Serveradmin

InnoGames GmbH

Aug 02, 2023

CONTENTS

CHAPTER
ONE

EXTENDING SERVERADMIN

Serveradmin is a Django application. General knowledge about running Django applications would be useful.

1.1 Running Serveradmin

We provide a docker-compose setup that gives you a local development instance with 2 commands.

First make sure you have docker-compose installed as described here.

Then run these two commands:

```
cp .env.dist .env
docker-compose up
```

The default values in .env.dist are sufficient however feel free to adjust them to your needs.

You can access the web service to execute Django commands and run scripts:

```
docker-compose exec web

# Example: Run Django management commands
pipenv run python -m serveradmin -h

# Example: Use the Python Remote API
pipenv run python -m adminapi "hostname=example.com"
```

Tip

You may still want to have a virtual environment for Serveradmin on your host machines and run pipenv install -D to have all modules available for your IDEs auto completion etc.

1.2 Database Dump

If you have a running instance of Serveradmin which is reachable via SSH you can update the PRODUCTION_DB variable in .env to your database host and run dump.sh from your **host** machine:

```
# .env
PRODUCTION_DB=your-serveradmin-db-host.example.com

# Execute on host
./dump.sh
```

1.3 Testing your changes

We have some tests which are executed when making a PR that you can already run locally to check if your changes are breaking anything existing. They are far from comprehensive at the time of writing this but can safe you some manual testing.

You can execute the tests with the following commands

> # Tests for the commandline interface: adminapi pipenv run python -m unittest discover adminapi -v
>
> # Tests for the backend code pipenv run python -Wall -m serveradmin test –noinput –parallel

1.4 Bonus: Setting up a cool debugger

Install `django-extensions` and `werkzeug` using pip:

```
pip install django-extensions werkzeug
```

and add `'django_extensions'` to your `INSTALLED_APPS` setting in the `local_settings.py`.

Now you can use `python -m serveradmin runserver_plus` to start the local test webserver with the Werkzeug debugger.

See http://packages.python.org/django-extensions/ for details.

1.5 Code style guideline

First of all, read the Python style guide (PEP 8). The most important things:

- Use 4 spaces for indention, **not** tabs
- Functions and variables use underscores (e.g. `config_dir`)
- Classes use CamelCase (e.g. `NagiosCommit`)
- Try to keep lines less than 80 chars

Warning: Ignoring the style guide will make your local Python expert quite sad!

1.6 Terminology

Just to have same names:

project: Many applications together with settings, a global `urls.py` and the `__main__.py` form a project. The "serveradmin" is a project.

application (or "app"): An application is basically a combination of several files for the same topic. You may have an application for nagios, graphs, the servershell etc. Applications consist of views, models and templates. If you are familiar with MVC pattern, think of views being the controllers and the templates the views.

models: The models will contain your application logic. This is mostly your database structure and operations on on it, but also stuff that's not related to the database. In your application you will find a `models.py` where you can put your code in. Django calls a class inheriting `django.db.models.Model` a model, which should not be mistaken for the models itself (e.g. a class for your database table and operations vs. your application logic in general)

views: The views will get the input from the user and ask the model for the execution of operations or fetch data from the model to pass it to the template. As already said, it's known as the controller in the MVC pattern. You will add your view functions to the `views.py` in your application.

templates: The template is - in most cases - just an ordinary HTML file with some template markup to display the data it got from the view. They usually reside in a directory named `yourapp/templates/yourapp`. You have to create it yourself for a new application.

1.7 Short git introduction

Set your name and email:

```
git config --global user.name "Your Name"
git config --global user.email your.name@innogames.de
```

Fetch new changes from remote repository:

```
git pull
```

For changes create a new branch, and switch to it:

```
git branch my_changes
git checkout my_changes
```

Do your code changes and don't forget to commit often. It's good to commit even small changes. Before you commit, you have to add files (*even just modified files*):

```
git add new_file
git add file_you_have_modified
git commit
```

Don't forget to put a meaningful commit message.

Once you have done all your changes and your version is ready for deployment you can merge it back to main. You may want to fetch changes from remote first:

```
git checkout main
git pull # Optionally fetch changes from remote
git merge my_changes
```

After merging was successful, you can delete your branch:

```
git branch -d my_changes
```

It is recommended to do a rebase. This will help to have a clear history:

```
git rebase
```

And finally push your changes to the remote repository:

```
git push
```

Have any changes you don't want to commit and still want to change branch? Use git stash:

```
git stash # Will save your uncomitted changes
# Do whatever you want (e.g. changing branches)
git stash pop # Will apply changes again and pop it from stash
```

1.8 Short Django introduction

If you have some time I recommend doing the Django Tutorial. It covers many topics and gives your a good overview.

For people in a hurry: You will find the Serveradmin in the `serveradmin` directory while the Remote API (aka. adminapi) is inside `adminapi`. We will only cover the Serveradmin in this document.

Inside the serveradmin you will find the following files:

- `urls.py`
- `settings.py`

The `settings.py` contains your settings. You have already edited this file. Inside the `urls.py` you can define URLs for the Serveradmin. In most cases you will have an own `urls.py` in your application.

We will create a small example application named "secinfo" (for "security information"). **Please don't commit this application, it is for learning purposes only!**

We will use `python -m serveradmin` to create our application:

```
python -m serveradmin startapp secinfo
```

Now we have a directory named `secinfo` with some files inside it. We will move it into the directory `serveradmin`.

1.9 Adding functions to the remote API

To create new functions which are callable by the Python remote API you have to define them inside the `api.py` file in your application. If it doesn't exist, you can just create it.

To export the function you will use the `api_function` decorator, as shown in the following example:

```
from serveradmin.api.decorators import api_function

@api_function(group='example')
def hello(name):
   return 'Hello {0}!'.format(name)
```

Now you can call this function remotely:

```
from adminapi import api

example = api.get('example')
print example.hello('world') # will print 'Hello world!'
```

The API uses JSON for communication, therefore you can only return and receive a restricted set of types. The following types are supported: string, integer, float, bool, dict, list and None. You can also receive and return datetime/date objects, but they will be converted to an unix timestamp prior sending. You have to convert them back manually by using `datetime.fromtimestamp`.

It has also limited support for exceptions. You can either raise a ValueError if you get invalid parameters or use serveradmin.api.ApiError for other exceptions. You can subclass ApiError for more specific exceptions. Raising exception has also one other restriction: you can only pass a message, but not additional attributes on the exception.

Look at the following example:

```
from serveradmin.api.decorators import api_function
from serveradmin.api import ApiError

@api_function(group='example')
def nagios_downtimes(from_time, to_time):
    if to_time < from_time:
        raise ValueError('From must be smaller than to')

    try:
        return get_nagios_downtimes(from_time, to_time)
    except NagiosError, e:
        # Propagating NagiosError would raise an exception in the
        # serveradmin, but not on the remote side. You have to catch
        # it and reraise it as ApiError or subclass of ApiError
        raise ApiError(e.message)
```

1.10 Handling Permissions

We will use Django's integrated Permission system. In Django, you will define permissions on a model. You will automatically get a few magic permissions named app_label.(add|change|delete)_modelname. For example: if you have a class Bird in your application bird you will get permissions named bird.add_bird etc. If you need own permissions, you have to define them like this:

```
class Bird(models.Model):
    # Fields left out

    class Meta:
       permissions = (
          ('can_fly', 'Can fly'),
       )
```

You will now get a permission named bird.can_fly.

If you don't have a model class you have to create one. This will normally also create a database table, but you can avoid it by setting managed to False. This will tell Django that it shouldn't manage the database for this model. See the following example:

```
class ddosmanager (models.Model):

    class Meta:
        managed = False
        permissions = (
            ('set_state',    'Can enable and disable DDoS Mitigation'),
            ('set_prefixes', 'Can modify prefixes announced to DDoS Mitigation provider
↪'),
            ('view', 'Can view DDoS Mitigation state and prefixes'),
        )
```

There are several ways to check for permissions at different levels. To check permissions on a view, use the `permission_required` decorator:

```
from django.contrib.auth.decorators import permission_required

@permission_required('can_view_graphs')
def view_graphs(request):
    pass # Do some stuff and render template
```

It will disallow calling this view for all users that don't have the required permission.

To check permissions in the template you can use the `perms` proxy. Look at the following example:

```
{% if perms.bird.add_bird %}
<a href="{% url bird_add %}">Add a bird</a>
{% endif %}
```

Warning: Just hiding things it the template might not be enough. For example you should not hide a form, but leave the view with form processing unchecked.

In the code permissions can be checked using the `user.has_perm` method. See the following example in a view:

```
def change_bird(request, name):
    bird = get_object_or_404(Bird, pk=range_id)

    if request.method == 'POST':
       can_delete = request.user.has_perm('bird.delete_bird')
       can_edit = request.user.has_perm('bird.change_bird')
       if action == 'delete' and can_delete:
           bird.delete()
       if action == 'edit' and can_edit:
           pass # edit ip range
```

To grant permissions to users, use the Django admin interface. Superusers will have all permissions be default.

See the Django documentation on permissions for details.

CHAPTER

TWO

PYTHON REMOTE API

The Adminapi provides a python module which can talk to the Serveradmin via an API. It provides functions for querying servers, modifying their attributes, triggering actions (e.g. committing nagios) etc.

Warning: This is only a draft. The API might change.

2.1 Authentication

Every script that uses the module must authorize itself before using the API. You need to generate a so called application for every script in the admin interface of the serveradmin. This has several benefits over using a generic password:

- Logging changes that were done by a specific script
- Providing a list with existing scripts which are using the API
- Possibility to revoke an authentication token without changing every script

The API allows authentication of an application either via public-key cryptography (ssh-keys) or pre shared keys (passwords). Using the new public-key style authentication has even more benefits:

- An application can have multiple public keys, making it easier to change them
- Adminapi can sign requests via keys in your local or forwarded ssh-agent
- Keys in the ssh-agent can be password protected on disk and decrypted only inside the agent. Adminapi never even sees the private part of the key
- Serveradmin only knows the public part of the key, while an admin can read all pre shared keys from serveradmin and use them to impersonate others.

To authenticate yourself via an ssh key you have to add the public part to an application in serveradmin. You can then either add the private key to your ssh-agent or export the SERVERADMIN_KEY_PATH environment variable to the path of the private key:

```
# Use ssh-agent, passwords protected keys are supported
ssh-add ~/.ssh/id_rsa

# Use environment vairable, passwords protected keys are _not_ supported
export SERVERADMIN_KEY_PATH=~/.ssh/id_rsa
```

Note that for ed25519 key support both adminapi and serveradmin must have paramiko 2.2 or newer installed.

To authenticate yourself via a pre shared key you need to set the SERVERADMIN_TOKEN environment variable or create a file called .adminapi in the home folder of your user:

```
# Use environment variable (Useful for transient jobs such as Jenkins)
export SERVERADMIN_TOKEN=MLifIK9FMQTaFDneDneNg30pb

# Use .adminapirc file in home folder
echo "auth_token=MLifIK9FMQTaFDneDneNg30pb" >> ~/.adminapirc
chmod 0600 ~/.adminapirc
```

The order of prevalence is:

- SERVERADMIN_KEY_PATH if set
- SERVERADMIN_TOKEN if set
- ~/.adminapirc if present
- ssh-agent if present

Note that we try to authenticate with all keys in the agent. If multiple keys, belonging to different applications, match you will get a permission denied. This is because the associated apps likely have different permissions and we don't want to guess which to enforce. Trying to authenticate with more than 20 keys will also be denied to prevent a DOS.

2.2 Querying and modifying servers

Using the `dataset` module you can filter servers by given criteria and modify their attributes.

2.2.1 Basic queries

You can use the *`adminapi.dataset.Query`* function to find servers which match certain criteria. See the following example which will find all webservers of Tribal Wars:

```
from adminapi.dataset import Query

hosts = Query({'servertype': 'vm', 'game_function': 'web'})

for host in hosts:
    print(host['hostname'])
```

The Query class takes keyword arguments which contain the filter conditions. Each key is an attribute of the server while the value is the value that must match. You can either use strings, integers or booleans for exact value matching. All filter conditions will be ANDed.

More often you need filtering with more complex conditions, for example regular expression matching, comparison (less than, greater than) etc. For this kind of queries there is a filters modules which defines some filters you can use. The following example will give you all Tribal Wars webservers, which world number is between 20 and 30:

```
from adminapi.filters import All, GreaterThan, LessThan

hosts = Query({
    'servertype': 'vm',
    'game_function': 'web',
    'game_world': All(GreaterThan(20), LessThan(30)),
})
```

2.2.2 Accessing and modifying attributes

Each server is represented by a server object which allows a dictionary-like access to their attributes. This means you will have the usual behaviour of a dictionary with methods like `keys()`, `values()`, `update(...)` etc.

You can get server objects by iterating over a query or by calling `get()` on the query. Changes to the attributes are not directly committed. To commit them you must call `commit()` on the query.

Here is an example which cancels all servers for Seven Lands:

```
hosts = Query({'servertype': 'hardware'}, ['canceled'])
for host in hosts:
    hosts['canceled'] = True
hosts.commit()
```

Another example will print all attributes of VM objects and check for the existence of the `function` attribute:

```
vm = Query().new_object('vm')
for attr, val in vm.items():
     print('{} => {}'.format(attr, val))

if 'function' not in techerror:
     print('Something is wrong!')'
```

Multi attributes are stored as instances of *`adminapi.dataset.MultiAttr`*, which is a subclass of set. Take a look at `set` for the available methods. See the following example which iterates over all additional IPs and adds another one:

```
techerror = Query({'hostname': 'techerror.support.ig.local'}, ['additional_ips']).get()
for ip in techerror['additional_ips']:
     print(ip)
techerror['additional_ips'].add('127.0.0.1')
```

Warning: Modifying attributes of a server object that is marked for deleting will raise an exception. The `update()` function will skip servers that are marked for deletion.

2.2.3 Query Reference

The *`adminapi.dataset.Query`* function returns a query object that supports iteration and some additional methods.

`class Query`

`__iter__()`
Return an iterator that can be used to iterate over the query. The result itself is cached, iterating several times will not hit thedatabase again. You usually don't call this function directly, but use the class' object in a for-loop.

`__len__()`
Return the number of servers that where returned. This will fetch all results.

`get()`
Return the first server in the query, but only if there is just one server in the query. Otherwise, you will get an exception. #FIXME: Decide kind of exception

commit_state()
Return the state of the object.

commit()
Commit the changes that were done by modifying the attributes of servers in the query. Please note: This will only affect servers that were accessed through this query!

rollback()
Rollback all changes on all servers in the query. If the server is marked for deletion, this will be undone too.

delete()
Marks all server in the query for deletion. You need to commit to execute the deletion.

Warning: This is a weapon of mass destruction. Test your script carefully before using this method!

update(***attrs*)
Mass update for all servers in the query using keyword args. Example: You want to cancel all Seven Land servers:

```
Query({'servertype': 'hardware'}).update(canceled=True)
```

This method will skip servers that are marked for deletion.

You still have to commit this change.

2.2.4 Server object reference

The reference will only include the additional methods of the server object. For documentation of the dictionary-like access see `dict`.

class DatasetObject

old_values
Dictionary which contains the values of the attributes before they were changed.

is_dirty()
Return True, if the server object has uncomitted changes, False otherwise.

is_deleted()
Return True, if the server object is marked for deletion.

delete()
Mark the server for deletion. You need to commit to delete it.

2.3 Making API calls

API calls are split into several groups. To call a method you need to get a group object first. See the following example for getting a free IP:

```
# Do authentication first as described in section "Authentication"
from adminapi import api

nagios = api.get('nagios')
nagios.commit('push', 'john.doe', project='techerror')
```

CHAPTER

THREE

ADMINAPI MODULE DOCUMENTATION

`class adminapi.dataset.Query(filters=None, restrict=['hostname'], order_by=None)`

`class adminapi.dataset.MultiAttr(other, obj, attribute_id)`

This class must redefine all mutable methods of the set class to maintain the old values on the DatasetObject.

`add(elem)`

Add an element to a set.

This has no effect if the element is already present.

`clear()`

Remove all elements from this set.

`copy()`

Return a shallow copy of a set.

`difference_update(*others)`

Remove all elements of another set from this set.

`discard(elem)`

Remove an element from a set if it is a member.

If the element is not a member, do nothing.

`intersection_update(*others)`

Update a set with the intersection of itself and another.

`pop()`

Remove and return an arbitrary set element. Raises KeyError if the set is empty.

`remove(elem)`

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

`symmetric_difference_update(other)`

Update a set with the symmetric difference of itself and another.

`update(*others)`

Update a set with the union of itself and others.

INDEX