Django Synctool Documentation

Release 1.0.0

Preston Timmons

November 01, 2014

Contents

Synctool is a library for Django to make syncing querysets between databases easy. No more manually dumping or entering data. No more out-of-date fixtures. Just get the data you want, on demand.

CHAPTER 1

Basic usage

Here's an example for syncing the `django.contrib.sites` app.

1. Create an api view

```
# myapp.views

from synctools.routing import Route

route = Route()

@route.app("sites", "sites")
```

2. Add the urls to your project

```
# myproject.urls

from django.conf.urls import include, url
from myapp.views import route

urlpatterns += [
    url("^sync/", include(route.urlpatterns)),
]
```

3. Sync data from the remote endpoint

```
# myclient.py

from synctools.client import Client

client = Client(
    api_url="https://myserver.com/sync/",
    api_token="<token>",
)

if __name__ == "__main__":
    client.sync("sites")
```

The sites app can now be synced locally from a remote data source by calling:

```
python myclient.py
```

1.1 How it works

Under the hood Synctool uses the Django JSON serializer to pass data between servers. Synctool isn't limited to syncing whole applications. It can also sync custom querysets and even download associated images.

CHAPTER 2

Installation

Synctool can be installed from PyPI:

```
pip install django-synctool
```

CHAPTER 3

Contents

3.1 Adding api views

In order to sync data, a remote server must make it available for download. This can be achieved using the `synctool.routing.Route` class. This class takes care of:

- Creating the url pattern
- Serializing the returned data using the Django JSON serializer
- Returning an `application/json` response object
- Adding http basic authentication to the api

3.1.1 Serializing an application

The `route.app` function can be used to serialize a whole application. The returned data is the same as that of the `python manage.py dumpdata` command.

`route.`**`app`**`(`*path*, *label*`)`

Parameters

- **path** – The path of the urlpattern to register.
- **label** – The label of the installed application to serialize.

Example:

```
from synctools.routing import Route

route = Route()

# Sync the ``django.contrib.sites`` app
route.app("sites", "sites")

# Sync an application call myblogapp
route.app("blogs", "myblogapp")
```

Once your urlpatterns are registered, you can open the url in a browser to inspect the data.

3.1.2 Serializing querysets

The `route.queryset` decorator can be used to register any function that returns one or more querysets.

`route.queryset(path)`

Parameters **path** – The path of the urlpattern to register.

Returning a single queryset

Example:

```
from myapp.models import Blog

@route.queryset("blogs")
def blogs():
    return Blog.objects.all()
```

Returning multiple querysets

Multiple querysets can be returned as a list or tuple:

```
from myapp.models import Blog, Post

@route.queryset("blogs")
def blogs():
    return [
        Blog.objects.all(),
        Post.objects.all(),
    ]
```

Filtering and slicing

Querysets can be filtered or sliced. This is useful when you only want to return a subset of a table.

Example:

```
@route.queryset("blogs")
def blogs():
    return [
        Blog.objects.all()[:100],
    ]
```

Accepting arguments

The route argument is a url regular expression. This means views can take arguments from the url.

Example:

```
@route.queryset("blog/(?P<slug>[^/]+)")
def blog(slug):
    return Blog.objects.filter(slug=slug)
```

Modifying querysets

Querysets can be modified before returning them. This can be helpful if you want to exclude certain information in the output.

For example, if `Blog` had a `User` relation, we could return a `Blog` queryset but leave out the user information.

Example:

```
@route.queryset("blogs")
def pickle_blog():
    queryset = Blog.objects.all()
    for blog in queryset:
        blog.user = None
    return queryset
```

Note: This example assumes the user field is nullable.

Order is important

When syncing applications using the `route.app` function, model dependencies are automatically calculated and sorted. When using `route.queryset`, this is not the case. Therefore, you must pay attention to the order in which you return querysets.

For example, assume you had a `Blog` and `Post` model. The `Post` model has a foreign key to `Blog`:

```
@route.queryset("blogs")
def blogs():
    # Good: Post depends on blog existing first. Since blog
    # is serialized first, blog will be saved first.
    return [
        Blog.objects.all(),
        Post.objects.all(),
    ]

@route.queryset("blogs")
def blogs():
    # Bad: The sync client can fail with an IntegrityError because
    # post points to blogs that haven't been saved locally yet.
    return [
        Post.objects.all(),
        Blog.objects.all(),
    ]
```

3.1.3 Authentication

HTTP basic authentication is added to each view created using the `Route` class. By default, the credential is set to `settings.SYNCTOOL_API_TOKEN`. A custom token can be specified as an argument to the `Route` class.

Example:

```
route = Route(api_token="mytoken")
```

A sample call that grants access to this view would be:

```
$ curl https://myserver.com/sync/sites -u mytoken:
```

Note: If you want your queryset information and credentials to remain private, make sure to serve your API over SSL only.

3.1.4 Including urls

The routed url patterns can be included in your project in the same way as another other url pattern:

Example:

```
# myproject.urls

from myapp.views import route

urlpatterns += patterns("",
    url("^sync/", include(route.urlpatterns)),
)
```

3.2 Syncing data from a remote api

This library provides a client for syncing data from a remote url.

Begin by creating a client instance:

```
from synctools.client import Client

client = Client(
    api_url="https://<remote-server.com>/sync/",
    api_token="<mytoken>",
)
```

Now, you can try syncing data from a remote api:

```
client.sync("<path>")
```

3.2.1 Cleaning local data

It's often preferable to delete local data before saving the remote information. The `sync` function will do this if you passing the `clean` argument.

```
client.sync("<path>", clean=True)
```

This step is especially helpful if you've made local edits to your database. If you've made local changes that conflict with the remote queryset, and do not clear your local data, the sync process can fail with an `IntegrityError`.

3.2.2 Downloading images

The `sync` command can be instructed to download remote images in addition to saving the database information. To do this, pass `images` argument.

```
client.sync("<path>", images=True)
```

When images are synced this way, images will be downloaded for every `ImageField` of every model type in the data returned by the remote url. If images already exist locally, the download will be skipped.

3.2.3 Reset sequence

After completing a sync, the `sync_data` command resets the primary key sequence for each application the remote querysets belonged to. Without this step, adding new items to your local database may fail with integrity errors.

If you do not want to reset the sequence for some reason, pass `reset` as `False`.

```
client.sync("<path>", reset=False)
```

3.2.4 Downloading images manually

You can manually initiate the download image process. This is helpful if:

- You already have local data that you want to download images for
- You only want to download images for certain fields of a model

```
client = Client(
    media_url="http://<remote-server.com>/<mediaroot>/",
)

client.images(
    queryset=Post.objects.all(),
    field="hero_image"
)
```

This will download and save an image for each entry in the queryset. If the image entry is empty, or the local image already exists, the download is skipped.

Note: This function assumes you're using file storage in your local environment.

3.3 Creating a command-line interface

If you sync more than a few models, it's nice to wrap that up in a command-line interface. This can be done easily using the Click library.

3.3.1 Example interface

```
# sync.py

import os
os.environ["DJANGO_SETTINGS_MODULE"] = "mysite.settings"

import django
django.setup()

import click
from synctool.client import Client


client = Client(
    api_url="<remote-url>",
    media_url="<media-url>",
)
```

```
@click.group()
def cli():
    """A tool for syncing data."""


@cli.command()
@click.option("--clean/--no-clean", default=False)
@click.option("--images/--no-images", default=False)
def blogs(clean, images):
    """ Sync blogs """
    client.sync("blogs", clean=True)


if __name__ == "__main__":
    cli()
```

Now you can sync data using a command like:

```
python sync.py blogs --clean
```

You can make this yet better by integrating with setuptools.

This would enable you to simplify it to something like:

```
sync blogs --clean
```

Further, if your application is installed in a virtualenv, you can call the command without needing to activate the virtualenv.

3.4 API reference

This provides a reference to the public classes and functions.

3.4.1 The Client class

class `synctools.client.`**`Client`**

The `Client` class provides functions for downloading remote data and images.

Methods

`Client.`**`__init__`**(*api_token=None*, *api_url=None*, *media_url=None*)

Instantiates a `Client` object.

`api_token` is the username used for HTTP basic authentication with the remote api. If this value isn't provided, it defaults to `settings.SYNCTOOL_CLIENT_TOKEN`.

`api_url` is the base url of the remote api to connect with. This would be something like `https://myserver.com/sync/`. This value is prefixed to the url provided to the `Client.sync` function. If this value isn't provided, it defaults to `settings.SYNCTOOL_CLIENT_ENDPOINT`.

`media_url` is the base url from where remote media files are served. This is used if the client is instructed to download images.

`Client.`**`sync`**`(`*url*, *clean=False*, *reset=True*, *images=False*`)`

Syncs data from a remote api.

`url` is the remote url to connect to. This is only the part of the url after `self.api_url`. For example, if the api url is `https://<remote-server>/sync/`, `client.sync("sites")` would connect to `https://<remote-server>/sync/sites`.

`clean` tells the client whether to delete local information before saving the remote data.

`reset` tells the client whether reset the primary key sequence of the application tables after the sync is finished.

`images` tells the client whether to download images for any image fields contained in the synced data.

`Client.`**`images`**`(`*queryset*, *field*`)`

Download remote images for a queryset. Images will be downloaded from the `media_url` client option.

`queryset` is the queryset to download images for, i.e. `Blog.objects.all()`

`field` is the name of the image field to download images for.

3.4.2 The Route class

class `synctools.routing.`**`Route`**

The `Route` class creates views and urls for sync apis.

Methods

`Route.`**`__init__`**`(`*api_token=None*`)`

Instantiates a `Route` object.

`api_token` is the authentication token to require for any clients connecting to this api. If this value isn't provided, it defaults to `settings.SYNCTOOL_API_TOKEN`.

`Route.`**`app`**`(`*path*, *label*`)`

Creates a view to serialize data from a given app label.

Example:

```
route.app("blogs", "myblogapp")
```

`path` is the url regex to serve the view from.

`label` is the installed application label to serialize.

`Route.`**`queryset`**`(`*path*`)`

A decorator factory for views that serialize a given queryset.

Example:

```
@route.queryset("blogs")
def blogs():
    return Blog.objects.all()
```

`path` is the url regex to serve the view from.

Python Module Index