Update pre-commit (#138) [Serkan Hosca]

2.1.11 0.9.2 (2019-05-10)

- Drop trailing zeros on float to decimal conversion (#137) [Serkan Hosca]

2.1.12 0.9.1 (2019-04-28)

- Track committed models (#136) [Serkan Hosca]
- Fix topic. [Serkan Hosca]

2.1.13 0.9.0 (2019-04-23)

- Trigger configure_mappers before commands (#135) [Serkan Hosca]
- Add migrations on tutorial (#134) [Serkan Hosca]
- Drop sqlalchemy init django dependency (#133) [Serkan Hosca]
- Drop deprecated functions (#132) [Serkan Hosca]
- Raising in middleware on error (#131) [Miroslav Shubernetskiy]
- Allow limit_choices_to to be callable (#130) [Serkan Hosca]
- Add a bunch of docs and fix lint issues (#129) [Serkan Hosca]

2.1.14 0.8.12 (2019-03-15)

- Fix field run_validator (#128) [Serkan Hosca]

2.1.15 0.8.11 (2019-02-15)

- Fix db operations import from django for db ranges (#127) [Serkan Hosca]
- Adding black badge (#126) [Miroslav Shubernetskiy]

2.1.16 0.8.10 (2019-02-06)

- Fixing test_site to allow to create test migrations (#125) [Miroslav Shubernetskiy]

2.1.17 0.8.9 (2019-02-05)

- Adding default maxlength/min/maxvalue validators to CharField/IntField (#124) [Miroslav Shubernetskiy]
- Include migration mako script (#123) [Serkan Hosca]

2.1.18 0.8.8 (2019-01-29)

- Raise validation error with field name on coersion (#121) [Serkan Hosca]
- Add docs for testing. [Serkan Hosca]

2.1.19 0.8.7 (2019-01-26)

- Add autogenerate foreign key indexes (#118) [Serkan Hosca]
- Adding Transact testing utility with transact pytest fixture (#119) [Miroslav Shubernetskiy]

2.1.20 0.8.6 (2019-01-11)

- Added signals for create_all and drop_all (#117) [Miroslav Shubernetskiy]

2.1.21 0.8.5 (2019-01-10)

- Fixing composite field validation (#116) [Miroslav Shubernetskiy]

2.1.22 0.8.4 (2019-01-09)

- Adding OneToOne relationship shortcut (#115) [Miroslav Shubernetskiy]

2.1.23 0.8.3 (2019-01-08)

- Validate only pre-loaded models (#114) [Miroslav Shubernetskiy]

2.1.24 0.8.2 (2019-01-04)

- Fix decimal cleaning with thousand separator (#113) [Serkan Hosca]

2.1.25 0.8.1 (2019-01-04)

- Split choice from enum column info (#111) [Serkan Hosca]
- Regenerate docs. [Serkan Hosca]

2.1.26 0.8.0 (2019-01-02)

- Refactor coercers (#110) [Serkan Hosca]
- Added django-like filtering support to Query (#108) [Miroslav Shubernetskiy]

2.1.27 0.7.7 (2018-12-11)

- Make statement recording optional (#107) [Serkan Hosca]

2.1.28 0.7.6 (2018-12-10)

- Add query recording to profiler (#106) [Serkan Hosca]

2.1.29 0.7.5 (2018-12-04)

- Fix field column naming (#105) [Serkan Hosca]
- Parallel resetdb (#104) [Serkan Hosca]
- Refactor full_clean validation (#103) [Serkan Hosca]

2.1.30 0.7.4 (2018-11-29)

- Add validation runner and refactor validation (#102) [Serkan Hosca]

2.1.31 0.7.3 (2018-11-28)

- Fix event deque mutation (#101) [Serkan Hosca]

2.1.32 0.7.2 (2018-11-25)

- Add more tests (#100) [Serkan Hosca]

2.1.33 0.7.1 (2018-11-24)

- Fix boolean field constraint name (#99) [Serkan Hosca]
- Meta docs and more meta usage (#98) [Serkan Hosca]
- Nicer meta reprs (#97) [Serkan Hosca]

2.1.34 0.7.0 (2018-11-23)

- Refactor formfield mapping (#95) [Serkan Hosca]

2.1.35 0.6.18 (2018-11-20)

- Added full_clean(recursive=True) for adhoc full tree validation (#96) [Miroslav Shubernetskiy]

2.1.36 0.6.17 (2018-11-19)

- Implement formfield support in fields (#93) [Serkan Hosca]
- Remove yapf config. [Serkan Hosca]

2.1.37 0.6.16 (2018-11-16)

- Fix docs build. [Serkan Hosca]
- Add TimestampField (#74) [Serkan Hosca]

2.1.38 0.6.15 (2018-11-14)

- Fix edge case with enum field (#69) [Serkan Hosca]

2.1.39 0.6.14 (2018-11-14)

- Refactor autocoercers to allow coerce individual attrs (#68) [Serkan Hosca]
- Bump pre-commit check versions (#67) [Serkan Hosca]
- Caching pip and pre-commit. [Miroslav Shubernetskiy]
- Tiny fixup (#65) [Anthony Sottile]

2.1.40 0.6.13 (2018-11-08)

- Fixing DecimalField not honoring max_digits and decimal_places (#64) [Miroslav Shubernetskiy]

2.1.41 0.6.12 (2018-11-07)

- Allowing to set if field is required separately from nullable (#63) [Miroslav Shubernetskiy]
- Fix coercer issues (#62) [Serkan Hosca]

2.1.42 0.6.11 (2018-11-05)

- Implement autocoerce using form fields (#61) [Serkan Hosca]
- Update lock. [Serkan Hosca]
- Adding more validators (#60) [Miroslav Shubernetskiy]

2.1.43 0.6.10 (2018-10-31)

- List primary keys directly (#59) [Serkan Hosca]
- Passing model-defined validators to field_kwargs (#58) [Miroslav Shubernetskiy]
- Ignoring schema names in alembic version table for sqlite (#57) [Miroslav Shubernetskiy]

2.1.44 0.6.9 (2018-10-17)

- Not running field validations when column has default value (#56) [Miroslav Shubernetskiy]

2.1.45 0.6.8 (2018-10-16)

- Rename OPTIONS to ALCHEMY_OPTIONS (#55) [Serkan Hosca]
- Relock (#54) [Serkan Hosca]

2.1.46 0.6.7 (2018-10-03)

- Allowing to customize whether to log or add headers in profiler (#53) [Miroslav Shubernetskiy]

2.1.47 0.6.6 (2018-09-27)

- Merge pull request #51 from shosca/fields. [Serkan Hosca]
- Django-like fields. [Serkan Hosca]

2.1.48 0.6.5 (2018-09-21)

- Merge pull request #52 from shosca/engine_options. [Serkan Hosca]
- Support for more engine options in url. [Miroslav Shubernetskiy]

2.1.49 0.6.4 (2018-09-18)

- Merge pull request #49 from shosca/deserialize. [Serkan Hosca]
- Added tests for relation_info. [Miroslav Shubernetskiy]
- Using local_remote_pairs_for_identity_key to backfill models relations in deserialize. [Miroslav Shubernetskiy]
- Try backpopulate by fk's on deserialize. [Serkan Hosca]
- Deserialize model instance. [Serkan Hosca]
- Merge pull request #50 from shosca/refactor-fieldmapper. [Serkan Hosca]
- Refactor field mapping. [Serkan Hosca]

2.1.50 0.6.3 (2018-09-04)

- Merge pull request #48 from shosca/url. [Serkan Hosca]
- Only popping custom engine parameters from url. [Miroslav Shubernetskiy]

2.1.51 0.6.2 (2018-08-31)

- Merge pull request #47 from shosca/signals. [Serkan Hosca]
- Fix profile middleware bug by lazily attaching signals. [Miroslav Shubernetskiy]

2.1.52 0.6.1 (2018-08-28)

- Merge pull request #46 from shosca/query-options. [Serkan Hosca]
- Add get query options. [Serkan Hosca]
- Merge pull request #45 from shosca/profiler-middleware. [Serkan Hosca]
- Start/stop in profiler middleware. [Serkan Hosca]

2.1.53 0.6.0 (2018-08-25)

- Merge pull request #40 from shosca/alembic. [Serkan Hosca]
- Fixing import issue after rebase. [Miroslav Shubernetskiy]
- Fixing test_sql not expecting "Running migrations..." messages. [Miroslav Shubernetskiy]
- Not printing "Running migrations..." message when –sql is used. [Miroslav Shubernetskiy]
- Removing import hook. instead adding alembic_app_created signal. [Miroslav Shubernetskiy]
- Checking if migrations are present before configuring alembic. [Miroslav Shubernetskiy]
- Renaming makemigrations to revision and importing migrations.__init__ [Miroslav Shubernetskiy]
- Matching parameters to alembic and minor improvements. [Miroslav Shubernetskiy]
- Added –no-color to all ./manage.py sorcery command in tests. [Miroslav Shubernetskiy]
- Added SQLAlchemy.models_registry. [Miroslav Shubernetskiy]
- Add alembic support. [Serkan Hosca]
- Added prefix to composite columns constraint names. [Miroslav Shubernetskiy]
- Added way to customize metadata options via config. (#43) [Miroslav Shubernetskiy]
- Run tests on pg (#42) [Serkan Hosca]

2.1.54 0.5.5 (2018-07-28)

- Fix scoped session proxying (#41) [Serkan Hosca]

2.1.55 0.5.4 (2018-07-19)

- Adding profiler with middleware and pytest plugin (#39) [Miroslav Shubernetskiy]

2.1.56 0.5.3 (2018-07-18)

- Multi db transaction (#36) [Serkan Hosca]

2.1.57 0.5.2 (2018-07-17)

- Added sane CompositeBase.__bool__ which checks all attributes (#38) [Miroslav Shubernetskiy]

2.1.58 0.5.1 (2018-07-16)

- Allowing to specify via env var some engine options (#37) [Miroslav Shubernetskiy]

2.1.59 0.5.0 (2018-07-05)

- Add namespaced command (#35) [Serkan Hosca]
- Fix unique validator and add declare last signal (#34) [Serkan Hosca]

2.1.60 0.4.13 (2018-07-03)

- Fix unique column validator (#32) [Serkan Hosca]
- Refactored all relations to separate module. also moving declare_first as signal (#31) [Miroslav Shubernetskiy]

2.1.61 0.4.12 (2018-06-30)

- Fix packaging. [Serkan Hosca]

2.1.62 0.4.11 (2018-06-30)

- Snakify table names (#30) [Serkan Hosca]

2.1.63 0.4.10 (2018-06-28)

- Add Unique validator (#29) [Serkan Hosca]

2.1.64 0.4.9 (2018-06-26)

- Fix init kwargs (#28) [Serkan Hosca]
- Add composite cloning and serialization (#27) [Serkan Hosca]

2.1.65 0.4.8 (2018-06-23)

- Add docs (#26) [Serkan Hosca]
- Wire up form to do model clean (#25) [Serkan Hosca]

2.1.66 0.4.7 (2018-06-23)

- Drop drf dependency (#24) [Serkan Hosca]

2.1.67 0.4.6 (2018-06-22)

- Added CompositeField and all related goodies (#23) [Miroslav Shubernetskiy]

2.1.68 0.4.5 (2018-06-14)

- Merge pull request #22 from shosca/config_refactor. [Serkan Hosca]
- Pass along kwargs with custom sqla class. [Serkan Hosca]

2.1.69 0.4.4 (2018-06-13)

- Merge pull request #21 from shosca/config_refactor. [Serkan Hosca]
- Grab only custom sqla class from config. [Serkan Hosca]

2.1.70 0.4.3 (2018-06-09)

- Merge pull request #20 from shosca/config_refactor. [Serkan Hosca]
- Remove engine hacks and refactor config for custom sqla class. [Serkan Hosca]

2.1.71 0.4.2 (2018-06-04)

- 0.4.2. [Serkan Hosca]
- Merge pull request #19 from shosca/inlineformset. [Serkan Hosca]
- Inline formsets. [Serkan Hosca]

2.1.72 0.4.1 (2018-05-31)

- 0.4.1. [Serkan Hosca]
- Merge pull request #18 from shosca/docs. [Serkan Hosca]
- Add more docs for viewsets. [Serkan Hosca]

2.1.73 0.4.0 (2018-05-31)

- 0.4.0. [Serkan Hosca]
- Add basic viewset support. [Serkan Hosca]

2.1.74 0.3.3 (2018-05-21)

- 0.3.3. [Serkan Hosca]
- Merge pull request #15 from shosca/middleware-logger. [Serkan Hosca]
- Add middleware logger. [Serkan Hosca]
- Merge pull request #14 from shosca/docs. [Serkan Hosca]
- More docs. [Serkan Hosca]
- Merge pull request #13 from shosca/docs. [Serkan Hosca]
- Add a test_site and docs. [Serkan Hosca]

2.1.75 0.3.2 (2018-05-17)

- 0.3.2. [Serkan Hosca]
- Merge pull request #12 from shosca/middleware. [Serkan Hosca]
- Refactor middleware. [Serkan Hosca]

2.1.76 0.3.1 (2018-05-17)

- 0.3.1. [Serkan Hosca]
- Merge pull request #11 from shosca/shortcuts. [Serkan Hosca]
- Add get_list_or_404 shortcut. [Serkan Hosca]
- Add get_object_or_404 shortcut. [Serkan Hosca]

2.1.77 0.3.0 (2018-05-16)

- 0.3.0. [Serkan Hosca]
- Merge pull request #10 from shosca/url-refactory. [Serkan Hosca]
- Refactor url generation and allow query settings. [Serkan Hosca]

2.1.78 0.2.8 (2018-05-14)

- 0.2.8. [Serkan Hosca]
- Merge pull request #9 from shosca/refactor-enum. [Serkan Hosca]
- Refactor enum field. [Serkan Hosca]

2.1.79 0.2.7 (2018-05-12)

- 0.2.7. [Serkan Hosca]
- Merge pull request #8 from shosca/enum-field. [Serkan Hosca]
- Enum field fixes. [Serkan Hosca]

2.1.80 0.2.6 (2018-05-09)

- 0.2.6. [Serkan Hosca]
- Merge pull request #7 from shosca/middeware-signals. [Serkan Hosca]
- Add middleware signals. [Serkan Hosca]

2.1.81 0.2.5 (2018-05-09)

- 0.2.5. [Serkan Hosca]
- Merge pull request #6 from shosca/lazy-init. [Serkan Hosca]
- Lazy create engine. [Serkan Hosca]

2.1.82 0.2.4 (2018-05-08)

- 0.2.4. [Serkan Hosca]
- Merge pull request #5 from shosca/field-map. [Serkan Hosca]
- Use mro in python_type field mapping. [Serkan Hosca]

2.1.83 0.2.3 (2018-05-08)

- 0.2.3. [Serkan Hosca]

2.1.84 0.2.2 (2018-05-08)

- 0.2.2. [Serkan Hosca]
- Merge pull request #4 from shosca/app-label-template. [Serkan Hosca]
- Use app config label in template name. [Serkan Hosca]

2.1.85 0.2.1 (2018-05-07)

- 0.2.1. [Serkan Hosca]
- Merge pull request #3 from shosca/transaction. [Serkan Hosca]
- Add transaction tests. [Serkan Hosca]
- Merge pull request #2 from shosca/proxy. [Serkan Hosca]
- Refactor scoped session proxy. [Serkan Hosca]
- Merge pull request #1 from shosca/field-mapping. [Serkan Hosca]
- More field mapping coverage. [Serkan Hosca]

2.1.86 0.2.0 (2018-05-07)

Fix

- Model choice field iterator. [Serkan Hosca]

Other

- 0.2.0. [Serkan Hosca]
- Increase test coverage. [Serkan Hosca]
- Increase test coverage. [Serkan Hosca]

2.1.87 0.1.1 (2018-05-05)

- Fix meta test. [Serkan Hosca]

2.1.88 0.1.0 (2018-05-05)

- Initial commit. [Serkan Hosca]

2.2 API Documentation

2.2.1 django_sorcery

django_sorcery package

Subpackages

django_sorcery.db package

Package with SQLAlchemy abstractions to interact with the database. All tools implemented here assume unit-of-work usage pattern.

Connecting

Connecting to the DB is as simple as using a url with *SQLAlchemy* abstraction:

```
>>> from django_sorcery.db import SQLAlchemy
>>> db = SQLAlchemy('sqlite://')
```

You can also use an alias to connect to a database as well. Just like Django's `DATABASES`, connection settings for aliases are stored in `SQLALCHEMY_CONNECTIONS`. As such all databases are referred by their aliases just like in Django ORM. For example `"default"` is an alias to the default connect:

```
>>> db = SQLAlchemy('default')
```

ORM Goodies

SQLAlchemy is itself also a threadlocal manager that proxies calls to a thread local `Session` instance and provides a couple of shortcuts when interacting with SQLAlchemy session. To have them installed, you can subclass from the declarative base generated by the DB. In addition *SQLAlchemy* also exposes most of SQLALchemy's elements so a single import should suffice to define most tables:

```
>>> class BarModel(db.Model):
...     id = db.Column(db.Integer(), primary_key=True)

>>> class FooModel(db.Model):
...     id = db.Column(db.Integer(), primary_key=True)
...     id2 = db.Column(db.Integer(), primary_key=True)
...     name = db.Column(db.String(length=32))
...     bar_id = db.Column(db.Integer(), db.ForeignKey(BarModel.id, name='FK_foo_bar',
↪ use_alter=True))
...     bar = db.relationship(BarModel)

>>> db.create_all()
```

Doing so will allow to use a couple of useful shortcuts:

Query.get():

Identity map in SQLAlchemy is amazing. Composite keys not so much. Put them together is even worse. `SQLAlchemy` uses a special session which allows keyword arguments when using `.get()`. It is much more explicit and is less error-prone as SQLAlchemy's default implementation expects only positional arguments where order of primary keys matters. For example:

```
>>> db.query(FooModel).get(id=123, id2=456)
```

SQLAlchemy.queryproperty():

Allows to create a property which will normalize to a query object of the model. It will use the correct session within the transaction so no need to pass session around. For example:

```
>>> class MyView(object):
...     queryset = db.queryproperty(FooModel)
```

You can even pass default filtering criteria if needed:

```
>>> class MyView(object):
...     queryset = db.queryproperty(FooModel, to_be_deleted=False)
```

In addition this pattern can be used to implement Django's ORM style model managers:

```
>>> class UserModel(db.Model):
...     id = db.Column(db.Integer(), primary_key=True)
...     username = db.Column(db.String())
...     is_active = db.Column(db.Boolean())
...
...     objects = db.queryproperty()
...     active = db.queryproperty(is_active=True)
```

That can be used directly:

```
>>> UserModel.metadata.create_all(bind=db.engine)
>>> db.add_all([
```

(continues on next page)

(continued from previous page)

```
...     UserModel(id=1, username='foo', is_active=False),
...     UserModel(id=2, username='bar', is_active=True),
... ])
>>> db.flush()

>>> UserModel.objects.all()
[UserModel(id=1, is_active=False, username='foo'), UserModel(id=2, is_
↪active=True, username='bar')]
>>> UserModel.active.all()
[UserModel(id=2, is_active=True, username='bar')]
```

This pattern is very useful when combined with Django style views:

```
>>> class MyView(object):
...     queryset = UserModel.active

>>> MyView().queryset.all()
[UserModel(id=2, is_active=True, username='bar')]
```

Additional filters/options can be applied as well:

```
>>> class MyView(object):
...     queryset = UserModel.active.filter(UserModel.username == 'test')

>>> MyView().queryset.all()
[]
```

Transactions

If you need to explicitly use transactions, you can use *SQLAlchemy.atomic()*:

```
>>> with db.atomic(savepoint=False):
...     _ = db.query(UserModel).filter_by(username='hello').update(dict(username=
↪'world'))

>>> @db.atomic(savepoint=False)
... def do_something():
...     db.query(UserModel).filter_by(username='hello').update(username='world')
```

Warning: You better know what you are doing if you need this. If you are not, sure, **JUST USE THE MIDDLEWARE** (see below).

Django

To complete the unit-of-work *SQLAlchemy* ships with capability to generate Django middleware to commit at the end of the request for the used session within the request:

```
# settings.py
>>> MIDDLEWARE = [
...     # should be last in response life-cycle
...     # or first in middleware list
```

(continues on next page)

(continued from previous page)

```
...     'path.to.db.middleware',
...     # rest of middleware
... ]
```

Subpackages

django_sorcery.db.alembic package

Submodules

django_sorcery.db.alembic.base module

Default alembic config things.

django_sorcery.db.alembic.base.**setup_config**(*config*)
The default alembic config created handler.

django_sorcery.db.alembic.signals module

Default alembic signal configuration.

django_sorcery.db.alembic.signals.**include_object**(*obj*, *name*, *type_*, *reflected*, *compare_to*)
The default include_object handler for alembic, delegates to alembic_include_object signal.

The return value will be considered True if all the signal handlers return True for the object to be included in migration.

django_sorcery.db.alembic.signals.**process_revision_directives**(*context*, *revision*, *directives*)
The default process_revision_directives handler for alembic, delegates to alembic_process_revision_directives.

django_sorcery.db.meta package

Submodules

django_sorcery.db.meta.base module

Base model metadata things.

class django_sorcery.db.meta.base.**model_info_meta**
Bases: type

Model metadata singleton.

django_sorcery.db.meta.column module

Django-esque field metadata and interface providers.

class django_sorcery.db.meta.column.**boolean_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for boolean columns.

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**choice_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for enum columns with simple choices.

default_form_class
alias of `django.forms.fields.TypedChoiceField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: `object`

A helper class that makes sqlalchemy property and column inspection easier.

attname

attribute

choices

clean(*value*, *instance*)
Convert the value's type and run validation.

Validation errors from to_python() and validate() are propagated. Return the correct value if no error is raised.

coercer
Form field to be used to coerce data types.

column

default

default_error_messages = {'blank': 'This field cannot be blank.', 'invalid_choice':

default_form_class = None

empty_values

error_messages

field_kwargs

form_class

formfield(*form_class=None*, ***kwargs*)
Returns the form field for the field.

help_text

is_relation = False

label

name

null

parent

parent_model

property

required

run_validators(*value*)
Run field's validators and raise ValidationError if necessary.

to_python(*value*)
Convert input value to appropriate python object.

unique

validate(*value*, *instance*)
Validate value and raise ValidationError if necessary.

validators

widget

class django_sorcery.db.meta.column.**date_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)
Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for date columns.

coercer
Form field to be used to coerce data types.

default_form_class
alias of `django.forms.fields.DateField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**datetime_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)
Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for datetime columns.

coercer
Form field to be used to coerce data types.

default_form_class
alias of `django.forms.fields.DateTimeField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**enum_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)
Bases: *django_sorcery.db.meta.column.choice_column_info*

Provides meta info for enum columns with Enum choices.

default_form_class
alias of *django_sorcery.fields.EnumField*

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**float_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for float columns.

default_form_class
alias of `django.forms.fields.FloatField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**integer_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for integer columns.

default_form_class
alias of `django.forms.fields.IntegerField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**interval_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for interval columns.

default_form_class
alias of `django.forms.fields.DurationField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**numeric_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for numeric columns.

decimal_places

default_form_class
alias of `django.forms.fields.DecimalField`

max_digits

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**string_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for string columns.

default_form_class
alias of `django.forms.fields.CharField`

to_python(*value*)
Convert input value to appropriate python object.

class django_sorcery.db.meta.column.**text_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.string_column_info*

Provides meta info for text columns.

class django_sorcery.db.meta.column.**time_column_info**(*column*, *prop=None*, *parent=None*, *name=None*)

Bases: *django_sorcery.db.meta.column.column_info*

Provides meta info for time columns.

default_form_class

alias of `django.forms.fields.TimeField`

to_python(*value*)

Convert input value to appropriate python object.

django_sorcery.db.meta.composite module

Metadata for composite sqlalchemy properties.

class django_sorcery.db.meta.composite.**composite_info**(*composite*, *parent=None*)

Bases: `object`

A helper class that makes sqlalchemy composite model inspection easier.

attribute

Returns composite field instrumented attribute for generating query expressions.

clean_fields(*instance*, *exclude=None*)

Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

field_names

Returns field names used in composite.

full_clean(*instance*, *exclude=None*)

Call clean_fields(), clean(), and run_validators() on the composite model.

Raise a ValidationError for any errors that occur.

model_class

Returns the composite class.

name

Returns composite field name.

parent

parent_model

Returns the model class that the attribute belongs to.

prop

properties

run_validators(*instance*)

Run composite field's validators and raise ValidationError if necessary.

django_sorcery.db.meta.model module

Metadata for sqlalchemy models.

`django_sorcery.db.meta.model.`**`Identity`**
alias of `django_sorcery.db.meta.model.Key`

class `django_sorcery.db.meta.model.`**`model_info`**(*model*)
Bases: `object`

A helper class that makes sqlalchemy model inspection easier.

`app_config`

`app_label`

`clean_fields`(*instance*, *exclude=None*, ***kwargs*)
Clean all fields on object.

`clean_nested_fields`(*instance*, *exclude=None*, ***kwargs*)
Clean all nested fields which includes composites.

`clean_relation_fields`(*instance*, *exclude=None*, ***kwargs*)
Clean all relation fields.

`column_properties`

`composites`

`concrete_fields`

`field_names`

`fields`

`full_clean`(*instance*, *exclude=None*, ***kwargs*)
Run model's full clean chain.

This will run all of these in this order:

- will validate all columns by using `clean_<column>` methods
- will validate all nested objects (e.g. composites) with `full_clean`
- will run through all registered validators on `validators` attribute
- will run full model validation with `self.clean()`
- if `recursive` kwarg is provided, will recursively clean all relations. Useful when all models need to be explicitly cleaned without flushing to DB.

`get_field`(*field_name*)

`get_key`(*instance*)
Returns the primary key tuple from the `instance`

`identity_key_from_dict`(*kwargs*)
Returns identity key from a dictionary for the given model.

`identity_key_from_instance`(*instance*)
Returns the primary key tuple from the `instance`

`label`

`label_lower`

`local_fields`

mapper

model

model_class

model_name

object_name

opts

ordering

primary_keys

primary_keys_from_dict(*kwargs*)
Returns the primary key tuple from a dictionary to be used in a sqlalchemy query.get() call.

primary_keys_from_instance(*instance*)
Return a dict containing the primary keys of the `instance`

private_fields

properties

relationships

run_validators(*instance*, *exclude=None*, ***kwargs*)
Check all model validators registered on `validators` attribute.

sa_state(*instance*)
Returns sqlalchemy instance state.

unique_together

verbose_name

verbose_name_plural

django_sorcery.db.meta.relations module

Metadata for sqlalchemy model relationships.

class django_sorcery.db.meta.relations.**relation_info**(*relationship*)
Bases: `object`

A helper class that makes sqlalchemy relationship property inspection easier.

attribute

direction

field_kwargs

foreign_keys

formfield(*form_class=None*, ***kwargs*)

get_form_class()

local_remote_pairs

local_remote_pairs_for_identity_key

name

parent_mapper

parent_model

parent_table

related_mapper

related_model

related_table

relationship

uselist

Submodules

django_sorcery.db.composites module

Support for reusable sqlalchemy composite fields.

class django_sorcery.db.composites.**BaseComposite**(**args*, ***kwargs*)
Bases: object

Base class for creating composite classes which *CompositeField* will understand.

For example:

```
class MyComposite(object):
    foo = db.Column(db.Integer())
    bar = db.Column(db.Integer())

class MyModel(db.Model):
    test = db.CompositeField(MyComposite)
    # both test_foo and test_bar columns will be added to model
    # their instrumented properties will be _test_foo and _test_bar
```

as_dict()
Serializer composite to a dictionary.

class django_sorcery.db.composites.**CompositeField**(*class_*, ***kwargs*)
Bases: sqlalchemy.orm.descriptor_props.CompositeProperty

Composite field which understands composite objects with builtin columns.

See *BaseComposite* for examples.

instrument_class(*mapper*)
Hook called by the Mapper to the property to initiate instrumentation of the class attribute managed by this MapperProperty.

The MapperProperty here will typically call out to the attributes module to set up an InstrumentedAttribute.

This step is the first of two steps to set up an InstrumentedAttribute, and is called early in the mapper setup process.

The second step is typically the init_class_attribute step, called from StrategizedProperty via the post_instrument_class() hook. This step assigns additional state to the InstrumentedAttribute (specifically the "impl") which has been determined after the MapperProperty has determined what kind of persistence management it needs to do (e.g. scalar, object, collection, etc).

django_sorcery.db.fields module

Django-esque declarative fields for sqlalchemy.

class django_sorcery.db.fields.**BigIntegerField**(**args*, ***kwargs*)
Bases: django_sorcery.db.fields.ValidateIntegerFieldMixin, *django_sorcery.db.fields.Field*

Django like big integer field.

default_validators = [<function validate_integer>]

form_class
alias of django.forms.fields.IntegerField

type_class
alias of sqlalchemy.sql.sqltypes.BigInteger

class django_sorcery.db.fields.**BinaryField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like binary field.

length_is_required = False

type_class
alias of sqlalchemy.sql.sqltypes.LargeBinary

class django_sorcery.db.fields.**BooleanField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like boolean field.

form_class
alias of django.forms.fields.BooleanField

get_column_kwargs(*kwargs*)
Returns sqlalchemy column kwargs.

get_type_kwargs(*type_class*, *kwargs*)
Returns sqlalchemy type kwargs.

type_class
alias of sqlalchemy.sql.sqltypes.Boolean

class django_sorcery.db.fields.**CharField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like char field.

form_class
alias of django.forms.fields.CharField

get_type_kwargs(*type_class*, *kwargs*)
Returns sqlalchemy type kwargs.

get_validators(*validators*)
Returns django validators for the field.

length_is_required = True

type_class
alias of sqlalchemy.sql.sqltypes.String

class django_sorcery.db.fields.**DateField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like date field.

form_class
alias of `django.forms.fields.DateField`

type_class
alias of `sqlalchemy.sql.sqltypes.Date`

class django_sorcery.db.fields.**DateTimeField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like datetime field.

form_class
alias of `django.forms.fields.DateTimeField`

type_class
alias of `sqlalchemy.sql.sqltypes.DateTime`

class django_sorcery.db.fields.**DecimalField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like decimal field.

form_class
alias of `django.forms.fields.DecimalField`

get_type_kwargs(*type_class*, *kwargs*)
Returns sqlalchemy type kwargs.

get_validators(*validators*)
Returns django validators for the field.

type_class
alias of `sqlalchemy.sql.sqltypes.Numeric`

class django_sorcery.db.fields.**DurationField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like duration field.

form_class
alias of `django.forms.fields.DurationField`

type_class
alias of `sqlalchemy.sql.sqltypes.Interval`

class django_sorcery.db.fields.**EmailField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.CharField*

Django like email field.

default_validators = [<django.core.validators.EmailValidator object>]

form_class
alias of `django.forms.fields.EmailField`

class django_sorcery.db.fields.**EnumField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like choice field that uses an enum sqlalchemy type.

get_form_class(*kwargs*)
Returns form field class.

get_type(*type_class*, *type_kwargs*)
Returns sqlalchemy column type instance for the field.

get_type_kwargs(*type_class*, *kwargs*)
Returns sqlalchemy type kwargs.

type_class
alias of `sqlalchemy.sql.sqltypes.Enum`

class django_sorcery.db.fields.**Field**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.schema.Column`

Base django-esque field.

default_validators = []

form_class = None

get_column_kwargs(*kwargs*)
Returns sqlalchemy column kwargs.

get_form_class(*kwargs*)
Returns form field class.

get_type(*type_class*, *type_kwargs*)
Returns sqlalchemy column type instance for the field.

get_type_class(*kwargs*)
Returns sqlalchemy column type.

get_type_kwargs(*type_class*, *kwargs*)
Returns sqlalchemy type kwargs.

get_validators(*validators*)
Returns django validators for the field.

type_class = None

widget_class = None

class django_sorcery.db.fields.**FloatField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like float field.

form_class
alias of `django.forms.fields.FloatField`

get_type_kwargs(*type_class*, *kwargs*)
Returns sqlalchemy type kwargs.

type_class
alias of `sqlalchemy.sql.sqltypes.Float`

class django_sorcery.db.fields.**IntegerField**(**args*, ***kwargs*)
Bases: `django_sorcery.db.fields.ValidateIntegerFieldMixin`, *django_sorcery.db.fields.Field*

Django like integer field.

default_validators = [<function validate_integer>]

form_class
alias of `django.forms.fields.IntegerField`

type_class
alias of `sqlalchemy.sql.sqltypes.Integer`

class `django_sorcery.db.fields.`**NullBooleanField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.BooleanField*

Django like nullable boolean field.

form_class
alias of `django.forms.fields.NullBooleanField`

get_column_kwargs(*kwargs*)
Returns sqlalchemy column kwargs.

class `django_sorcery.db.fields.`**SlugField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.CharField*

Django like slug field.

default_validators = [<django.core.validators.RegexValidator object>]

form_class
alias of `django.forms.fields.SlugField`

class `django_sorcery.db.fields.`**SmallIntegerField**(**args*, ***kwargs*)
Bases: `django_sorcery.db.fields.ValidateIntegerFieldMixin`, *django_sorcery.db.fields.Field*

Django like small integer field.

default_validators = [<function validate_integer>]

form_class
alias of `django.forms.fields.IntegerField`

type_class
alias of `sqlalchemy.sql.sqltypes.SmallInteger`

class `django_sorcery.db.fields.`**TextField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.CharField*

Django like text field.

form_class
alias of `django.forms.fields.CharField`

length_is_required = False

type_class
alias of `sqlalchemy.sql.sqltypes.Text`

widget_class
alias of `django.forms.widgets.Textarea`

class `django_sorcery.db.fields.`**TimeField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.Field*

Django like time field.

form_class
alias of `django.forms.fields.TimeField`

type_class
alias of sqlalchemy.sql.sqltypes.Time

class django_sorcery.db.fields.**TimestampField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.DateTimeField*

Django like datetime field that uses timestamp sqlalchemy type.

type_class
alias of sqlalchemy.sql.sqltypes.TIMESTAMP

class django_sorcery.db.fields.**URLField**(**args*, ***kwargs*)
Bases: *django_sorcery.db.fields.CharField*

Django like url field.

default_validators = [<django.core.validators.URLValidator object>]

form_class
alias of django.forms.fields.URLField

django_sorcery.db.middleware module

Django middleware support for sqlalchemy.

class django_sorcery.db.middleware.**BaseMiddleware**(*get_response=None*)
Bases: object

Base middleware implementation that supports unit of work per request for django.

logger = <Logger django_sorcery.db.middleware (WARNING)>

process_request(*request*)
Hook for adding arbitrary logic to request processing.

process_response(*request*, *response*)
Commits or rollbacks scoped sessions depending on status code then removes them.

return_response(*request*, *response*)
Hook for adding arbitrary logic to response processing.

class django_sorcery.db.middleware.**SQLAlchemyDBMiddleware**(*get_response=None*)
Bases: *django_sorcery.db.middleware.BaseMiddleware*

A base SQLAlchemy db middleware.

Used by SQLAlchemy to provide a default middleware for a single db, it will first try to flush and if successfull, proceed with commit. If there are any errors during flush, will issue a rollback.

commit(*request*, *response*)
Commits current scoped session.

db = None

flush(*request*, *response*)
Flushes current scoped session.

remove(*request*, *response*)
Removes current scoped session.

rollback(*request*, *response*)
Rolls back current scoped session.

class django_sorcery.db.middleware.**SQLAlchemyMiddleware**(*get_response=None*)
Bases: *django_sorcery.db.middleware.SQLAlchemyDBMiddleware*

A sqlalchemy middleware that manages all the dbs configured and initialized.

it will first try to flush all the configured and initialized SQLAlchemy instances and if successfull, proceed with commit. If there are any errors during flush, all transactions will be rolled back.

db = {}

django_sorcery.db.mixins module

Common mixins used in models.

class django_sorcery.db.mixins.**CleanMixin**
Bases: object

Mixin for adding django-style `full_clean` validation to any object.

Base model in `sqlalchemy.SQLAlchemy` already uses this mixin applied.

For example:

```
class Address(db.Model):
    city = db.Column(db.String(20))
    state = db.Column(db.String(2))
    date = db.Column(db.Date())

    validators = [
        ValidateTogetherModelFields(["city", "state"]),
    ]

    def clean_date(self):
        if self.date > datetime.date.today():
            raise ValidationError("Cant pick future date")

    def clean(self):
        if self.date.year < 1776 and self.state == "NY":
            raise ValidationError("NY state did not exist before 1776")
```

clean(***kwargs*)
Hook for adding custom model validations before model is flushed.

Should raise `ValidationError` if any errors are found.

clean_fields(*exclude=None*, ***kwargs*)
Clean all fields on object.

clean_nested_fields(*exclude=None*, ***kwargs*)
Clean all nested fields which includes composites.

clean_relation_fields(*exclude*, ***kwargs*)
Clean all relation fields.

full_clean(*exclude=None*, ***kwargs*)
Run model's full clean chain.

This will run all of these in this order:

- will validate all columns by using `clean_<column>` methods
- will validate all nested objects (e.g. composites) with `full_clean`

- will run through all registered validators on `validators` attribute
- will run full model validation with `self.clean()`
- if `recursive` kwarg is provided, will recursively clean all relations. Useful when all models need to be explicitly cleaned without flushing to DB.

run_validators(***kwargs*)
Check all model validators registered on `validators` attribute.

django_sorcery.db.models module

sqlalchemy model related things.

class django_sorcery.db.models.**Base**
Bases: *django_sorcery.db.mixins.CleanMixin*

Base model class for SQLAlchemy.

Can be overwritten by subclasses:

query_class = None

Automatically added by declarative base for easier querying:

query = None objects = None

as_dict()
Return a dict of column attributes.

class django_sorcery.db.models.**BaseMeta**(*classname*, *bases*, *dict_*, ***kw*)
Bases: sqlalchemy.orm.decl_api.DeclarativeMeta

Base metaclass for models which registers models to DB model registry when models are created.

django_sorcery.db.models.**autocoerce**(*cls*)
This function automatically registers attribute events that coerces types for the attribute using django's form fields for a given model classs. If no class is provided, it will wire up coersion for all mappers so it can be used as a class decorator or globally.

```
@autocoerce_properties
class MyModel(db.Model):
    ...
```

or:

```
class MyModel(db.Model):
    ...

autocoerce_properties()
```

Since django form fields are used for coersion, localization settings such as *USE_THOUSAND_SEPARATOR*, *DATE_INPUT_FORMATS* and *DATETIME_INPUT_FORMATS* control type conversions.

django_sorcery.db.models.**autocoerce_properties**(**attrs*)
This function automatically registers attribute events that coerces types for given attributes using django's form fields.

::

class MyModel(db.Model): field1 = Column(...) field2 = Column(...) field3 = Column(...) ...

autocoerce_properties(MyModel.field1, MyModel.field2) # Will only force autocoersion on field1 and field2

django_sorcery.db.models.**clone**(*instance*, **rels*, ***kwargs*)

instance: Model a model instance

relations: list or relations or a tuple of relation and kwargs for that relation relationships to be cloned with relationship and optionally kwargs

kwargs: dict string of any attribute values to be overridden

django_sorcery.db.models.**deserialize**(*model*, *data*)

model: a model class

data: dict values

django_sorcery.db.models.**full_clean_flush_handler**(*session*, ***kwargs*)
Signal handler for executing `full_clean` on all dirty and new objects in session.

django_sorcery.db.models.**instant_defaults**(*cls*)
This function automatically registers attribute events that sets the column defaults to a model instance at model instance initialization provided that default values are simple types:

```
@instant_defaults
class MyModel(db.Model):
    attr = db.Column(..., default=1)

assert MyModel().default == 1
```

django_sorcery.db.models.**serialize**(*instance*, **rels*)

instance: a model instance

rels: list of relations relationships to be serialized

django_sorcery.db.models.**simple_repr**(*instance*, *fields=None*)

instance: Model a model instance

fields: list list of fields to display on repr

django_sorcery.db.profiler module

sqlalchemy profiling things.

class django_sorcery.db.profiler.**Query**(*timestamp*, *statement*, *parameters*, *duration*)
Bases: `tuple`

duration
Alias for field number 3

parameters
Alias for field number 2

statement
Alias for field number 1

timestamp
Alias for field number 0

class django_sorcery.db.profiler.**SQLAlchemyProfiler**(*exclude=None*, *record_queries=True*)

Bases: object

A sqlalchemy profiler that hooks into sqlalchemy engine and pool events and generate stats.

Can also capture executed sql statements. Useful for profiling or testing sql statements.

clear()
Clears collected stats.

counts
Returns a dict of counts per sqlalchemy event operation like executed statements, commits, rollbacks, etc..

duration
Return total statement execution duration.

queries
Returns executed statements.

start()
Starts profiling by wiring up sqlalchemy events.

stats
Returns profiling stats.

stop()
Stops profiling by detaching wired up sqlalchemy events.

class django_sorcery.db.profiler.**SQLAlchemyProfilingMiddleware**(*get_response=None*)

Bases: object

Django middleware that provides sqlalchemy statistics.

header_results
Determines if stats should be returned as headers or not.

log(***kwargs*)
Log sqlalchemy stats for current request.

log_results
Determines if stats should be logged or not.

logger = <Logger django_sorcery.db.profiler (WARNING)>

process_request(*request*)
Starts profiling and resets stats doe the request.

process_response(*request*, *response*)
Logs current request stats and also returns stats as headers.

start()
Starts profiling and disables restarts.

django_sorcery.db.query module

sqlalchemy query related things.

class django_sorcery.db.query.**Operation**(*name*, *args*, *kwargs*)

Bases: tuple

args
Alias for field number 1

kwargs
Alias for field number 2

name
Alias for field number 0

class django_sorcery.db.query.**Query**(*entities*, *session=None*)
Bases: sqlalchemy.orm.query.Query

A customized sqlalchemy query.

filter(**args*, ***kwargs*)
Standard SQLAlchemy filtering plus django-like expressions can be provided:

For example:

```
MyModel.objects.filter(MyModel.id == 5)
MyModel.objects.filter(id=5)
MyModel.objects.filter(id__gte=5)
MyModel.objects.filter(relation__id__gte=5)
```

get(**args*, ***kwargs*)
Return an instance based on the given primary key identifier, either as args or kwargs for composite keys.

If no instance is found, returns None.

order_by(**criterion*)
Standard SQLAlchemy ordering plus django-like expressions can be provided:

For example:

```
MyModel.objects.order_by("-id")
MyModel.objects.order_by("name")
```

class django_sorcery.db.query.**QueryProperty**(*db*, *model=None*, **args*, ***kwargs*)
Bases: object

A property class that returns a session scoped query object against the class when called. Used by the SQLAlchemy.queryproperty

For example:

```
>>> class MyView(object):
...     queryset = db.queryproperty(FooModel)
```

You can even pass default filtering criteria if needed:

```
>>> class MyView(object):
...     queryset = db.queryproperty(FooModel, to_be_deleted=False)
```

In addition this pattern can be used to implement Django's ORM style model managers:

```
>>> class UserModel(db.Model):
...     id = db.Column(db.Integer(), primary_key=True)
...     username = db.Column(db.String())
...     is_active = db.Column(db.Boolean())
...
...     active = db.queryproperty(is_active=True)
```

That can be used directly:

```
>>> UserModel.metadata.create_all(bind=db.engine)
>>> db.add_all([
...     UserModel(id=1, username='foo', is_active=False),
...     UserModel(id=2, username='bar', is_active=True),
... ])
>>> db.flush()

>>> UserModel.objects.all()
[UserModel(id=1, is_active=False, username='foo'), UserModel(id=2, is_active=True,
↪ username='bar')]
>>> UserModel.active.all()
[UserModel(id=2, is_active=True, username='bar')]
```

This pattern is very useful when combined with Django style views:

```
>>> class MyView(object):
...     queryset = UserModel.active

>>> MyView().queryset.all()
[UserModel(id=2, is_active=True, username='bar')]
```

Additional filters/options can be applied as well:

```
>>> class MyView(object):
...     queryset = UserModel.active.filter(UserModel.username == 'test')

>>> MyView().queryset.all()
[]
```

django_sorcery.db.relations module

sqlalchemy relationship related things.

class django_sorcery.db.relations.**RelationsMixin**

Bases: object

Mixin that provides django like shortcuts for relationships.

ManyToMany(*remote_cls*, *table_name=None*, ***kwargs*)

Use an event to build many-to-many relationship on a model and auto generates an association table or if a model is provided as secondary argument:

```
class ModelOne(db.Model):
    pk = db.Column(.., primary_key=True)
    m2s = db.ManyToMany("ModelTwo", backref="m1s", table_name='m1m2s', ...)

class ModelTwo(db.Model):
    pk = db.Column(.., primary_key=True)
    ...
```

or with back_populates:

```
class ModelOne(db.Model):
    pk = db.Column(.., primary_key=True)
    m2s = db.ManyToMany("ModelTwo", back_populates="m1s", table_name='m1m2s',
↪ ...)
```

(continues on next page)

(continued from previous page)

```
class ModelTwo(db.Model):
    pk = db.Column(.., primary_key=True)
    m1s = db.ManyToMany("ModelOne", back_populates="m2s", table_name='m1m2s',
↪ ...)
```

will create ModelOne.m2s and ModelTwo.m1s relationship thru a provided secondary argument. If no secondary argument is provided, table_name is required as it will be used for the autogenerated association table.

In the case of back_populates you have to provide the same table_name argument on both many-to-many declarations

ManyToOne(*remote_cls*, ***kwargs*)

Use an event to build many-to-one relationship on a model and auto generates foreign key relationship on the remote table:

```
class ModelOne(db.Model):
    pk = db.Column(.., primary_key=True)
    m2 = db.ManyToOne("ModelTwo", ...)

class ModelTwo(db.Model):
    pk = db.Column(.., primary_key=True)
    ...
```

will create ModelOne.m2_pk automatically for the relationship

OneToMany(*remote_cls*, ***kwargs*)

Use an event to build one-to-many relationship on a model and auto generates foreign key relationship from the remote table:

class ModelOne(db.Model): pk = db.Column(.., primary_key=True) m2 = db.OneToMany("ModelTwo", ...)

class ModelTwo(db.Model): pk = db.Column(.., primary_key=True) ...

will create ModelTwo.m1_pk automatically for the relationship

OneToOne(*remote_cls*, ***kwargs*)

Use an event to build one-to-many relationship on a model and auto generates foreign key relationship from the remote table:

class ModelOne(db.Model): pk = db.Column(.., primary_key=True) m2 = db.OneToOne("ModelTwo", ...)

class ModelTwo(db.Model): pk = db.Column(.., primary_key=True) ...

will create ModelTwo.m1_pk automatically for the relationship

django_sorcery.db.relations.**declare_first_relationships_handler**(*cls*)

Declare first signal handler which connects relationships on the class.

Can be called multiple times so once relationships are set, they are removed from model

django_sorcery.db.session module

sqlalchemy session related things.

class django_sorcery.db.session.**SignallingSession**(**args*, ***kwargs*)
Bases: sqlalchemy.orm.session.Session

A custom sqlalchemy session implementation that provides signals.

query(**args*, ***kwargs*)
Override to try to use the model.query_class.

django_sorcery.db.session.**after_commit**(*session*)

django_sorcery.db.session.**after_flush**(*session*, *flush_context*)

django_sorcery.db.session.**after_rollback**(*session*)

django_sorcery.db.session.**before_commit**(*session*)

django_sorcery.db.session.**before_flush**(*session*, *flush_context*, *instances*)

django_sorcery.db.session.**record_models**(*session*, *flush_context=None*, *instances=None*)

django_sorcery.db.signals module

Signals

Implements some basic signals using blinker

class django_sorcery.db.signals.**Namespace**
Bases: blinker.base.Namespace

A signal namespace that also manages scoped signals.

scoped_signals
Returns all scoped signals.

scopedsignal(*name*, *doc=None*)
Returns the scoped signal for a given name.

class django_sorcery.db.signals.**ScopedSignal**(*name*, *doc=None*)
Bases: blinker.base.NamedSignal

Same as NamedSignal but signal is scoped to a thread.

In other words, if a receiver is attached within a specific thread, even if signal is sent in another thread, in that other thread no receivers will be present and hence nothing will execute. Useful for adding one-off signal handlers for example to be executed at the end of unit-of-work (e.g. request) without adding a possibility that another thread might start executing the receiver.

cleanup()
Cleans up signal for the current thread scope.

receivers
Return all thread scoped receivers.

django_sorcery.db.sqlalchemy module

SQLAlchemy goodies that provides a nice interface to using sqlalchemy with django.

class django_sorcery.db.sqlalchemy.**SQLAlchemy**(*url*, ***kwargs*)
Bases: *django_sorcery.db.relations.RelationsMixin*

This class itself is a scoped session and provides very thin and useful abstractions and conventions for using sqlalchemy with django.

class BaseComposite(**args*, ***kwargs*)

Bases: `object`

Base class for creating composite classes which *CompositeField* will understand.

For example:

```
class MyComposite(object):
    foo = db.Column(db.Integer())
    bar = db.Column(db.Integer())

class MyModel(db.Model):
    test = db.CompositeField(MyComposite)
    # both test_foo and test_bar columns will be added to model
    # their instrumented properties will be _test_foo and _test_bar
```

as_dict()

Serializer composite to a dictionary.

class CompositeField(*class_*, ***kwargs*)

Bases: `sqlalchemy.orm.descriptor_props.CompositeProperty`

Composite field which understands composite objects with builtin columns.

See *BaseComposite* for examples.

instrument_class(*mapper*)

Hook called by the Mapper to the property to initiate instrumentation of the class attribute managed by this MapperProperty.

The MapperProperty here will typically call out to the attributes module to set up an InstrumentedAttribute.

This step is the first of two steps to set up an InstrumentedAttribute, and is called early in the mapper setup process.

The second step is typically the init_class_attribute step, called from StrategizedProperty via the post_instrument_class() hook. This step assigns additional state to the InstrumentedAttribute (specifically the "impl") which has been determined after the MapperProperty has determined what kind of persistence management it needs to do (e.g. scalar, object, collection, etc).

Table(*name*, **args*, ***kwargs*)

Returns a sqlalchemy table that is automatically added to metadata.

add(**args*, ***kwargs*)

add_all(**args*, ***kwargs*)

args(**args*, ***kwargs*)

Useful for setting table args and mapper args on models.

atomic(*savepoint=True*)

Create a savepoint or transaction scope.

autocommit

autoflush

begin(**args*, ***kwargs*)

begin_nested(**args*, ***kwargs*)

bind

bind_mapper(**args*, ***kwargs*)

bind_table(**args*, ***kwargs*)

bulk_insert_mappings(**args*, ***kwargs*)

bulk_save_objects(**args*, ***kwargs*)

bulk_update_mappings(**args*, ***kwargs*)

close(**args*, ***kwargs*)

close_all(**args*, ***kwargs*)

commit(**args*, ***kwargs*)

connection(**args*, ***kwargs*)

connection_callable

create_all()
Create the schema in db.

delete(**args*, ***kwargs*)

deleted

dirty

dispatch

drop_all()
Drop the schema in db.

enable_baked_queries

enable_relationship_loading(**args*, ***kwargs*)

engine
Current engine.

execute(**args*, ***kwargs*)

expire(**args*, ***kwargs*)

expire_all(**args*, ***kwargs*)

expire_on_commit

expunge(**args*, ***kwargs*)

expunge_all(**args*, ***kwargs*)

flush(**args*, ***kwargs*)

future

get(**args*, ***kwargs*)

get_bind(**args*, ***kwargs*)

get_nested_transaction(**args*, ***kwargs*)

get_transaction(**args*, ***kwargs*)

hash_key

identity_key(**args*, ***kwargs*)

identity_map

in_nested_transaction(**args*, ***kwargs*)

in_transaction(**args*, ***kwargs*)

info

inspector
Returns engine inspector.

Useful for querying for db schema info.

invalidate(**args*, ***kwargs*)

is_active

is_modified(**args*, ***kwargs*)

make_middleware()
Creates a middleware to be used in a django application.

merge(**args*, ***kwargs*)

metadata_class
alias of `sqlalchemy.sql.schema.MetaData`

model_class
alias of *django_sorcery.db.models.Base*

new

no_autoflush

object_session(**args*, ***kwargs*)

prepare(**args*, ***kwargs*)

query(**args*, ***kwargs*)

query_class
alias of *django_sorcery.db.query.Query*

queryproperty(**args*, ***kwargs*)
Generate a query property for a model.

refresh(**args*, ***kwargs*)

registry
Returns scoped registry instance.

registry_class
alias of `sqlalchemy.util._collections.ThreadLocalRegistry`

remove()
Remove the current scoped session.

rollback(**args*, ***kwargs*)

scalar(**args*, ***kwargs*)

scalars(**args*, ***kwargs*)

session(***kwargs*)
Return the current session, creating it if necessary using session_factory for the current scope Any kwargs provided will be passed on to the session_factory.

If there's already a session in current scope, will raise InvalidRequestError

session_class
alias of *django_sorcery.db.session.SignallingSession*

session_factory
Current session factory to create sessions.

transaction

twophase

django_sorcery.db.sqlalchemy.**instrument**(*name*)

django_sorcery.db.sqlalchemy.**makeprop**(*name*)

django_sorcery.db.transaction module

sqlalchemy transaction related things.

class django_sorcery.db.transaction.**TransactionContext**(**dbs*, ***kwargs*)
Bases: object

Transaction context manager for maintaining a transaction or savepoint.

django_sorcery.db.url module

django_sorcery.db.url.**boolean**(*x*)

django_sorcery.db.url.**get_settings**(*alias*)
Returns database settings from either SQLALCHEMY_CONNECTIONS setting or DATABASES setting.

django_sorcery.db.url.**importable**(*x*)

django_sorcery.db.url.**importable_list**(*x*)

django_sorcery.db.url.**importable_list_tuples**(*x*)

django_sorcery.db.url.**integer**(*x*)

django_sorcery.db.url.**make_url**(*alias_or_url*)

alias_or_url: str name of the alias or url as string

django_sorcery.db.url.**make_url_from_settings**(*alias*)

alias: str name of the alias

Overall settings are very similar with django database settings with a few extra keys.

USER - database user

PASSWORD - database user password

HOST - database host

NAME - database name

PORT - database name

DIALECT - dialect to be used in url, if not provided, will use the DIALECT_MAP to figure out a dialect to be used in sqlalchemy url

DRIVER - If provided, will be used as the driver in sqlalchemy url

SQLALCHEMY - If provided, a custom sqlalchemy.SQLAlchemy class to be used

`QUERY` - querystring arguments for sqlalchemy url

`ALCHEMY_OPTIONS` - Optional arguments to be used to initialize the `sqlalchemy.SQLAlchemy` instance

- `session_class` - a custom session class to be used
- `registry_class` - a custom registy class to be used for scoping
- `model_class` - a custom base model class to be used for declarative base models.
- `metadata_class` - a custom metadata class used in delclarative models.
- `metadata_options` - custom options to use in metadata creation such as specifying naming conventions.
- `engine_options` - arguments for sqlalchemy `create_engine`
- `session_options` - arguments for sqlalchemy `sessionmaker`

Other options are ignored.

`django_sorcery.db.url.`**`string`**(*x*)

`django_sorcery.db.url.`**`string_list`**(*x*)

django_sorcery.db.utils module

class `django_sorcery.db.utils.`**`dbdict`**

Bases: `dict`

Holds all configured `sqlalchemy.SQLAlchemy` instances.

`atomic`(*savepoint=True*)

Returns a context manager/decorator that guarantee atomic execution of a given block or function across all configured and initialized SQLAlchemy instances.

`commit`()

Applies commit on all registered databases.

`flush`()

Applies flush on all registered databases.

`get`(*alias=None*, *cls=<class 'django_sorcery.db.sqlalchemy.SQLAlchemy'>*, ***kwargs*)

Returns a `sqlalchemy.SQLAlchemy` instance from configuration and registers it.

Can return a custom `sqlalchemy.SQLAlchemy` instance thru args or thru `SQLALCHEMY` database setting in configuration.

`remove`()

Applies remove on all registered databases.

`rollback`()

Applies rollback on all registered databases.

`update`(`[`*E*`]`, ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`django_sorcery.db.utils.`**`index_foreign_keys`**(**args*)

Generates indexes for all foreign keys for a table or metadata tables.

django_sorcery.formsets package

Submodules

django_sorcery.formsets.base module

Helper functions for creating FormSet classes from SQLAlchemy models.

class django_sorcery.formsets.base.**BaseModelFormSet**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *queryset=None*, *initial=None*, ***kwargs*)

Bases: django.forms.formsets.BaseFormSet

A FormSet for editing a queryset and/or adding new objects to it.

absolute_max = 2000

add_fields(*form*, *index*)
A hook for adding extra fields on to each form instance.

delete_existing(*obj*, ***kwargs*)
Deletes an existing model instance.

get_queryset()
Returns a query for the model.

initial_form_count()
Return the number of forms that are required in this FormSet.

model = None

save(*flush=True*, ***kwargs*)
Save model instances for every form, adding and changing instances as necessary, and return the list of instances.

session = None

unique_fields = {}

django_sorcery.formsets.base.**modelformset_factory**(*model*, *form=<class 'django_sorcery.forms.ModelForm'>*, *formfield_callback=None*, *formset=<class 'django_sorcery.formsets.base.BaseModelFormSet'>*, *extra=1*, *can_delete=False*, *can_order=False*, *max_num=None*, *fields=None*, *exclude=None*, *widgets=None*, *validate_max=False*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *min_num=None*, *validate_min=False*, *field_classes=None*, *session=None*)

Return a FormSet class for the given sqlalchemy model class.

django_sorcery.formsets.inline module

InlineFormSet for sqlalchemy.

class django_sorcery.formsets.inline.**BaseInlineFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=False*, *prefix=None*, *queryset=None*, ***kwargs*)

Bases: *django_sorcery.formsets.base.BaseModelFormSet*

A formset for child objects related to a parent.

classmethod get_default_prefix()

initial_form_count()

Return the number of forms that are required in this FormSet.

save(*flush=False*, ***kwargs*)

Save model instances for every form, adding and changing instances as necessary, and return the list of instances.

django_sorcery.formsets.inline.**inlineformset_factory**(*parent_model=None*, *model=None*, *relation=None*, *form=<class 'django_sorcery.forms.ModelForm'>*, *formset=<class 'django_sorcery.formsets.inline.BaseInlineFormSet'>*, *fk_name=None*, *fields=None*, *exclude=None*, *extra=3*, *can_order=False*, *can_delete=True*, *max_num=None*, *formfield_callback=None*, *widgets=None*, *validate_max=False*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *min_num=None*, *validate_min=False*, *field_classes=None*, *session=None*)

Return an `InlineFormSet` for the given kwargs.

`fk_name` must be provided if `model` has more than one `ForeignKey` to `parent_model`.

django_sorcery.management package

Subpackages

django_sorcery.management.commands package

Submodules

django_sorcery.management.commands.sorcery module

Sorcery command namespace.

class django_sorcery.management.commands.sorcery.**Command**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: *django_sorcery.management.base.NamespacedCommand*

Namespaced commands for sorcery.

class Meta

Bases: object

namespace = 'sorcery'

createall

alias of *django_sorcery.management.commands.sorcery_createall.CreateAll*

current

alias of *django_sorcery.management.commands.sorcery_current.Current*

downgrade

alias of *django_sorcery.management.commands.sorcery_downgrade.Downgrade*

dropall

alias of *django_sorcery.management.commands.sorcery_dropall.DropAll*

heads

alias of *django_sorcery.management.commands.sorcery_heads.ShowHeads*

help = 'django-sorcery management commands'

history

alias of *django_sorcery.management.commands.sorcery_history.History*

revision

alias of *django_sorcery.management.commands.sorcery_revision.Revision*

stamp

alias of *django_sorcery.management.commands.sorcery_stamp.Stamp*

upgrade

alias of *django_sorcery.management.commands.sorcery_upgrade.Upgrade*

django_sorcery.management.commands.sorcery_createall module

CreateAll command.

django_sorcery.management.commands.sorcery_createall.**Command**

alias of *django_sorcery.management.commands.sorcery_createall.CreateAll*

class django_sorcery.management.commands.sorcery_createall.**CreateAll**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: django.core.management.base.BaseCommand

Creates db schema using metadata.create_all.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Creates SQLAlchemy database schemas'

django_sorcery.management.commands.sorcery_current module

Current command.

django_sorcery.management.commands.sorcery_current.**Command**
alias of *django_sorcery.management.commands.sorcery_current.Current*

class django_sorcery.management.commands.sorcery_current.**Current**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: *django_sorcery.management.alembic.AlembicCommand*

Shows current db revisions.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

display_version(*rev*, *context*, *verbose=False*, *appconfig=None*)
Displays the alembic revision.

handle(*app_label=None*, *verbosity=0*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Show current db revisions'

django_sorcery.management.commands.sorcery_downgrade module

Downgrade command.

django_sorcery.management.commands.sorcery_downgrade.**Command**
alias of *django_sorcery.management.commands.sorcery_downgrade.Downgrade*

class django_sorcery.management.commands.sorcery_downgrade.**Downgrade**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: *django_sorcery.management.alembic.AlembicCommand*

Apply downgrade migration revisions.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

downgrade(*rev*, *context*, *appconfig*, *revision*)
Executes alembic downgrade revisions to the given revision.

handle(*app_label=None*, *revision=None*, *sql=False*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Apply migration revisions'

django_sorcery.management.commands.sorcery_dropall module

DropAll command.

django_sorcery.management.commands.sorcery_dropall.**Command**
alias of *django_sorcery.management.commands.sorcery_dropall.DropAll*

class django_sorcery.management.commands.sorcery_dropall.**DropAll**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: django.core.management.base.BaseCommand

Drops database schemas using metadata.drop_all.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Drops SQLAlchemy database schemas'

django_sorcery.management.commands.sorcery_heads module

Heads command.

django_sorcery.management.commands.sorcery_heads.**Command**
alias of *django_sorcery.management.commands.sorcery_heads.ShowHeads*

class django_sorcery.management.commands.sorcery_heads.**ShowHeads**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: *django_sorcery.management.alembic.AlembicCommand*

Display alembic revision heads.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

handle(*app_label=None*, *verbosity=0*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Display revision heads'

django_sorcery.management.commands.sorcery_history module

History command.

django_sorcery.management.commands.sorcery_history.**Command**
alias of *django_sorcery.management.commands.sorcery_history.History*

class django_sorcery.management.commands.sorcery_history.**History**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: *django_sorcery.management.alembic.AlembicCommand*

Display alembic revisions.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

handle(*app_label=None*, *rev_range=None*, *verbosity=0*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Display alembic revisions'

print_history(*appconfigs*, *verbose*, *base*, *head*)
Prints alembic revision history.

django_sorcery.management.commands.sorcery_revision module

Revision command.

django_sorcery.management.commands.sorcery_revision.**Command**
alias of *django_sorcery.management.commands.sorcery_revision.Revision*

class django_sorcery.management.commands.sorcery_revision.**Revision**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)
Bases: *django_sorcery.management.alembic.AlembicCommand*

Creates an alembic migration revision.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

generate_migration(*rev*, *context*, *appconfig=None*)
Generate alembic migration.

handle(*app_label*, *message=None*, *head=None*, *splice=None*, *branch_label=None*, *depends_on=None*, *rev_id=None*, *autogenerate=None*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Create a migration revision'

django_sorcery.management.commands.sorcery_stamp module

Stamp command.

django_sorcery.management.commands.sorcery_stamp.**Command**
alias of *django_sorcery.management.commands.sorcery_stamp.Stamp*

class django_sorcery.management.commands.sorcery_stamp.**Stamp**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)
Bases: *django_sorcery.management.alembic.AlembicCommand*

Stamp the revision table with migration revisions, doesn't run any migrations.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

handle(*app_label=None*, *revision=None*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = "Stamp the revision table with migration revisions, doesn't run any migrations"

stamp(*rev*, *context*, *appconfig*, *revision*)
Stamp the alembic revision.

django_sorcery.management.commands.sorcery_upgrade module

Upgrade command.

django_sorcery.management.commands.sorcery_upgrade.**Command**
alias of *django_sorcery.management.commands.sorcery_upgrade.Upgrade*

class django_sorcery.management.commands.sorcery_upgrade.**Upgrade**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)
Bases: *django_sorcery.management.alembic.AlembicCommand*

Apply upgrade migration revisions.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

handle(*app_label=None*, *revision=None*, *sql=False*, ***kwargs*)
The actual logic of the command. Subclasses must implement this method.

help = 'Apply migration revisions'

upgrade(*rev*, *context*, *appconfig*, *revision*)
Executes alembic upgrade revisions to the given revision.

Submodules

django_sorcery.management.alembic module

Alembic Django command things.

class django_sorcery.management.alembic.**AlembicAppConfig**(*name*, *config*, *script*, *db*, *app*, *version_path*, *tables*)
Bases: tuple

app
Alias for field number 4

config
Alias for field number 1

db
Alias for field number 3

name
Alias for field number 0

script
Alias for field number 2

tables
Alias for field number 6

version_path
Alias for field number 5

class django_sorcery.management.alembic.**AlembicCommand**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: `django.core.management.base.BaseCommand`

Base alembic django command.

get_app_config(*app*, *db*)
Return alembic config for an app.

get_app_version_path(*app*)
Returns the default migration directory location of al app.

get_common_config(*context*)
Common alembic configuration.

get_config_script(*config*)
Returns the alembic script directory for the config.

lookup_app(*app_label*)
Looks up an app's alembic config.

run_env(*context*, *appconfig*)
Executes an alembic context, just like the env.py file of alembic.

run_migrations_offline(*context*, *appconfig*)
Executes an offline alembic context.

run_migrations_online(*context*, *appconfig*)
Executes an online alembic context.

sorcery_apps
All sorcery apps and their alembic configs.

django_sorcery.management.base module

Namespaced Django command.

class django_sorcery.management.base.**NamespacedCommand**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Bases: `django.core.management.base.BaseCommand`

Namespaced django command implementation.

commands
Returns the subcommands in the namespace.

create_parser(*prog_name*, *subcommand*)
Create and return the `ArgumentParser` which will be used to parse the arguments to this command.

print_help(*prog_name*, *subcommand*)
Print the help message for this command, derived from `self.usage()`.

run_command_from_argv(*command*, *argv*)
Runs the subcommand with namespace adjusted argv.

run_from_argv(*argv*)
Set up any environment changes requested (e.g., Python path and Django settings), then run this command. If the command raises a `CommandError`, intercept it and print it sensibly to stderr. If the `--traceback` option is present or the raised `Exception` is not `CommandError`, raise it.

django_sorcery.validators package

Submodules

django_sorcery.validators.base module

Validators.

class django_sorcery.validators.base.**ValidateCantRemove**(*field*, *message=None*, *code=None*)

Bases: object

Validate that for data cannot be removed for given field.

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.String())

    validators = [
        ValidateCantRemove('foo'),
    ]
```

code = 'remove'

message = 'Cannot remove existing data.'

class django_sorcery.validators.base.**ValidateEmptyWhen**(*field*, *predicate*, *message=None*, *code=None*)

Bases: object

Validator for checking a field is empty when predicate is True.

Useful to conditionally enforce a field is not provided depending on related field

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.String())
    bar = db.Column(db.String())

    validators = [
        # do not allow to set bar unless foo is present
        ValidateEmptyWhen('bar', lambda m: not m.foo),
    ]
```

allow_empty = True

code = 'empty'

message = 'Cannot provide a value to this field.'

class django_sorcery.validators.base.**ValidateNotEmptyWhen**(*field*, *predicate*, *message=None*, *code=None*)

Bases: *django_sorcery.validators.base.ValidateEmptyWhen*

Validator for checking a field is provided when predicate is True.

Useful to conditionally enforce a field is provided depending on related field

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.String())
    bar = db.Column(db.String())

    validators = [
        # enforce bar is provided when foo is provided
        ValidateNotEmptyWhen('bar', lambda m: m.foo),
    ]
```

allow_empty = False

code = 'not_empty'

message = 'This field is required.'

class django_sorcery.validators.base.**ValidateOnlyOneOf**(*fields*, *required=True*, *message=None*, *code=None*)

Bases: object

Validate that only one of given fields is provided.

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.String())
    bar = db.Column(db.String())

    validators = [
        # enforce only either foo or bar are provided
        ValidateOnlyOneOf(['foo', 'bar']),
    ]
```

code = 'exclusive'

message = 'Only one of %(fields)s is allowed.'

class django_sorcery.validators.base.**ValidateTogetherModelFields**(*fields*, *message=None*, *code=None*)

Bases: object

Validator for checking that multiple model fields are always saved together.

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.Integer())
    bar = db.Column(db.Integer())

    validators = [
        ValidateTogetherModelFields(["foo", "bar"]),
    ]
```

code = 'required'

message = 'All %(fields)s are required.'

class django_sorcery.validators.base.**ValidateUnique**(*session*, **args*, ***kwargs*)

Bases: object

Validator for checking uniqueness of arbitrary list of attributes on a model.

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.Integer())
    bar = db.Column(db.Integer())
    name = db.Column(db.Integer())

    validators = [
        ValidateUnique(db, "name"),       # checks for name uniqueness
        ValidateUnique(db, "foo", "bar"),  # checks for foo and bar combination␣
↪uniqueness
    ]
```

code = 'required'

message = '%(fields)s must make a unique set.'

class django_sorcery.validators.base.**ValidateValue**(*field*, *predicate*, *message=None*, *code=None*)

Bases: object

Validator for checking correctness of a value by using a predicate callable.

Useful when other multiple fields need to be consulted to check if particular field is valid.

For example:

```
class MyModel(db.Model):
    foo = db.Column(db.String())
    bar = db.Column(db.String())

    validators = [
        # allow only "bar" value when model.foo == "foo"
        ValidateValue('bar', lambda m: m.foo != 'foo' or b.bar == 'bar'),
    ]
```

code = 'invalid'

message = 'Please enter valid value.'

negated = False

django_sorcery.validators.runner module

Validation runner.

class django_sorcery.validators.runner.**ValidationRunner**(*name=None*, *validators=None*, *errors=None*)

Bases: object

Helper class that can execute a bunch of validators on a given object.

is_valid(*value*, *raise_exception=False*)

Runs all validators on given value and collect validation errors.

django_sorcery.views package

Submodules

django_sorcery.views.base module

Base model view things with sqlalchemy.

class django_sorcery.views.base.**BaseMultipleObjectMixin**
Bases: *django_sorcery.views.base.SQLAlchemyMixin*

Provides sqlalchemy support for list views.

allow_empty = True

get_allow_empty()
Return `True` if the view should display empty lists and `False` if a 404 should be raised instead.

get_ordering()
Return the field or fields to use for ordering the queryset.

get_paginate_by(*queryset*)
Get the number of items to paginate by, or `None` for no pagination.

get_paginate_orphans()
Return the maximum number of orphans extend the last page by when paginating.

get_paginator(*queryset*, *per_page*, *orphans=0*, *allow_empty_first_page=True*, ***kwargs*)
Return an instance of the paginator for this view.

get_queryset()
Return the list of items for this view.

The return value must be an iterable and may be an instance of *QuerySet* in which case *QuerySet* specific behavior will be enabled.

ordering = None

page_kwarg = 'page'

paginate_by = None

paginate_orphans = 0

paginate_queryset(*queryset*, *page_size*)
Paginate queryset.

paginator_class
alias of `django.core.paginator.Paginator`

class django_sorcery.views.base.**BaseSingleObjectMixin**
Bases: *django_sorcery.views.base.SQLAlchemyMixin*

Provides sqlalchemy support for detail views.

get_object(*queryset=None*)
Return the object the view is displaying.

Require *self.queryset* and the primary key attributes or the slug attributes in the URLconf. Subclasses can override this to return any object

get_slug_field()
Get the name of a slug field to be used to look up by slug.

object = None

query_pkg_and_slug = False

slug_field = 'slug'

slug_url_kwarg = 'slug'

class django_sorcery.views.base.**SQLAlchemyMixin**
Bases: django.views.generic.base.ContextMixin

Provides sqlalchemy model view support like query, model, session, etc..

context_object_name = None

classmethod get_model()
Returns the model class.

get_model_template_name()
Returns the base template path.

get_query_options()
Returns sqlalchemy query options.

get_queryset()
Return the *QuerySet* that will be used to look up the object.

This method is called by the default implementation of get_object() and may not be called if get_object() is overridden.

get_session()
Returns the sqlalchemy session.

model = None

query_options = None

queryset = None

session = None

django_sorcery.views.detail module

Django detail view things for sqlalchemy.

class django_sorcery.views.detail.**BaseDetailView**(***kwargs*)
Bases: *django_sorcery.views.detail.SingleObjectMixin*, django.views.generic.base.View

A base view for displaying a single object.

get(*request*, **args*, ***kwargs*)
Handles GET on detail view.

class django_sorcery.views.detail.**DetailView**(***kwargs*)
Bases: *django_sorcery.views.detail.SingleObjectTemplateResponseMixin*, *django_sorcery.views.detail.BaseDetailView*

Render a "detail" view of an object.

By default this is a model instance looked up from *self.queryset*, but the view will support display of *any* object by overriding *self.get_object()*.

class django_sorcery.views.detail.**SingleObjectMixin**
Bases: *django_sorcery.views.base.BaseSingleObjectMixin*

Provide the ability to retrieve a single object for further manipulation.

get_context_data(***kwargs*)
Insert the single object into the context dict.

get_context_object_name(*obj*)
Get the name to use for the object.

class django_sorcery.views.detail.**SingleObjectTemplateResponseMixin**
Bases: django.views.generic.base.TemplateResponseMixin

Provide the ability to retrieve template names.

get_template_names()
Returns template names for detail view.

template_name_field = None

template_name_suffix = '_detail'

django_sorcery.views.edit module

Django edit view things for sqlalchemy.

class django_sorcery.views.edit.**BaseCreateView**(***kwargs*)
Bases: *django_sorcery.views.edit.ModelFormMixin*, *django_sorcery.views.edit.ProcessFormView*

Base view for creating a new object instance.

Using this base class requires subclassing to provide a response mixin.

get(*request*, **args*, ***kwargs*)
Handle GET requests: instantiate a blank version of the form.

post(*request*, **args*, ***kwargs*)
Handle POST requests: instantiate a form instance with the passed POST variables and then check if it's valid.

class django_sorcery.views.edit.**BaseDeleteView**(***kwargs*)
Bases: *django_sorcery.views.edit.DeletionMixin*, *django_sorcery.views.detail.BaseDetailView*

Base view for deleting an object.

Using this base class requires subclassing to provide a response mixin.

class django_sorcery.views.edit.**BaseFormView**(***kwargs*)
Bases: django.views.generic.edit.FormMixin, *django_sorcery.views.edit.ProcessFormView*

A base view for displaying a form.

class django_sorcery.views.edit.**BaseUpdateView**(***kwargs*)
Bases: *django_sorcery.views.edit.ModelFormMixin*, *django_sorcery.views.edit.ProcessFormView*

Base view for updating an existing object.

Using this base class requires subclassing to provide a response mixin.

get(*request*, **args*, ***kwargs*)
Handle GET requests: instantiate a blank version of the form.

post(*request*, **args*, ***kwargs*)
Handle POST requests: instantiate a form instance with the passed POST variables and then check if it's valid.

class django_sorcery.views.edit.**CreateView**(***kwargs*)
Bases: *django_sorcery.views.detail.SingleObjectTemplateResponseMixin*, *django_sorcery.views.edit.BaseCreateView*

View for creating a new object, with a response rendered by a template.

template_name_suffix = '_form'

class django_sorcery.views.edit.**DeleteView**(***kwargs*)
Bases: *django_sorcery.views.detail.SingleObjectTemplateResponseMixin*, *django_sorcery.views.edit.BaseDeleteView*

View for deleting an object retrieved with self.get_object(), with a response rendered by a template.

template_name_suffix = '_confirm_delete'

class django_sorcery.views.edit.**DeletionMixin**
Bases: object

Provide the ability to delete objects.

delete(*request*, **args*, ***kwargs*)
Call the delete() method on the fetched object and then redirect to the success URL.

get_success_url()
Returns the URL to redirect to after processing deletion.

post(*request*, **args*, ***kwargs*)
Handle POST request on deleve view.

success_url = None

class django_sorcery.views.edit.**FormView**(***kwargs*)
Bases: django.views.generic.base.TemplateResponseMixin, *django_sorcery.views.edit.BaseFormView*

A view for displaying a form and rendering a template response.

class django_sorcery.views.edit.**ModelFormMixin**
Bases: django.views.generic.edit.FormMixin, *django_sorcery.views.detail.SingleObjectMixin*

Provide a way to show and handle a ModelForm in a request.

fields = None

form_valid(*form*)
If the form is valid, redirect to the supplied URL.

get_form_class()
Return the form class to use.

get_form_kwargs()
Return the keyword arguments for instantiating the form.

get_success_url()
Return the URL to redirect to after processing a valid form.

class django_sorcery.views.edit.**ProcessFormView**(***kwargs*)
Bases: django.views.generic.base.View

Render a form on GET and processes it on POST.

get(*request*, **args*, ***kwargs*)
Handle GET requests: instantiate a blank version of the form.

post(*request*, **args*, ***kwargs*)
Handle POST requests: instantiate a form instance with the passed POST variables and then check if it's valid.

put(*request*, **args*, ***kwargs*)
Handle POST requests: instantiate a form instance with the passed POST variables and then check if it's valid.

class django_sorcery.views.edit.**UpdateView**(***kwargs*)
Bases: *django_sorcery.views.detail.SingleObjectTemplateResponseMixin*, *django_sorcery.views.edit.BaseUpdateView*

View for updating an object, with a response rendered by a template.

template_name_suffix = '_form'

django_sorcery.views.list module

Django list view things for sqlalchemy.

class django_sorcery.views.list.**BaseListView**(***kwargs*)
Bases: *django_sorcery.views.list.MultipleObjectMixin*, django.views.generic.base.View

A base view for displaying a list of objects.

get(*request*, **args*, ***kwargs*)
Handle GET request on list view.

class django_sorcery.views.list.**ListView**(***kwargs*)
Bases: *django_sorcery.views.list.MultipleObjectTemplateResponseMixin*, *django_sorcery.views.list.BaseListView*

Render some list of objects, set by *self.model* or *self.queryset*.

self.queryset can actually be any iterable of items, not just a queryset.

class django_sorcery.views.list.**MultipleObjectMixin**
Bases: *django_sorcery.views.base.BaseMultipleObjectMixin*

A mixin for views manipulating multiple objects.

context_object_name = 'object_list'

get_context_data(*object_list=None*, ***kwargs*)
Get the context for this view.

get_context_object_name(*object_list*)
Get the name to use for the object.

class django_sorcery.views.list.**MultipleObjectTemplateResponseMixin**
Bases: django.views.generic.base.TemplateResponseMixin

Mixin for responding with a template and list of objects.

get_template_names()
Return a list of template names to be used for the request.

Must return a list. May not be called if render_to_response is overridden.

template_name_suffix = '_list'

django_sorcery.viewsets package

class django_sorcery.viewsets.**GenericViewSet**(***kwargs*)

Bases: django.views.generic.base.TemplateResponseMixin, django.views.generic.base.View

Base class for all sqlalchemy model generic viewsets.

classmethod as_view(*actions=None*, ***initkwargs*)

Main entry point for a request-response process.

classmethod get_extra_actions()

Get the methods that are marked as an extra ViewSet @*action*.

get_template_names()

Return a list of template names to be used for the request. Must return a list. May not be called if render_to_response() is overridden.

class django_sorcery.viewsets.**ModelViewSet**(***kwargs*)

Bases: *django_sorcery.viewsets.mixins.CreateModelMixin*, *django_sorcery.viewsets.mixins.UpdateModelMixin*, *django_sorcery.viewsets.mixins.DeleteModelMixin*, *django_sorcery.viewsets.base.ReadOnlyModelViewSet*

A viewset that provides default *new()*, *create()*, *retrieve()*, *edit()*, update()‘, *confirm_destroy())*, *destroy()* and *list()* actions.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/	list	<resource name>-list
POST	/	create	<resource name>-list
GET	/new/	new	<resource name>-new
GET	/<pk>/	retrieve	<resource name>-detail
POST	/<pk>/	update	<resource name>-detail
PUT	/<pk>/	update	<resource name>-detail
PATCH	/<pk>/	update	<resource name>-detail
DELETE	/<pk>/	destroy	<resource name>-detail
GET	/<pk>/edit/	edit	<resource name>-edit
GET	/<pk>/delete/	confirm_destoy	<resource name>-delete
POST	/<pk>/delete/	destroy	<resource name>-delete

class django_sorcery.viewsets.**ReadOnlyModelViewSet**(***kwargs*)

Bases: *django_sorcery.viewsets.mixins.ListModelMixin*, *django_sorcery.viewsets.mixins.RetrieveModelMixin*, *django_sorcery.viewsets.base.GenericViewSet*

A viewset that provides default *list()* and *retrieve()* actions.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/	list	<resource name>-list
GET	/<pk>/	retrieve	<resource name>-detail

class django_sorcery.viewsets.**CreateModelMixin**

Bases: *django_sorcery.viewsets.mixins.ModelFormMixin*

A mixin for supporting creating objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
POST	/	create	<resource name>-list
GET	/new/	new	<resource name>-new

create(*request*, **args*, ***kwargs*)
Create action for creating an object.

get_create_context_data(***kwargs*)
Returns new context data for template rendering.

new(*request*, **args*, ***kwargs*)
New action for displaying a form for creating an object.

class django_sorcery.viewsets.**DeleteModelMixin**
Bases: *django_sorcery.viewsets.mixins.RetrieveModelMixin*

A mixin for supporting deleting objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/<pk>/delete/	confirm_destoy	<resource name>-delete
POST	/<pk>/delete/	destroy	<resource name>-delete
DELETE	/<pk>/	destroy	<resource name>-detail

confirm_destroy(*request*, **args*, ***kwargs*)
Confirm_destory action for displaying deletion confirmation for an object.

destroy(*request*, **args*, ***kwargs*)
Destroy action for deletion of an object.

destroy_success_url = None

get_destroy_context_data(***kwargs*)
Returns destory context data for rendering deletion confirmation page.

get_destroy_success_url()
Return the url to redirect to after successful deletion of an object.

perform_destoy(*obj*)
Performs the deletion operation.

class django_sorcery.viewsets.**ListModelMixin**
Bases: *django_sorcery.views.base.BaseMultipleObjectMixin*

A mixin for displaying a list of objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/	list	<resource name>-list

get_list_context_data(***kwargs*)
Returns context data for list action.

get_list_context_object_name(*object_list*)
Get the name to use for the object.

list(*request*, **args*, ***kwargs*)
List action for displaying a list of objects.

class django_sorcery.viewsets.**ModelFormMixin**
Bases: `django.views.generic.edit.FormMixin`, *django_sorcery.viewsets.mixins.RetrieveModelMixin*

Common mixin for handling sqlalchemy model forms in viewsets.

fields = None

form_class = None

form_invalid(*form*)
Handles invalid form.

form_valid(*form*)
Processes a valid form.

get_form_class()
Returns the form class to be used.

get_form_context_data(***kwargs*)
Return form context data for template rendering.

get_form_kwargs()
Returns the keyword arguments for instantiating the form.

get_success_url()
Return the URL to redirect to after processing a valid form.

process_form(*form*)
Checks if form is valid and processes accordingly.

success_url = None

class django_sorcery.viewsets.**RetrieveModelMixin**
Bases: *django_sorcery.views.base.BaseSingleObjectMixin*

A mixin for displaying a single object.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/<pk>/	retrieve	<resource name>-detail

get_detail_context_data(***kwargs*)
Returns detail context data for template rendering.

get_detail_context_object_name(*obj*)
Get the name to use for the object.

get_url_kwargs(*obj*)

retrieve(*request*, **args*, ***kwargs*)
List action for displaying a single object.

class django_sorcery.viewsets.**UpdateModelMixin**
Bases: *django_sorcery.viewsets.mixins.ModelFormMixin*

A mixin for supporting updating objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/<pk>/edit/	edit	<resource name>-edit
POST	/<pk>/	update	<resource name>-detail
PUT	/<pk>/	update	<resource name>-detail
PATCH	/<pk>/	update	<resource name>-detail

edit(*request*, **args*, ***kwargs*)
Edit action for displaying a form for editing an object.

get_update_context_data(***kwargs*)
Returns edit context data for template rendering.

update(*request*, **args*, ***kwargs*)
Update action for updating an object.

Submodules

django_sorcery.viewsets.base module

Django REST Framework like model viewsets.

class django_sorcery.viewsets.base.**GenericViewSet**(***kwargs*)
Bases: `django.views.generic.base.TemplateResponseMixin`, `django.views.generic.base.View`

Base class for all sqlalchemy model generic viewsets.

classmethod as_view(*actions=None*, ***initkwargs*)
Main entry point for a request-response process.

classmethod get_extra_actions()
Get the methods that are marked as an extra ViewSet *@action*.

get_template_names()
Return a list of template names to be used for the request. Must return a list. May not be called if render_to_response() is overridden.

class django_sorcery.viewsets.base.**ModelViewSet**(***kwargs*)
Bases: *django_sorcery.viewsets.mixins.CreateModelMixin*, *django_sorcery.viewsets.mixins.UpdateModelMixin*, *django_sorcery.viewsets.mixins.DeleteModelMixin*, *django_sorcery.viewsets.base.ReadOnlyModelViewSet*

A viewset that provides default *new()*, *create()*, *retrieve()*, *edit()*, update()‘, *confirm_destroy())*, *destroy()* and *list()* actions.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/	list	<resource name>-list
POST	/	create	<resource name>-list
GET	/new/	new	<resource name>-new
GET	/<pk>/	retrieve	<resource name>-detail
POST	/<pk>/	update	<resource name>-detail
PUT	/<pk>/	update	<resource name>-detail
PATCH	/<pk>/	update	<resource name>-detail
DELETE	/<pk>/	destroy	<resource name>-detail
GET	/<pk>/edit/	edit	<resource name>-edit
GET	/<pk>/delete/	confirm_destoy	<resource name>-delete
POST	/<pk>/delete/	destroy	<resource name>-delete

class django_sorcery.viewsets.base.**ReadOnlyModelViewSet**(***kwargs*)

Bases: *django_sorcery.viewsets.mixins.ListModelMixin*, *django_sorcery.viewsets.mixins.RetrieveModelMixin*, *django_sorcery.viewsets.base.GenericViewSet*

A viewset that provides default *list()* and *retrieve()* actions.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/	list	<resource name>-list
GET	/<pk>/	retrieve	<resource name>-detail

django_sorcery.viewsets.mixins module

Django REST Framework like viewset mixins for common model sqlalchemy actions.

class django_sorcery.viewsets.mixins.**CreateModelMixin**

Bases: *django_sorcery.viewsets.mixins.ModelFormMixin*

A mixin for supporting creating objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
POST	/	create	<resource name>-list
GET	/new/	new	<resource name>-new

create(*request*, **args*, ***kwargs*)

Create action for creating an object.

get_create_context_data(***kwargs*)

Returns new context data for template rendering.

new(*request*, **args*, ***kwargs*)

New action for displaying a form for creating an object.

class django_sorcery.viewsets.mixins.**DeleteModelMixin**

Bases: *django_sorcery.viewsets.mixins.RetrieveModelMixin*

A mixin for supporting deleting objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/<pk>/delete/	confirm_destoy	<resource name>-delete
POST	/<pk>/delete/	destroy	<resource name>-delete
DELETE	/<pk>/	destroy	<resource name>-detail

confirm_destroy(*request*, **args*, ***kwargs*)
Confirm_destory action for displaying deletion confirmation for an object.

destroy(*request*, **args*, ***kwargs*)
Destroy action for deletion of an object.

destroy_success_url = None

get_destroy_context_data(***kwargs*)
Returns destory context data for rendering deletion confirmation page.

get_destroy_success_url()
Return the url to redirect to after successful deletion of an object.

perform_destoy(*obj*)
Performs the deletion operation.

class django_sorcery.viewsets.mixins.**ListModelMixin**
Bases: *django_sorcery.views.base.BaseMultipleObjectMixin*

A mixin for displaying a list of objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/	list	<resource name>-list

get_list_context_data(***kwargs*)
Returns context data for list action.

get_list_context_object_name(*object_list*)
Get the name to use for the object.

list(*request*, **args*, ***kwargs*)
List action for displaying a list of objects.

class django_sorcery.viewsets.mixins.**ModelFormMixin**
Bases: django.views.generic.edit.FormMixin, *django_sorcery.viewsets.mixins.RetrieveModelMixin*

Common mixin for handling sqlalchemy model forms in viewsets.

fields = None

form_class = None

form_invalid(*form*)
Handles invalid form.

form_valid(*form*)
Processes a valid form.

get_form_class()
Returns the form class to be used.

get_form_context_data(***kwargs*)
Return form context data for template rendering.

get_form_kwargs()
Returns the keyword arguments for instantiating the form.

get_success_url()
Return the URL to redirect to after processing a valid form.

process_form(*form*)
Checks if form is valid and processes accordingly.

success_url = None

class django_sorcery.viewsets.mixins.**RetrieveModelMixin**
Bases: *django_sorcery.views.base.BaseSingleObjectMixin*

A mixin for displaying a single object.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/<pk>/	retrieve	<resource name>-detail

get_detail_context_data(***kwargs*)
Returns detail context data for template rendering.

get_detail_context_object_name(*obj*)
Get the name to use for the object.

get_url_kwargs(*obj*)

retrieve(*request*, **args*, ***kwargs*)
List action for displaying a single object.

class django_sorcery.viewsets.mixins.**UpdateModelMixin**
Bases: *django_sorcery.viewsets.mixins.ModelFormMixin*

A mixin for supporting updating objects.

When used with router, it will map the following operations to actions on the viewset

Method	Path	Action	Route Name
GET	/<pk>/edit/	edit	<resource name>-edit
POST	/<pk>/	update	<resource name>-detail
PUT	/<pk>/	update	<resource name>-detail
PATCH	/<pk>/	update	<resource name>-detail

edit(*request*, **args*, ***kwargs*)
Edit action for displaying a form for editing an object.

get_update_context_data(***kwargs*)
Returns edit context data for template rendering.

update(*request*, **args*, ***kwargs*)
Update action for updating an object.

Submodules

django_sorcery.exceptions module

Exceptions.

exception django_sorcery.exceptions.**NestedValidationError**(*message*, *code=None*, *params=None*)

Bases: django.core.exceptions.ValidationError

Django Validation error except which allows nested errors.

Useful for validating composite objects.

For example:

```
raise NestedValidationError({
    "field": ["error"],
    "composite": {
        "field": ["error"],
    }
})
```

update_error_dict(*error_dict*)

django_sorcery.fields module

Field mapping from SQLAlchemy type's to form fields.

class django_sorcery.fields.**EnumField**(*enum_class=None*, *choices=None*, ***kwargs*)

Bases: django.forms.fields.ChoiceField

Form field for using an Enum as choices.

bound_data(*data*, *initial*)

Return the value that should be shown for this field on render of a bound form, given the submitted POST data for the field and the initial data, if any.

For most fields, this will simply be data; FileFields need to handle it a bit differently.

empty_value = None

prepare_value(*value*)

to_python(*value*)

Return a string.

valid_value(*value*)

Check to see if the provided value is a valid choice.

class django_sorcery.fields.**ModelChoiceField**(*model*, *session*, *empty_label='———'*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=''*, *to_field_name=None*, *limit_choices_to=None*, ***kwargs*)

Bases: django.forms.fields.ChoiceField

A ChoiceField whose choices are a sqlalchemy model relationship.

choices

default_error_messages = {'invalid_choice': 'Select a valid choice. That choice is no

get_bound_field(*form*, *field_name*)
Return a BoundField instance that will be used when accessing the form field in a template.

get_limit_choices_to()
Returns limit_choices_to for this model.

If it is a callable, invoke it and return the result

get_object(*value*)
Returns model instance.

iterator
alias of *ModelChoiceIterator*

label_from_instance(*obj*)
Returns label from model instance.

prepare_instance_value(*obj*)
Returns primary key from instance.

prepare_value(*obj*)

queryset

to_python(*value*)
Return a string.

validate(*value*)
Validate that the input is in self.choices.

class django_sorcery.fields.**ModelChoiceIterator**(*field*)
Bases: object

Iterator for sqlalchemy query for model choice fields.

choice(*obj*)
Returns choice item for django choice field.

class django_sorcery.fields.**ModelMultipleChoiceField**(**args*, ***kwargs*)
Bases: *django_sorcery.fields.ModelChoiceField*

A ChoiceField whose choices are a sqlalchemy model list relationship.

default_error_messages = {'invalid_choice': 'Select a valid choice. %(value)s is not

hidden_widget
alias of django.forms.widgets.MultipleHiddenInput

prepare_value(*value*)

to_python(*value*)
Return a string.

widget
alias of django.forms.widgets.SelectMultiple

django_sorcery.forms module

Helper functions for creating Form classes from SQLAlchemy models.

class django_sorcery.forms.**BaseModelForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*, *session=None*)

Bases: django.forms.models.BaseModelForm

Base ModelForm for sqlalchemy models.

is_valid(*rollback=True*)

Return True if the form has no errors, or False otherwise.

Will also rollback the session transaction.

model_to_dict()

Returns a dict containing the data in instance suitable for passing as forms initial keyword argument.

save(*flush=True*, ***kwargs*)

Makes form's self.instance model persistent and flushes the session.

save_instance(*instance=None*, *cleaned_data=None*)

Updates form's instance with cleaned data.

update_attribute(*instance*, *name*, *field*, *value*)

Provides hooks for updating form instance's attribute for a field with value.

class django_sorcery.forms.**ModelForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*, *session=None*)

Bases: *django_sorcery.forms.BaseModelForm*

ModelForm base class for sqlalchemy models.

base_fields = {}

declared_fields = {}

media

class django_sorcery.forms.**ModelFormMetaclass**

Bases: django.forms.forms.DeclarativeFieldsMetaclass

ModelForm metaclass for sqlalchemy models.

class django_sorcery.forms.**SQLAModelFormOptions**(*options=None*)

Bases: django.forms.models.ModelFormOptions

Model form options for sqlalchemy.

django_sorcery.forms.**apply_limit_choices_to_form_field**(*formfield*)

Apply limit_choices_to to the formfield's query if needed.

django_sorcery.forms.**fields_for_model**(*model*, *session*, *fields=None*, *exclude=None*, *widgets=None*, *formfield_callback=None*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *field_classes=None*, *apply_limit_choices_to=True*, ***kwargs*)

Returns a dictionary containing form fields for a given model.

django_sorcery.forms.**model_to_dict**(*instance*, *fields=None*, *exclude=None*)
Return a dict containing the data in `instance` suitable for passing as a Form's `initial` keyword argument.

`fields` is an optional list of field names. If provided, return only the named.

`exclude` is an optional list of field names. If provided, exclude the named from the returned dict, even if they are listed in the `fields` argument.

django_sorcery.forms.**modelform_factory**(*model*, *form=<class 'django_sorcery.forms.ModelForm'>*, *formfield_callback=None*, ***kwargs*)
Return a ModelForm class containing form fields for the given model.

django_sorcery.pytest_plugin module

pytest plugins.

django_sorcery.pytest_plugin.**sqlalchemy_profiler**()
pytest fixture for sqlalchemy profiler.

django_sorcery.pytest_plugin.**transact**()
pytest transact fixture for sqlalchemy.

django_sorcery.routers module

Django REST Framework like router for viewsets.

class django_sorcery.routers.**BaseRouter**
Bases: `object`

Base router.

get_default_base_name(*viewset*)
If *base_name* is not specified, attempt to automatically determine it from the viewset.

get_urls()
Return a list of URL patterns, given the registered viewsets.

register(*prefix*, *viewset*, *base_name=None*)
Registers a viewset for route generation.

urls
URL's routed.

class django_sorcery.routers.**DynamicRoute**(*url*, *name*, *detail*, *initkwargs*)
Bases: `tuple`

detail
Alias for field number 2

initkwargs
Alias for field number 3

name
Alias for field number 1

url
Alias for field number 0

class django_sorcery.routers.**Route**(*url*, *mapping*, *name*, *detail*, *initkwargs*)
Bases: `tuple`

detail
Alias for field number 3

initkwargs
Alias for field number 4

mapping
Alias for field number 1

name
Alias for field number 2

url
Alias for field number 0

class django_sorcery.routers.**SimpleRouter**(*trailing_slash=True*)
Bases: *django_sorcery.routers.BaseRouter*

Generates url patterns that map requests to a viewset's action functions.

It will map the following operations to following actions on the viewset:

Method	Path	Action	Route Name
GET	/<resource>/	list	<resource>-list
POST	/<resource>/	create	<resource>-list
GET	/<resource>/new/	new	<resource>-new
GET	/<resource>/<pk>/	retrieve	<resource>-detail
POST	/<resource>/<pk>/	update	<resource>-detail
PUT	/<resource>/<pk>/	update	<resource>-detail
PATCH	/<resource>/<pk>/	update	<resource>-detail
DELETE	/<resource>/<pk>/	destroy	<resource>-detail
GET	/<resource>/<pk>/edit/	edit	<resource>-edit
GET	/<resource>/<pk>/delete/	confirm_destoy	<resource>-delete
POST	/<resource>/<pk>/delete/	destroy	<resource>-delete

get_default_base_name(*viewset*)
If *base_name* is not specified, attempt to automatically determine it from the viewset.

get_lookup_regex(*viewset*, *lookup_prefix=''*)
Given a viewset, return the portion of URL regex that is used to match against a single instance.

Note that lookup_prefix is not used directly inside REST rest_framework itself, but is required in order to nicely support nested router implementations, such as drf-nested-routers.

https://github.com/alanjds/drf-nested-routers

get_method_map(*viewset*, *method_map*)
Given a viewset, and a mapping of http methods to actions, return a new mapping which only includes any mappings that are actually implemented by the viewset.

get_routes(*viewset*)
Augment *self.routes* with any dynamically generated routes.

Returns a list of the Route namedtuple.

get_urls()
Use the registered viewsets to generate a list of URL patterns.

routes = [Route(url='^{prefix}{trailing_slash}$', mapping={'get': 'list', 'post': 'c

django_sorcery.routers.**action**(*methods=None*, *detail=None*, *url_path=None*, *url_name=None*, ***kwargs*)
Mark a ViewSet method as a routable action.

Set the *detail* boolean to determine if this action should apply to instance/detail requests or collection/list requests.

django_sorcery.routers.**escape_curly_brackets**(*url_path*)
Double brackets in regex of url_path for escape string formatting.

django_sorcery.shortcuts module

Some Django like shortcuts that support sqlalchemy models.

django_sorcery.shortcuts.**get_list_or_404**(*klass*, **args*, ***kwargs*)
Use filter() to return a list of objects, or raise a Http404 exception if the count is 0.

klass may be a Model or Query object. All other passed arguments used in filter() and keyword arguments are used in filter_by().

django_sorcery.shortcuts.**get_object_or_404**(*klass*, **args*, ***kwargs*)
Use session.get() to return an object, or raise a Http404 exception if the object does not exist.

klass may be a Model, or Query object. All other passed arguments and keyword arguments are used in the get() query.

django_sorcery.testing module

Testing utilities.

exception django_sorcery.testing.**CommitException**
Bases: BaseException

Raised when a commit happens during testing.

class django_sorcery.testing.**Transact**
Bases: object

Helper context manager for handling transactions during tests. Perfect for testing a single unit of work.

For testing sqlalchemy apps the common pattern is to usually, start a transaction or savepoint, run the tests and once the test is done rollback. Additionally it also prevents commits to the database so that every test gets the same database state.

end()
Rolls back transaction and removes session.

hook()
Hooks commit events to prevent commits.

start()
Starts transaction management and hooks commit events.

stop()
Stops transaction management, rolls back transactions and unhooks commit events.

unhook()
Unhooks commit events.

django_sorcery.utils module

Some common utilities.

django_sorcery.utils.**get_args**(*func*)
Returns the names of the positional arguments for composite model inspection.

django_sorcery.utils.**lower**(*value*)
Convert value to lowercase if possible.

For example:

```
>>> print(lower('HELLO'))
hello
>>> print(lower(5))
5
```

django_sorcery.utils.**make_args**(**args*, ***kwargs*)
Useful for setting table args and mapper args on models and other things.

django_sorcery.utils.**sanitize_separators**(*value*)
Sanitize a value according to the current decimal and thousand separator setting.

Used with form field input.

django_sorcery.utils.**setdefaultattr**(*obj*, *name*, *value*)
setdefault for object attributes.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W