minke

Thomas Leichtfuß

Sep 03, 2020

CONTENTS:

1 A framework for remote-control- and configuration-management-systems **1**

2 Indices and tables **13**

Index **15**

CHAPTER ONE

A FRAMEWORK FOR REMOTE-CONTROL- AND CONFIGURATION-MANAGEMENT-SYSTEMS

Minke is a framework that combines the full power of djangos admin-interface with the reliability and configurability of fabric2 and celery. A pure open-source- and pure python-solution to realize the most different szenarios concerning remote-control and configuration-managment.

Imagine you just need to setup some managment-commands for a handful of servers - you can do that with minke. Imagine you have lots of servers with different setups that you need to group and address seperatly - you can do that with minke. Imagine you have servers with multiple subsystems installed on each of them and you not just want to manage those systems but also to track configurations, version-numbers, installed extensions or modules and to filter for all of those values within your backend - you can do that with minke.

1.1 Minke uses

- django as backend and ORM
- fabric2 for remote-execution
- celery for asynchronous session-processing

1.2 Minke features

- full integration with django's admin-site
- asynchronously and parrallel session-execution
- realtime monitoring of executed sessions
- session- and command-history
- CLI using django's management-commands

1.2.1 Installation and Setup

Installation

To install django-minke and all its dependencies use pip:

```
$ pip install django-minke
```

The dependencies are:

- Django (>=1.11)
- celery (>=4.2.2)
- fabric2 (>=2.4.0)
- djangorestframework (>=3.9.2)

Setup

Django

Minke is build as a django-application. For more informations about how to setup a django-project please see the django-documentation:

- Getting started with django

Celery

Minke uses celery to realize asynchrouniously and parrallel task-execution. For more informations about how to setup a django-project with celery please see the celery-documentation:

- Getting started with celery
- First steps with django

Note: Minke depends on a celery-setup with a working result-backend. We recommend to use the django-celery-result-extension:

- General informations about result-backends
- django-celery-result

Fabric

Minke uses fabric to realize remote-execution. Fabric itself is build on invoke (command-execution) and paramiko (ssh-connections). Fabric and invoke are highly configurable. Please see *Fabric-integration* for more informations about how to configure fabric within a minke-project.

1.2.2 Getting started

Install rabbitmq-server as message-broker used by celery (debian):

```
$ apt-get install rabbitmq-server
```

Install the following python-libraries:

```
$ pip install minke-django
$ pip install django-celery-results
```

Setup a django-celery-project as described here:

- First steps with django

Add the following django-apps to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    'minke',
    'rest_framework',
    'django_celery_results',
    'my_minke_app',
    ...
    ...
]
```

Add the following setting-parameters to your *settings.py*:

```
CELERY_BROKER_URL = 'amqp://guest:guest@localhost//'
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_RESULT_BACKEND = 'django-db'
CELERY_CACHE_BACKEND = 'django-cache'
```

Create your first session within *session.py*:

```
from minke.models import Host
from minke.sessions import CommandChainSession


class ServerInfos(CommandChainSession):
    verbose_name = 'Get informations about the server.'
    work_on = (Host,)
    commands = [
        'lsb_release --all',
        'hostname --long',
        'hostname --ip-address']
```

Now start your django-development-server. . . :

```
$ python manage.py runserver
```

. . . and the celery worker-process:

```
$ celery worker --concurrency=8 --events --loglevel=INFO --app my_minke_app
```

Now you should be able to visit your django-admin-site and create some hosts pointing to your servers. By default fabric uses your local ssh-agent to initialize remote-connections. So your minke-app will be able to work on all those servers you have access to via PubkeyAuthentication.

1.2.3 Concept

The main idea of minke is to define an arbitrary data-structure that represents your server- and subsystem-landscape as django-models. And then be able to run specific remote-tasks for those models right out of their changelist-sites. This works pretty much as django's admin-actions: Simply choose the items and start a session on them. Now those sessions could be anything about remote-control and system-managment, but also fetching relevant data to update your selected items itself.

To make this possible there are three main elements:

- hosts,
- minke-models,
- and sessions.

Hosts

A Host is a django-model that is basically the database-representation of a specific ssh-config. It holds every information needed by fabric to connect to the remote-system.

Read more about hosts.

Minke-Models

Minke-models now are all those models that you want to run remote-sessions with. This could be the data-representation of a server, but also of web-applications running on this server, and even something like installed extensions, patches, backups and everything else you want to track and manage in your minke-app.

Read more about minke-models.

Sessions

A session-class defines the code to be executed for minke-models. Sessions are similar to fabric-tasks. Within a session you have access to a connection-object (as implemented by fabric) as well as to the minke-model-object you are working with. So a session could be as simple as running a single shell-command on the remote-machine, up to complex management-routines including manipulating the data of the minke-model-object itself.

Read more about sessions.

1.2.4 Hosts

TODO

1.2.5 Minke-models

To run sessions with a specific model three conditions must be complied:

- The model must be a subclass of `MinkeModel`.
- The model's ModelAdmin must be a subclass of `MinkeAdmin`.
- And the model must have a connection to a `Host`.

For minke to be able to run a session for a specific minke-model, it must be able to associate the model with a host in a distinct way. By default minke looks for a `host`-attribute as a OneToOne- or a ManyToOne-relation:

```
host = models.ForeignKey(Host, on_delete=models.CASCADE)
```

It is also possible to work with intermediated models. Assumed you have a model-structure as follows:

```
class Server(MinkeModel):
    host = models.OneToOneField(Host, on_delete=models.CASCADE)
    hostname = models.CharField(max_length=128)


class Drupal(MinkeModel):
    server = models.ForeignKey(Server, on_delete=models.CASCADE)
    name = models.CharField(max_length=128)
    version = models.CharField(max_length=128)

    HOST_LOOKUP = 'server__host'


class DrupalModule(MinkeModel):
    drupal = models.ForeignKey(Drupal, on_delete=models.CASCADE)
    name = models.CharField(max_length=128)
    version = models.CharField(max_length=128)

    HOST_LOOKUP = 'drupal__server__host'
```

To be able to use each of those models as a minke-model, you need to assign a lookup-string to `HOST_LOOKUP` with a format you would also use when `filtering a queryset`:

```
$ DrupalModul.objects.filter(drupal__server__host=host)
```

1.2.6 Sessions

TODO

1.2.7 Messages

TODO

1.2.8 Fabric-integration

Minke uses fabric to realize remote-execution. Fabric itself is build on invoke (command-execution) and paramiko (ssh-connections). Fabric and invoke are highly configurable.

TODO

1.2.9 Adminsite

TODO

1.2.10 Contrib-commands

TODO

1.2.11 Api-commands

TODO

1.2.12 Sessions

class `minke.sessions.`**`Session`**(*con*, *db*, *minkeobj=None*)

Base-class for all session-classes.

All session-classes must inherit from Session. By defining a subclass of Session the subclass will be implicitly registered as session-class and also a run-permission will be created for it. To prevent this behavior use an abstract session by setting *`abstract`* to True.

Each session will be instantiated with a fabric-connection and an object of `MinkeSession`. The connection-object provides the remote-access, while the minkesession is the database- representation of a specific session running for a specific `minkemodel-object`.

For a session-class to be useful you at least has to define the *`process()`*-method and add one or more `MinkeModel` to *`work_on`*-attribute.

`abstract = True`

An abstract session-class won't be registered itself. This is useful if your session-class should be a base-class for other sessions.

Abstract session-classes can be registered manually by calling its classmethod `register()`:

```
MySession.register()
```

This won't add a run-permission-lookup-string to *`permissions`*. To do so use the classmethod `SessionRegistration.add_permission()`:

```
MySession.add_permission()
```

`verbose_name = None`

Display-name for sessions.

`group = None`

Group-name used as optgroup-tag in the select-widget. Best practice to group sessions is to use a `SessionGroup`.

work_on = ()
Tuple of minke-models. Models the session can be used with.

permissions = ()
Tuple of permission-strings. To be able to run a session a user must have all the permissions listed. The strings should have the following format: "<app-label>.<permission's-codename>.

auto_permission = True
If True a lookup-string for a session-specific run-permission will be automatically added to *permissions*.

Note: To avoid database-access on module-level we won't create the permission itself. Once you setup your sessions you could create run-permissions for all sessions using the api-command:

```
$ ./manage.py minkeadm --create-permissions
```

form = None
An optional form that will be rendered before the session will be processed. The form-data will be accessible within the session as the data-property. Use it if the session's processing depends on additional user-input-data.

Instead of setting the form-attribute you can also directly overwrite *get_form()*.

confirm = False
If confirm is true, the admin-site asks for a user-confirmation before processing a session, which also allows to review the objects the session was revoked with.

invoke_config = {}
Session-specific fabric- and invoke-configuration-parameters which will be used to initialize a `fabric-connection`. The keys must be formatted in a way that is accepted by `load_snakeconfig()`.

See also the documentation for the configuration of fabric and invoke.

parrallel_per_host = False
Allow parrallel processing of multiple celery-tasks on a single host. If multiple minke-objects are associated with the same host all tasks running on them would be processed in a serial manner by default. This is to protect the ressources of the host-system. If you want to allow parrallel processing of multiple celery-tasks on a single host set parrallel_per_host to True.

Note: To perform parrallel task-execution on a single host we make use of celery's chords-primitive, which needs a functioning result-backend to be configured. Please see the celery-documentation for more details.

classmethod get_form()
Return *form* by default.

Overwrite this method if you need to setup your form-class dynamically.

property c
Refers to the `fabric.connection.Connection`-object the session was initialized with.

property minkeobj
Refers to `models.MinkeSession.minkeobj`.

property status
Refers to `models.MinkeSession.session_status`.

property data
Refers to `models.MinkeSession.session_data`. This model-field holds all the data that comes from *form*.

stop (**arg*, ***kwargs*)
Interrupt the session's processing.

This method could be called twice. The first time it will initiate a soft interruption which means a current remote-process won't be interrupted. The session will be stopped subsequently. If it is called meanwhile a second time, the session will be killed immediately.

Note: It seems that there is no chance to interrupt a shell-process started by fabric if no pty is in use. fabric.runners.Remote.send_interrupt says:

```
... in v1, we just reraised the KeyboardInterrupt unless a PTY was
present; this seems to have been because without a PTY, the
below escape sequence is ignored, so all we can do is immediately
terminate on our end...
```

Thus killing a session makes most sense if it has the run.pty-config in use. Otherwise you just will be disconnected from the remote-process.

process ()
Real work is done here…

This is the place for your own session-code.

add_msg (*msg*, *level=None*)
Add a message.

Parameters

- **msg** – You could either pass an instance of any `message-class`, or any type the different message-classes are initiated with:
  - a string for a `PreMessage`
  - a tuple or list for a `TableMessage`
  - an object of `Result` for a `ExecutionMessage`
- **level** (*string or bool (optional)*) – This could be one of 'info', 'warning' or 'error'. If you pass a bool True will be 'info' and False will be 'error'.

set_status (*status*, *update=True*)
Set session-status. Pass a valid session-status or a bool.

Parameters

- **status** (*string or bool*) – Valid `status` as string or True for 'success' and False for 'error'.
- **update** (*bool (optional)*) – If True the session-status could only be raised. Lower values as current will be ignored.

format_cmd (*cmd*)
Use the *data* and the fields of the *minkeobj* as parameters for `format()` to prepare the given command.

Parameters **cmd** (*string*) – a format-string

Returns The formatted command.

Return type string

run(*cmd*, ***invoke_params*)
Run a command.

Basically call `run()` on the `Connection`-object with the given command and invoke-parameters.

Additionally save the `Result`-object as an `models.CommandResult`-object.

Parameters

- **cmd** (*string*) – The shell-command to be run.
- **(optional)** (***invoke_params*) – Parameters that will be passed to `run()`

Returns

Return type object of `models.CommandResult`

frun(*cmd*, ***invoke_params*)
Same as *run()*, but use *format_cmd()* to prepare the command-string.

xrun(*cmd*, ***invoke_params*)
Same as *frun()*, but also add a `ExecutionMessage` and update the session-status.

update_field(*field*, *cmd*, *regex=None*, ***invoke_params*)
Running a command and update a field of *minkeobj*.

Assign either result.stdout or if available the first matched regex-group. If result.failed is True or result.stdout is empty or the given regex does not match, the field is updated with None. In this case an error-message will be added.

Parameters

- **field** (*string*) – Name of the field that should be updated.
- **cmd** (*string*) – The shell-command to be run.
- **regex** (*string (optional)*) – A regex-pattern the `CommandResult` will be initialized with.
- **(optional)** (***invoke_params*) – Parameters that will be passed to `run()`

Returns False if the field was updated with None. True otherwise.

Return type bool

Raises AttributeError – If the given field does not exists on *minkeobj*.

class minke.sessions.**SingleCommandSession**(*con*, *db*, *minkeobj=None*)
An abstract *Session*-class to execute a single command.

If you want your session to execute a single command simply create a subclass of SingleCommandSession and overwrite the *command*-attribute. The command will be executed with *xrun()*.

Example

```
class MyDrupalModel(models.Model):
    root = models.CharField(max_length=255)

class MySession(SingleCommandSession):
    work_on = (MyDrupalModel,)
    command = 'drush --root={root} cache-clear all'
```

command = None
Shell-command to be executed.

process()
Real work is done here…

This is the place for your own session-code.

class minke.sessions.**CommandFormSession**(*con*, *db*, *minkeobj=None*)
Same as class:.*SingleCommandSession* but rendering a TextField to enter the command.

Example

```
class MySession(CommandFormSession):
    work_on = (MyModel,)
```

form
alias of `minke.forms.CommandForm`

class minke.sessions.**CommandChainSession**(*con*, *db*, *minkeobj=None*)
An abstract *Session*-class to execute a sequence of commands.

The commands will be invoked one after another. If one of the commands return with a state defined in *break_states* no further commands will be executed.

Example

```
class MySession(CommandChainSession):
    work_on = (MyServer,)
    commands = (
        'a2ensite mysite'
        'apachectl configtest',
        'service apache2 reload')
```

commands = ()
tuple of shell-commands

break_states = ('error',)
tuple of `models.CommandResult.status` on which the session will be interrupted

process()
Real work is done here…

This is the place for your own session-code.

class minke.sessions.**SessionChain**(*con*, *db*, *minkeobj=None*)
An abstract *Session*-class to execute a sequence of sessions.

If you have some sessions you want to be able to process separately or as a sequence you could make use of a `SessionChain`.

All sessions have to work with the same `minkeobj`. If one of the sessions ends up with a status defined in `break_states` no further sessions will be executed.

Example

```
class MySession(SessionChain):
    work_on = (MyServer,)
    sessions = (
        UpdateSQLServer,
        RestartSQLServer,
        UpdateApache,
        RestartApache)
```

Note: It is possible to add abstract sessions to `sessions`.

Warning: Only the `invoke_config` of the main session will be applied. The `invoke_config` of the sessions in `sessions` will be ignored.

The same is true for `get_form()`. Only the form returned by the main session's `get_form()` will be rendered.

`sessions = ()`
tuple of `Session`

`break_states = ('error',)`
tuple of `Session.status` on which further processing will be skipped.

`process``()`
Real work is done here…

This is the place for your own session-code.

1.2.13 Messages

TODO

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

abstract (*minke.sessions.Session attribute*), 6
add_msg() (*minke.sessions.Session method*), 8
auto_permission (*minke.sessions.Session attribute*), 7

B

break_states (*minke.sessions.CommandChainSession attribute*), 10
break_states (*minke.sessions.SessionChain attribute*), 11

C

c() (*minke.sessions.Session property*), 7
command (*minke.sessions.SingleCommandSession attribute*), 10
CommandChainSession (*class in minke.sessions*), 10
CommandFormSession (*class in minke.sessions*), 10
commands (*minke.sessions.CommandChainSession attribute*), 10
confirm (*minke.sessions.Session attribute*), 7

D

data() (*minke.sessions.Session property*), 7

F

form (*minke.sessions.CommandFormSession attribute*), 10
form (*minke.sessions.Session attribute*), 7
format_cmd() (*minke.sessions.Session method*), 8
frun() (*minke.sessions.Session method*), 9

G

get_form() (*minke.sessions.Session class method*), 7
group (*minke.sessions.Session attribute*), 6

I

invoke_config (*minke.sessions.Session attribute*), 7

M

minkeobj() (*minke.sessions.Session property*), 7

P

parrallel_per_host (*minke.sessions.Session attribute*), 7
permissions (*minke.sessions.Session attribute*), 7
process() (*minke.sessions.CommandChainSession method*), 10
process() (*minke.sessions.Session method*), 8
process() (*minke.sessions.SessionChain method*), 11
process() (*minke.sessions.SingleCommandSession method*), 10

R

run() (*minke.sessions.Session method*), 9

S

Session (*class in minke.sessions*), 6
SessionChain (*class in minke.sessions*), 10
sessions (*minke.sessions.SessionChain attribute*), 11
set_status() (*minke.sessions.Session method*), 8
SingleCommandSession (*class in minke.sessions*), 9
status() (*minke.sessions.Session property*), 7
stop() (*minke.sessions.Session method*), 8

U

update_field() (*minke.sessions.Session method*), 9

V

verbose_name (*minke.sessions.Session attribute*), 6

W

work_on (*minke.sessions.Session attribute*), 6

X

xrun() (*minke.sessions.Session method*), 9