django-remote-submission Documentation

Release 0.3.0

Tanner Hobson

January 10, 2017

Contents:

CHAPTER 1

Testing the Library

There are a few steps for testing the library.

1. Install dependencies
2. Modify your settings
3. Run `make test` or `make test-all`

1.1 Install Dependencies

In order to run the tests, the dependencies need to be installed. To do this, run these commands

```
$ python3 -m virtualenv venv
$ source venv/bin/activate
(venv)$ python3 -m pip install -r requirements_test.txt
```

1.2 Modify Settings

Then copy `.env.base` to `.env` and edit the file. For example, the default `.env.base` file right now is:

```
################
# Testing

# These next variables control the test server used for making sure remote
# execution is working.

# The hostname of the server to connect to
TEST_SERVER_HOSTNAME=

# The SSH port of the server to connect to
TEST_SERVER_PORT=22

# The remote user's username
TEST_REMOTE_USER=

# The remote user's password
TEST_REMOTE_PASSWORD=

# The remote directory to store scripts in
TEST_REMOTE_DIRECTORY=
```

```
# The filename to store the scripts as
TEST_REMOTE_FILENAME=

# The interpreter name that will run the script
TEST_INTERPRETER_NAME=

# The interpreter path that will run the script
TEST_INTERPRETER_PATH=
```

1.3 Run Tests

To run the tests on your current Python version, use the target `test`. To run tests on multiple Python versions, use the target `test-all`.

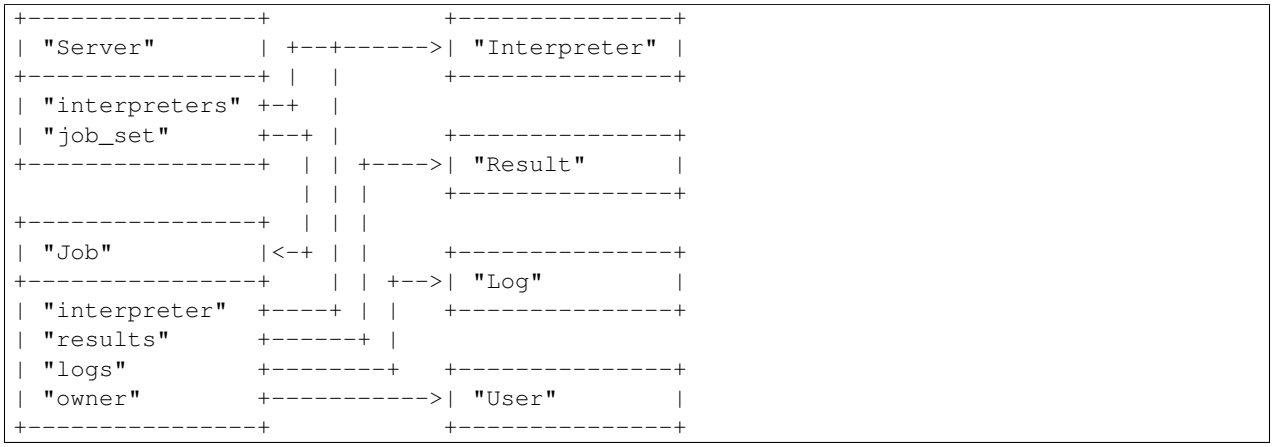
```
(venv)$ make test  # current python version
(venv)$ make test-all  # multiple python versions
```

CHAPTER 2

Modules Reference

2.1 Models

Provide the Django models for interfacing with the job submission tasks.



class `django_remote_submission.models.`**`Server`**(**args*, ***kwargs*)

Encapsulates the remote server identifiers.

```
>>> from django_remote_submission.models import Server
>>> server = Server(
...     title='Remote',
...     hostname='foo.invalid',
...     port=22,
... )
>>> server.interpreters.set([python3])
>>> server
<Server: Remote <foo.invalid:22>>
```

Parameters

- **`id`** (*`AutoField`*) – Id
- **`created`** (*`AutoCreatedField`*) – Created
- **`modified`** (*`AutoLastModifiedField`*) – Modified
- **`title`** (*`CharField`*) – The human-readable name of the server
- **`hostname`** (*`CharField`*) – The hostname used to connect to the server

- **port** (*IntegerField*) – The port to connect to for SSH (usually 22)
- **interpreters** (*ManyToManyField*) – List of interpreters available for this server

__unicode__()
Convert model to string, e.g. "Remote <foo.invalid:22>".

class django_remote_submission.models.**Job**(**args*, ***kwargs*)
Encapsulates the information about a particular job.

```
>>> from django_remote_submission.models import Job
>>> job = Job(
...     title='My Job',
...     program='print("hello world")',
...     remote_directory='/tmp/',
...     remote_filename='foobar.py',
...     owner=user,
...     server=server,
...     interpreter=python3,
... )
>>> job
<Job: My Job>
```

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **title** (*CharField*) – The human-readable name of the job
- **program** (*TextField*) – The actual program to run (starting with a #!)
- **status** (*StatusField*) – The current status of the program
- **remote_directory** (*CharField*) – The directory on the remote host to store the program
- **remote_filename** (*CharField*) – The filename to store the program to (e.g. reduce.py)
- **owner_id** (ForeignKey to User) – The user that owns this job
- **server_id** (ForeignKey to *Server*) – The server that this job will run on
- **interpreter_id** (ForeignKey to *Interpreter*) – The interpreter that this job will run on

clean()
Ensure that the selected interpreter exists on the server.

To use effectively, add this to the django.db.models.signals.pre_save() signal for the *Job* model.

__unicode__()
Convert model to string, e.g. "My Job".

class django_remote_submission.models.**Interpreter**(**args*, ***kwargs*)
Encapsulates the executable and required arguments for each interpreter.

```
>>> from django_remote_submission.models import Interpreter
>>> python3 = Interpreter(
...     name='Python 3',
...     path='/usr/bin/python3',
...     arguments=['-u'],
... )
>>> python3
<Interpreter: Python 3 (/usr/bin/python3)>
```

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **name** (*CharField*) – The human-readable name of the interpreter
- **path** (*CharField*) – The full path of the interpreter path.
- **arguments** (*ListField*) – The arguments used when running the interpreter

__unicode__()
Convert model to string, e.g. `"Python 3 (/usr/bin/python3)"`.

class django_remote_submission.models.**Result**(**args*, ***kwargs*)
Encapsulates a resulting file produced by a job.

```
>>> from django_remote_submission.models import Result
>>> result = Result(
...     remote_filename='1.txt',
...     job=job,
... )
>>> result
<Result: 1.txt <My Job>>
```

Parameters

- **id** (*AutoField*) – Id
- **created** (*AutoCreatedField*) – Created
- **modified** (*AutoLastModifiedField*) – Modified
- **remote_filename** (*TextField*) – The filename on the remote server for this result, relative to the remote directory of the job
- **local_file** (*FileField*) – The filename on the local server for this result
- **job_id** (ForeignKey to *Job*) – The job this result came from

__unicode__()
Convert model to string, e.g. `"1.txt <My Job>"`.

class django_remote_submission.models.**Log**(**args*, ***kwargs*)
Encapsulates a log message printed from a job.

```
>>> from django_remote_submission.models import Log
>>> from datetime import datetime
>>> log = Log(
...     time=datetime(year=2017, month=1, day=2, hour=3, minute=4, second=5),
```

```
...     content='Hello World',
...     stream='stdout',
...     job=job,
... )
>>> log
<Log: 2017-01-02 03:04:05 My Job>
```

Parameters

- **id** (*AutoField*) – Id
- **time** (*AutoCreatedField*) – The time this log was created
- **content** (*TextField*) – The content of this log message
- **stream** (*CharField*) – Output communication channels. Either stdout or stderr.
- **job_id** (ForeignKey to *Job*) – The job this log came from

__unicode__()
Convert model to string, e.g. `"2017-01-02 03:04:05 My Job"`.

django_remote_submission.models.**job_result_path**(*instance*, *filename*)
Produce the path to locally store the job results.

Parameters

- **instance** (Result) – the *Result* instance to produce the path for
- **filename** (*str*) – the original filename

2.2 Tasks

Submit a job to a remote server and handle logging.

This module can be used either with Celery, in which case it will run in a background thread, or as a normal function call, in which case it will block the current execution thread.

class django_remote_submission.tasks.**LogPolicy**
Specify how logging should be done when running a job.

LOG_NONE = 0
Don't log anything from the running job.

LOG_LIVE = 1
Create Log objects immediately when they are received.

LOG_TOTAL = 2
Combine all of stdout and stderr at the end of the job.

django_remote_submission.tasks.**is_matching**(*filename*, *patterns=None*)
Check if a filename matches the list of positive and negative patterns.

Positive patterns are strings like `"1.txt"`, `"[23].txt"`, or `"*.txt"`.

Negative patterns are strings like `"!1.txt"`, `"![23].txt"`, or `"!*.txt"`.

Each pattern is checked in turn, so the list of patterns `["!*.txt", "1.txt"]` will still match `"1.txt"`.

```
>>> from django_remote_submission.tasks import is_matching
>>> is_matching("1.txt", patterns=["1.txt"])
True
>>> is_matching("1.txt", patterns=["[12].txt"])
True
>>> is_matching("1.txt", patterns=["*.txt"])
True
>>> is_matching("1.txt", patterns=["1.txt", "!*.txt"])
False
>>> is_matching("1.txt", patterns=["!*.txt", "[12].txt"])
True
```

class django_remote_submission.tasks.**LogContainer**(*job*, *log_policy*)
Manage logs sent by a job according to the log policy.

```
>>> from django_remote_submission.tasks import LogContainer, LogPolicy
>>> from datetime import datetime
>>> now = datetime(year=2017, month=1, day=2, hour=3, minute=4, second=5)
>>> logs = LogContainer(job, LogPolicy.LOG_LIVE)
>>> logs.write_stdout(now, 'hello world')
>>> Log.objects.get()
<Log: 2017-01-02 03:04:05 My Job>
```

__init__(*job*, *log_policy*)
Instantiate a log container.

Parameters

- **job** (models.Job) – the job these logs are coming from
- **log_policy** (LogPolicy) – the policy to use for logging

class LogLine(*now*, *output*)

now
Alias for field number 0

output
Alias for field number 1

LogContainer.**job = None**
The job that these logs are coming from.

LogContainer.**log_policy = None**
The policy to use when logging.

LogContainer.**write_stdout**(*now*, *output*)
Write some output from a job's stdout stream.

Parameters

- **now** (*datetime.datetime*) – the time this output was produced
- **output** (*str*) – the output that was produced

LogContainer.**write_stderr**(*now*, *output*)
Write some output from a job's stderr stream.

Parameters

- **now** (*datetime.datetime*) – the time this output was produced
- **output** (*str*) – the output that was produced

LogContainer.**flush**()
Flush the stdout and stderr lists to Django models.

If the *log_policy* is *LogPolicy.LOG_TOTAL*, this method will need to be called at the end of the job to ensure all the data gets written out.

There is no penalty for calling this method multiple times, so it can be called at the end of the job regardless of which log policy is used.

django_remote_submission.tasks.**submit_job_to_server**(*job_pk*, *password*, *username=None*, *timeout=None*, *log_policy=1*, *store_results=None*)
Submit a job to the remote server.

This can be used as a Celery task, if the library is installed and running.

Parameters

- **job_pk** (*int*) – the primary key of the models.Job to submit
- **password** (*str*) – the password of the user submitting the job
- **username** (*str*) – the username of the user submitting, if it is different from the owner of the job
- **timeout** (*datetime.timedelta*) – the timeout for running the job
- **log_policy** (LogPolicy) – the policy to use for logging
- **store_results** (*list(str)*) – the patterns to use for the results to store

2.3 Remote

Provides a wrapper around Paramiko to simplify the API.

This module is meant to be a general wrapper so that a LocalWrapper can also be created to run tests in continuous integration services where SSH is not available.

class django_remote_submission.remote.**RemoteWrapper**(*hostname*, *username*, *port=22*)
Wrapper around Paramiko which simplifies the remote connection API.

The goal with this class is to also be able to provide a LocalWrapper which works with the local file system, so that tests can be run on continuous integration servers.

__init__(*hostname*, *username*, *port=22*)
Initialize the wrapper.

Parameters

- **hostname** (*str*) – the hostname of the server to connect to
- **username** (*str*) – the username of the user on the remote server
- **port** (*int*) – the SSH port to connect to

connect(*password=None*, *public_key_filename=None*)
Connect to the remote host with the given password and public key.

Meant to be used like:

```
with wrapper.connect(password='password0'):
    pass
```

Parameters

- **password** (*str*) – the password of the user on the remote server
- **public_key_filename** (*str*) – the file containing the public key

close()
Close any open connections and clear their attributes.

chdir(*remote_directory*)
Change directories to the remote directory.

Parameters **remote_directory** (*str*) – the directory to change to

open(*filename*, *mode*)
Open a file from the last used remote directory.

Parameters

- **filename** (*str*) – the name of the file to open
- **mode** (*str*) – the mode to use to open the file (see `file()` 's documentation for more information)

listdir_attr()
Retrieve a list of files and their attributes.

Each object is guaranteed to have a `filename` attribute as well as an `st_mtime` attribute, which gives the last modified time in seconds.

exec_command(*args*, *workdir*, *timeout=None*, *stdout_handler=None*, *stderr_handler=None*)
Execute a command on the remote server.

An example of how to use this function:

```
from datetime import timedelta
wrapper.exec_command(
    args=["ls", "-la", "."],
    workdir="/",
    timeout=timedelta(minute=5),
    stdout_handler=lambda now, output: print('stdout, now, output),
    stderr_handler=lambda now, output: print('stderr, now, output),
)
```

Parameters

- **args** (*list(str)*) – the command and arguments to run
- **workdir** (*str*) – the directory to run the commands from
- **timeout** (*datetime.timedelta*) – the timeout to use for the command
- **stdout_handler** – a function that accepts `now` and `output` parameters and is called when new output appears on stdout.
- **stderr_handler** – a function that accepts `now` and `output` parameters and is called when new output appears on stderr.

django_remote_submission.remote.**deploy_key_if_it_doesnt_exist**(*client*, *public_key_filename*)
Deploy our public key to the remote server.

Parameters

- **client** (*paramiko.client.SSHClient*) – an existing Paramiko client
- **public_key_filename** (*str*) – the name of the file with the public key

2.4 Serializers

Provide default serializers for managing this package's models.

class django_remote_submission.serializers.**ServerSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Serialize *django_remote_submission.models.Server* instances.

```
>>> from django_remote_submission.serializers import ServerSerializer
>>> serializer = ServerSerializer(data={
...     'id': 1,
...     'title': 'My Server',
...     'hostname': 'foo.invalid',
...     'port': 22,
... })
>>> serializer.is_valid()
True
```

class django_remote_submission.serializers.**JobSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Serialize *django_remote_submission.models.Job* instances.

```
>>> from django_remote_submission.serializers import JobSerializer
>>> serializer = JobSerializer(data={
...     'id': 1,
...     'title': 'My Job',
...     'program': 'print("Hello world")',
...     'status': 'INITIAL',
...     'owner': 1,
...     'server': 1,
... })
>>> serializer.is_valid()
True
```

class django_remote_submission.serializers.**LogSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Serialize *django_remote_submission.models.Log* instances.

```
>>> from django_remote_submission.serializers import LogSerializer
>>> serializer = LogSerializer(data={
...     'id': 1,
...     'time': '2012-04-23T18:25:43.511Z',
...     'content': 'Hello world',
...     'stream': 'stdout',
...     'job': 1,
... })
>>> serializer.is_valid()
True
```

2.5 URLs

Provide default route mappings for serializers.

`django_remote_submission.urls.`**`urlpatterns`**
The URL patterns for the defined serializers.

2.6 Views

Provide default views for REST API.

class `django_remote_submission.views.`**`ServerViewSet`**(***kwargs*)
Allow users to create, read, and update `Server` instances.

`serializer_class`
alias of `ServerSerializer`

`pagination_class`
alias of `StandardPagination`

class `django_remote_submission.views.`**`JobViewSet`**(***kwargs*)
Allow users to create, read, and update `Job` instances.

`serializer_class`
alias of `JobSerializer`

`pagination_class`
alias of `StandardPagination`

class `django_remote_submission.views.`**`LogViewSet`**(***kwargs*)
Allow users to create, read, and update `Log` instances.

`serializer_class`
alias of `LogSerializer`

`pagination_class`
alias of `StandardPagination`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

d

Symbols

C

D

E

F

I

J

L