pyfarm.master Documentation

Release 0.8.5

Oliver Palmer, Guido Winkelmann

February 17, 2015

This package contains the models, web interface, APIs, and backend components necessary to scheduler and allocate jobs on PyFarm.

Contents

CHAPTER 1

Commands

1.1 Standard Commands

1.1.1 pyfarm-create-tables

```
usage: pyfarm-tables [-h] [--echo] [--drop-all] [--no-create-tables]

Creates PyFarm's tables

optional arguments:
  -h, --help          show this help message and exit
  --echo              If provided then echo the SQL queries being made
  --drop-all          If provided all tables will be dropped from the database
                      before they are created.
  --no-create-tables  If provided then no tables will be created.
```

1.2 Development Commands

1.2.1 pyfarm-master

```
usage: pyfarm-master [-h] [--drop-all] [--create-all]
                     [--confirm-drop CONFIRM_DROP]
                     [--allow-agent-loopback-addresses]

optional arguments:
  -h, --help            show this help message and exit
  --drop-all, -D        drop the existing tables before starting
  --create-all, -C      create all tables before starting
  --confirm-drop CONFIRM_DROP
  --allow-agent-loopback-addresses
```

CHAPTER 2

Environment Variables

PyFarm's master and models have several environment variables which can be used to change the operation at runtime. For more information see the individual sections below.

Note: Not all environment variables defined below are directly used by PyFarm. Many of these values are provided to make it easier to group settings together and so settings for PyFarm won't conflict with any existing software.

2.1 Database Schema

Environment variables that are used to setup or control the database backend.

Warning: These values are used to construct the database schema. If your schema already exists then changing these values may have uninteded consequences.

PYFARM_DB_PREFIX
The prefix for all table names. Normally this should never be changed but could be for testing or similiar similiar circumstances. **NOTE**: making this value to long may produce errors in some databases such as MySQL

PYFARM_DB_MAX_HOSTNAME_LENGTH
The maximum length a hostname can be **Default**: 255

PYFARM_DB_MAX_JOBTYPE_LENGTH
The maximum length for the name of a jobtype. **Default**: 64

PYFARM_DB_MAX_COMMAND_LENGTH
The maximum length a single command can be. **Default**: 64

PYFARM_DB_MAX_USERNAME_LENGTH
The maximum length a username can be. **Default**: 255

PYFARM_DB_MAX_EMAILADDR_LENGTH
The maximum length a email address can be **Default**: 255

PYFARM_DB_MAX_ROLE_LENGTH
The maximum length a role can be **Default**: 128

PYFARM_DB_MAX_TAG_LENGTH
The maximum length a tag can be **Default**: 64

NOTE PyFarm uses the word 'tag' in several places. This value controls the max length of any string which is a tag.

PYFARM_DB_MAX_PROJECT_NAME_LENGTH
The maximum length any one project name can be. **Default**: 32

2.2 Database Constraints and Validation

Unlike the above section, these values are checked when a database entry is modified or created. They are intended to provide validation so erroneous data cannot be inserted. Do note however the **max** value any integer can be raised to is 2147483647.

PYFARM_AGENT_CPU_ALLOCATION
The total amount of cpu space an agent is allowed to work in. For example if four jobs requires four cpus and `PYFARM_AGENT_CPU_ALLOCATION` is 1.0 then all those jobs can be assigned to the agent. If `PYFARM_AGENT_CPU_ALLOCATION` was .5 however only half of those jobs could be assigned. This value must always be greater than 0. **Default**: .8

PYFARM_AGENT_RAM_ALLOCATION
Same as `PYFARM_AGENT_CPU_ALLOCATION` except for ram resources. This value must always be greater than 0. **Default**: 1.0

PYFARM_AGENT_MIN_PORT
The minimum port an agent is allowed to communicate on. **Default**: 1024

PYFARM_AGENT_MAX_PORT
The maximum port an agent is allowed to communicate on. **Default**: 65535

PYFARM_AGENT_MIN_CPUS
The minimum number of cpus an agent is allowed to have. **Default**: 1

PYFARM_AGENT_MAX_CPUS
The maximum number of cpus an agent is allowed to have. **Default**: 256

PYFARM_AGENT_MIN_RAM
The minimum amount of ram, in megabytes, an agent is allowed to have. **Default**: 16

PYFARM_AGENT_MAX_RAM
The maximum amount of ram, in megabytes, an agent is allowed to have. **Default**: 262144

PYFARM_QUEUE_MIN_PRIORITY
The minimum priority any job or task is allowed to have. **Default**: -1000

PYFARM_QUEUE_MAX_PRIORITY
The maximum priority any job or task is allowed to have. **Default**: 1000

PYFARM_QUEUE_DEFAULT_PRIORITY
The default priority any new jobs or tasks are given **Default**: 0

PYFARM_QUEUE_MIN_BATCH
The minimum number of tasks which can be sent to a single agent for processing. **Default**: 1

PYFARM_QUEUE_MAX_BATCH
The maximum number of tasks which can be sent to a single agent for processing. **Default**: 64

PYFARM_QUEUE_DEFAULT_BATCH
The default number of tasks which can be sent to a single agent for processing. **Default**: 1

PYFARM_QUEUE_MIN_REQUEUE
The minimum number of times a task is allowed to reque. **Default**: 0

PYFARM_QUEUE_MAX_REQUEUE
The maximum number of times a task is allowed to reque. Not setting this value will allow **any** tasks to reque an infinite number of times if requested by a user. **Default**: 10

PYFARM_QUEUE_DEFAULT_REQUEUE
The default number of times a task is allowed to reque. **Default**: 3

PYFARM_QUEUE_MIN_CPUS
The minimum number of cpus that can be required to any one job. **Default**: 1

PYFARM_QUEUE_MAX_CPUS
The maximum number of cpus that can be required to any one job. **Default**: 256

PYFARM_QUEUE_DEFAULT_CPUS
The default number of cpus required for any one job. **Default**: 1

PYFARM_QUEUE_MIN_RAM
The minimum amount of ram, in megabytes, that can be required for any one job. **Default**: 16

PYFARM_QUEUE_MAX_RAM
The maximum number of cpus that can be required to any one job. **Default**: 256

PYFARM_QUEUE_DEFAULT_RAM
The default amount of ram, in megabytes, that is required for a job. **Default**: 32

PYFARM_REQUIRE_PRIVATE_IP
Whether pyfarm-master should reject agents with non-private IP addresses **Default**: False

2.3 Master

Environment variables that are used within the server processes on the master.

PYFARM_CONFIG
Controls which configuration should be loaded. Currently the only supported values are *debug* and *prod* and the configuration itself is handled internally.

PYFARM_DATABASE_URI
The URI to connect to the backend database. This should be a valid sqlalchemy uri which looks something like this:

```
dialect+driver://user:password@host/dbname[?key=value..]
```

PYFARM_SECRET_KEY
When present this value is used by forms and the password storage as a seed value for several operations.

PYFARM_CSRF_SESSION_KEY
Key used to set the cross site request forgery key for use by `wtforms`. If not provided this will be set to `PYFARM_SECRET_KEY`

PYFARM_JSON_PRETTY
If set to *true* then all json output by the REST api will be human readable. Setting `PYFARM_CONFIG` to *debug* will also produce the same effect.

PYFARM_API_VERSION
The version of the REST api used for varying points of logic and for constructing `PYFARM_API_PREFIX`

PYFARM_API_PREFIX
If set, this will establish the prefix for mounting the API. This value is combined with `PYFARM_API_VERSION` resulting in something along the lines of:

```
https://$hostname/$PYFARM_API_PREFIX$PYFARM_API_VERSION
```

JOBTYPE_DEFAULT_MAX_BATCH
Performs the same function as `PYFARM_QUEUE_MAX_BATCH` but provides an override specifically for `pyfarm.models.jobtype.JobType.max_batch`

JOBTYPE_DEFAULT_BATCH_CONTIGUOUS
Sets the default value for `pyfarm.models.jobtype.JobType.batch_contiguous`

CHAPTER 3

pyfarm.master package

3.1 Subpackages

3.1.1 pyfarm.master.admin package

Submodules

pyfarm.master.admin.agents module

pyfarm.master.admin.baseview module

pyfarm.master.admin.core module

pyfarm.master.admin.jobtypes module

pyfarm.master.admin.pathmaps module

pyfarm.master.admin.projects module

pyfarm.master.admin.software module

pyfarm.master.admin.tag module

pyfarm.master.admin.users module

pyfarm.master.admin.work module

Module contents

3.1.2 pyfarm.master.api package

Submodules

pyfarm.master.api.agent_updates module

Agent Updates The API allows access to agent update packages, possibly through redirects

class `pyfarm.master.api.agent_updates.`**`AgentUpdatesAPI`**

Bases: `flask.views.MethodView`

`get` (*version*)

A `GET` to this endpoint will return the update package as a zip file the specified version

`GET /api/v1/agents/updates/<string:version> HTTP/1.1`

Request

```
PUT /api/v1/agents/updates/1.2.3 HTTP/1.1
Accept: application/zip
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/zip

<binary data>
```

Statuscode 200 The update file was found and is returned

Statuscode 301 The update can be found under a different URL

Statuscode 400 there was something wrong with the request (such as an invalid version number specified or the mime type not being application/zip)

`methods` = ['GET', 'PUT']

`put` (*version*)

A `PUT` to this endpoint will upload a new version of pyfarm-agent to be used for agent auto-updates. The update must be a zip file.

`PUT /api/v1/agents/updates/<string:version> HTTP/1.1`

Request

```
PUT /api/v1/agents/updates/1.2.3 HTTP/1.1
Content-Type: application/zip

<binary data>
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
```

Statuscode 201 The update was put in place

Statuscode 400 there was something wrong with the request (such as an invalid version number specified or the mime type not being application/zip)

pyfarm.master.api.agents module

Agents Contained within this module are an API handling functions which can manage or query agents using JSON.

class `pyfarm.master.api.agents.`**`AgentIndexAPI`**

Bases: `flask.views.MethodView`

`get``()`

A `GET` to this endpoint will return a list of known agents, with id and name.

`GET /api/v1/agents/ HTTP/1.1`

Request

```
GET /api/v1/agents/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "hostname": "agent1",
        "id": "dd0c6da2-0c91-42cf-a82f-6d503aae43d3"
    },
    {
        "hostname": "agent2",
        "id": "8326779e-90b5-447c-8da8-1eaa154771d9"
    },
    {
        "hostname": "agent3.local",
        "id": "14b28230-64a1-4b62-803e-5fd1baa209e4"
    }
]
```

Request (with filters)

```
GET /api/v1/agents/?min_ram=4096&min_cpus=4 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
[
  {
    "hostname": "foobar",
    "port": 50000,
    "remote_ip": "127.0.0.1",
    "id": "e20bae92-6472-442e-98a8-0ea4c9ee41cd"
  }
]
```

Qparam min_ram If set, list only agents with `min_ram` ram or more

Qparam max_ram If set, list only agents with `max_ram` ram or less

Qparam min_cpus If set, list only agents with `min_cpus` cpus or more

Qparam max_cpus If set, list only agents with `max_cpus` cpus or less

Qparam hostname If set, list only agents matching `hostname`

Qparam remote_ip If set, list only agents matching `remote_ip`

Qparam port If set, list only agents matching `port`.

Statuscode 200 no error, host may or may not have been found

methods = ['GET', 'POST']

post()

A `POST` to this endpoint will either create or update an existing agent. The `port` and `id` columns will determine if an agent already exists.

- If an agent is found matching the `port` and `id` columns from the request the existing model will be updated and the resulting data and the `OK` code will be returned.
- If we don't find an agent matching the `port` and `id` however a new agent will be created and the resulting data and the `CREATED` code will be returned.

Note: The `remote_ip` field is not required and should typically not be included in a request. When not provided `remote_ip` is be populated by the server based off of the ip of the incoming request. Providing `remote_ip` in your request however will override this behavior.

POST /api/v1/agents/ HTTP/1.1

Request

```
POST /api/v1/agents/ HTTP/1.1
Accept: application/json

{
    "cpu_allocation": 1.0,
    "cpus": 14,
    "free_ram": 133,
    "hostname": "agent1",
    "id": "6a0c11df-660f-4c1e-9fb4-5fe2b8cd2437",
    "remote_ip": "10.196.200.115",
    "port": 64994,
    "ram": 2157,
    "ram_allocation": 0.8,
    "state": 8
 }
```

Response (agent created)

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "cpu_allocation": 1.0,
    "cpus": 14,
    "use_address": "remote",
    "free_ram": 133,
    "time_offset": 0,
    "hostname": "agent1",
    "id": "6a0c11df-660f-4c1e-9fb4-5fe2b8cd2437",
    "port": 64994,
    "ram": 2157,
    "ram_allocation": 0.8,
    "state": "online",
    "remote_ip": "10.196.200.115"
}
```

Response (existing agent updated)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "cpu_allocation": 1.0,
    "cpus": 14,
    "use_address": "remote",
    "free_ram": 133,
    "time_offset": 0,
    "hostname": "agent1",
    "id": "6a0c11df-660f-4c1e-9fb4-5fe2b8cd2437",
    "port": 64994,
    "ram": 2157,
    "ram_allocation": 0.8,
    "state": "online",
    "remote_ip": "10.196.200.115"
}
```

Statuscode 201 a new agent was created

Statuscode 200 an existing agent is updated with data from the request

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

class `pyfarm.master.api.agents.`**`SingleAgentAPI`**

Bases: `flask.views.MethodView`

API view which is used for retrieving information about and updating single agents.

`delete`(*agent_id*)

Delete a single agent

`DELETE /api/v1/agents/`(**uuid:** *agent_id*) **`HTTP/1.1`**

Request (agent exists)

```
DELETE /api/v1/agents/b25ee7eb-9586-439a-b131-f5d022e0d403 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO CONTENT
Content-Type: application/json
```

Statuscode 204 the agent was deleted or did not exist

get (*agent_id*)
Return basic information about a single agent

GET /api/v1/agents/ (**str:** *agent_id*) **HTTP/1.1**
Request (agent exists)

```
GET /api/v1/agents/4eefca76-1127-4c17-a3df-c1a7de685541 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "cpu_allocation": 1.0,
    "cpus": 14,
    "use_address": 311,
    "free_ram": 133,
    "time_offset": 0,
    "hostname": "agent1",
    "id": "322360ad-976f-4103-9acc-a811d43fd24d",
    "ip": "10.196.200.115",
    "port": 64994,
    "ram": 2157,
    "ram_allocation": 0.8,
    "state": 202,
    "remote_ip": "10.196.200.115"
 }
```

Request (no such agent)

```
GET /api/v1/agents/4eefca76-1127-4c17-a3df-c1a7de685541 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json

{"error": "Agent `4eefca76-1127-4c17-a3df-c1a7de685541` not "
          "found"}
```

Statuscode 200 no error

Statuscode 400 something within the request is invalid

Statuscode 404 no agent could be found using the given id

methods = ['DELETE', 'GET', 'POST']

post (*agent_id*)

Update an agent's columns with new information by merging the provided data with the agent's current definition in the database.

POST /api/v1/agents/ (**str:** *agent_id*) **HTTP/1.1**

Request

```
POST /api/v1/agents/29d466a5-34f8-408a-b613-e6c2715077a0 HTTP/1.1
Accept: application/json

{"ram": 1234}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "cpu_allocation": 1.0,
    "cpus": 14,
    "use_address": 311,
    "free_ram": 133,
    "time_offset": 0,
    "hostname": "agent1",
    "id": "29d466a5-34f8-408a-b613-e6c2715077a0",
    "ip": "10.196.200.115",
    "port": 64994,
    "ram": 1234,
    "ram_allocation": 0.8,
    "state": "running",
    "remote_ip": "10.196.200.115"
}
```

Statuscode 200 no error

Statuscode 400 something within the request is invalid

Statuscode 404 no agent could be found using the given id

class pyfarm.master.api.agents.**TasksInAgentAPI**

Bases: `flask.views.MethodView`

get (*agent_id*)

A `GET` to this endpoint will return a list of all tasks assigned to this agent.

GET /api/v1/agents/<str:agent_id>/tasks/ HTTP/1.1

Request

```
GET /api/v1/agents/bbf55143-f2b1-4c15-9d41-139bd8057931/tasks/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "state": "assign",
        "priority": 0,
        "job": {
            "jobtype": "TestJobType",
            "id": 1,
            "title": "Test Job",
            "jobtype_version": 1,
            "jobtype_id": 1
        },
        "hidden": false,
        "time_started": null,
        "project_id": null,
        "frame": 2.0
        "agent_id": "bbf55143-f2b1-4c15-9d41-139bd8057931",
        "id": 2,
        "attempts": 2,
        "project": null,
        "time_finished": null,
        "time_submitted": "2014-03-06T15:40:58.338904",
        "job_id": 1
    }
]
```

Statuscode 200 no error

Statuscode 404 agent not found

methods = ['GET', 'POST']

post (*agent_id*)

A POST to this endpoint will assign am existing task to the agent.

POST /api/v1/agents/<str:agent_id>/tasks/ HTTP/1.1

Request

```
POST /api/v1/agents/238d7334-8ca5-4469-9f54-e76c66614a43/tasks/ HTTP/1.1
Accept: application/json

{
    "id": 2
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "agent_id": 1,
    "parents": [],
    "attempts": 2,
    "children": [],
    "job": {
        "title": "Test Job",
        "id": 1
    },
    "project_id": null,
    "agent": {
        "ip": null,
        "hostname": "agent1",
        "port": 50000,
        "id": "238d7334-8ca5-4469-9f54-e76c66614a43"
    },
    "hidden": false,
    "job_id": 1,
    "time_submitted": "2014-03-06T15:40:58.338904",
    "frame": 2.0,
    "priority": 0,
    "state": "assign",
    "time_finished": null,
    "id": 2,
    "project": null,
    "time_started": null
}
```

Statuscode 200 no error

Statuscode 404 agent not found

`pyfarm.master.api.agents.`**`fail_missing_assignments`**`(`*agent*, *current_assignments*`)`

`pyfarm.master.api.agents.`**`or_`**`(`**clauses*`)`

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

pyfarm.master.api.agents.**schema**()

Returns the basic schema of `Agent`

GET /api/v1/agents/schema HTTP/1.1

Request

```
GET /api/v1/agents/schema HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "ram": "INTEGER",
    "free_ram": "INTEGER",
    "time_offset": "INTEGER",
    "use_address": "INTEGER",
    "hostname": "VARCHAR(255)",
    "cpus": "INTEGER",
    "port": "INTEGER",
    "state": "INTEGER",
    "ram_allocation": "FLOAT",
    "cpu_allocation": "FLOAT",
    "id": "UUIDType",
    "remote_ip": "IPv4Address"
}
```

Statuscode 200 no error

pyfarm.master.api.jobqueues module

Job Queues This module defines an API for managing and querying job queues

class pyfarm.master.api.jobqueues.**JobQueueIndexAPI**

Bases: `flask.views.MethodView`

get()

A `GET` to this endpoint will return a list of known job queues.

GET /api/v1/jobqueues/ HTTP/1.1

Request

```
GET /api/v1/jobqueues/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "priority": 5,
        "weight": 10,
        "parent_jobqueue_id": null,
        "name": "Test Queue",
        "minimum_agents": null,
        "id": 1,
        "maximum_agents": null
    },
    {
        "priority": 5,
        "weight": 10,
        "parent_jobqueue_id": null,
        "name": "Test Queue 2",
        "minimum_agents": null,
        "id": 2,
        "maximum_agents": null
    }
]
```

Statuscode 200 no error

methods = ['GET', 'POST']

post()

A POST to this endpoint will create a new job queue.

POST /api/v1/jobqueues/ HTTP/1.1

Request

```
POST /api/v1/jobqueues/ HTTP/1.1
Accept: application/json

{
    "name": "Test Queue"
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "weight": 10,
    "jobs": [],
    "minimum_agents": null,
    "priority": 5,
    "name": "Test Queue",
    "maximum_agents": null,
    "id": 1,
    "parent": null,
    "parent_jobqueue_id": null
}
```

Statuscode 201 a new job queue was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 409 a job queue with that name already exists

class `pyfarm.master.api.jobqueues.`**`SingleJobQueueAPI`**

Bases: `flask.views.MethodView`

`delete`(*queue_rq*)

A `DELETE` to this endpoint will delete the specified job queue

`DELETE /api/v1/jobqueue/HTTP/[<str:name>|<int:id>] 1.1`

Request

```
DELETE /api/v1/jobs/Test%20Queue HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO_CONTENT
```

Statuscode 204 the job queue was deleted or didn't exist

Statuscode 409 the job queue cannot be deleted because it still contains jobs or child queues

`get`(*queue_rq*)

A `GET` to this endpoint will return the requested job queue

`GET /api/v1/jobqueues/[<str:name>|<int:id>] HTTP/1.1`

Request

```
GET /api/v1/software/Test%20Queue HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "parent": [],
    "jobs": [],
    "weight": 10,
    "parent_jobqueue_id": null,
    "priority": 5,
    "minimum_agents": null,
    "name": "Test Queue",
    "maximum_agents": null
}
```

Statuscode 200 no error

Statuscode 404 the requested job queue was not found

methods = ['DELETE', 'GET', 'POST']

post (*queue_rq*)

A `POST` to this endpoint will update the specified queue with the data in the request. Columns not specified in the request will be left as they are.

POST /api/v1/jobqueues/[<str:name>|<int:id>] HTTP/1.1

Request

```
PUT /api/v1/jobs/Test%20Queue HTTP/1.1
Accept: application/json

{
    "priority": 6
}
```

Response

```
HTTP/1.1 201 OK
Content-Type: application/json

{
    "id": 1,
    "parent": [],
    "jobs": [],
    "weight": 10,
    "parent_jobqueue_id": null,
    "priority": 6,
    "minimum_agents": null,
    "name": "Test Queue",
    "maximum_agents": null
}
```

Statuscode 200 the job queue was updated

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

`pyfarm.master.api.jobqueues.`**schema**()

Returns the basic schema of `JobQueue`

GET /api/v1/jobqueues/schema HTTP/1.1

Request

```
GET /api/v1/jobqueues/schema HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": "INTEGER",
    "name": VARCHAR(255)",
    "minimum_agents": "INTEGER",
    "maximum_agents": "INTEGER",
    "priority": "INTEGER",
    "weight": "INTEGER",
    "parent_jobqueue_id": "INTEGER"
}
```

Statuscode 200 no error

pyfarm.master.api.jobs module

Jobs This module defines an API for managing and querying jobs

class pyfarm.master.api.jobs.**JobIndexAPI**
Bases: flask.views.MethodView

get()
A GET to this endpoint will return a list of all jobs.

GET /api/v1/jobs/ HTTP/1.1
Request

```
GET /api/v1/jobs/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "title": "Test Job",
        "state": "queued",
        "id": 1
    },
    {
        "title": "Test Job 2",
        "state": "queued",
        "id": 2
    }
]
```

Statuscode 200 no error

methods = ['GET', 'POST']

post()
A POST to this endpoint will submit a new job.

POST /api/v1/jobs/ HTTP/1.1
Request

```
POST /api/v1/jobs/ HTTP/1.1
Accept: application/json

{
    "end": 2.0,
    "title": "Test Job 2",
    "jobtype": "TestJobType",
    "data": {
        "foo": "bar"
    },
    "software_requirements": [
        {
        "software": "blender"
        }
    ],
    "start": 1.0
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "time_finished": null,
    "time_started": null,
    "end": 2.0,
    "time_submitted": "2014-03-06T15:40:58.335259",
    "jobtype_version": 1,
    "jobtype": "TestJobType",
    "jobqueue": None
    "start": 1.0,
    "priority": 0,
    "state": "queued",
    "parents": [],
    "hidden": false,
    "project_id": null,
    "ram_warning": null,
    "title": "Test Job 2",
    "tags": [],
    "user": null,
    "by": 1.0,
    "data": {
        "foo": "bar"
    },
    "ram_max": null,
    "notes": "",
    "batch": 1,
    "project": null,
    "environ": null,
    "requeue": 3,
    "software_requirements": [
        {
            "min_version": null,
            "max_version": null,
            "max_version_id": null,
            "software_id": 1,
            "min_version_id": null,
            "software": "blender"
        }
    ],
    "id": 2,
    "ram": 32,
    "cpus": 1,
    "children": []
}
```

Statuscode 201 a new job item was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 404 a referenced object, like a software or software version, does not exist

Statuscode 409 a conflicting job already exists

class `pyfarm.master.api.jobs.`**`JobNotifiedUsersIndexAPI`**

Bases: `flask.views.MethodView`

`get` (*job_name*)

A `GET` to this endpoint will return a list of all users to be notified on events in this job.

`GET /api/v1/jobs/[<str:name>|<int:id>]/notified_users/ HTTP/1.1`

Request

```
GET /api/v1/jobs/Test%20Job%202/notified_users/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "id": 1,
        "username": "testuser",
        "email": "testuser@localhost"
    }
]
```

Statuscode 200 no error

Statuscode 404 job not found

`methods` = ['GET', 'POST']

`post` (*job_name*)

A `POST` to this endpoint will add the specified user to the list of notified users for this job.

`POST /api/v1/jobs/[<str:name>|<int:id>]/notified_users/ HTTP/1.1`

Request

```
POST /api/v1/jobs/Test%20Job/notified_users/ HTTP/1.1
Accept: application/json

{
    "username": "testuser"
    "on_success": true,
    "on_failure": true,
    "on_deletion": false
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 1
    "username": "testuser"
    "email": "testuser@example.com"
}
```

Statuscode 201 a new notified user entry was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 404 the job or the specified user does not exist

class `pyfarm.master.api.jobs.`**`JobSingleNotifiedUserAPI`**

Bases: `flask.views.MethodView`

`delete`(*job_name*, *username*)

A `DELETE` to this endpoint will remove the specified user from the list of notified users for this job.

`DELETE /api/v1/jobs/[<str:name>|<int:id>]/notified_users/<str:username> HTTP/1.1`

Request

```
DELETE /api/v1/jobs/Test%20Job/notified_users/testuser HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO_CONTENT
```

Statuscode 204 the notified user was removed from this job or wasn't in the list in the first place

Statuscode 404 the job or the specified user does not exist

`methods` = ['DELETE']

class `pyfarm.master.api.jobs.`**`JobSingleTaskAPI`**

Bases: `flask.views.MethodView`

get (*job_name*, *task_id*)

A GET to this endpoint will return the requested task

GET /api/v1/jobs/[<str:name>|<int:id>]/tasks/<int:task_id> HTTP/1.1

Request

```
GET /api/v1/jobs/Test%20Job%202/tasks/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "time_finished": null,
    "agent": null,
    "attempts": 0,
    "frame": 2.0,
    "agent_id": null,
    "job": {
        "id": 1,
        "title": "Test Job"
    },
    "time_started": null,
    "state": "running",
    "project_id": null,
    "id": 2,
    "time_submitted": "2014-03-06T15:40:58.338904",
    "project": null,
    "parents": [],
    "job_id": 1,
    "hidden": false,
    "children": [],
    "priority": 0
}
```

Statuscode 200 no error

methods = ['GET', 'POST']

post (*job_name*, *task_id*)

A POST to this endpoint will update the specified task with the data in the request. Columns not specified in the request will be left as they are. The agent will use this endpoint to inform the master of its progress.

POST /api/v1/jobs/[<str:name>|<int:id>]/tasks/<int:task_id> HTTP/1.1

Request

```
PUT /api/v1/job/Test%20Job/tasks/1 HTTP/1.1
Accept: application/json

{
    "state": "running"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "time_finished": null,
    "agent": null,
    "attempts": 0,
    "failures": 0,
    "frame": 2.0,
    "agent_id": null,
    "job": {
        "id": 1,
        "title": "Test Job"
    },
    "time_started": null,
    "state": "running",
    "project_id": null,
    "id": 2,
    "time_submitted": "2014-03-06T15:40:58.338904",
    "project": null,
    "parents": [],
    "job_id": 1,
    "hidden": false,
    "children": [],
    "priority": 0
}
```

Statuscode 200 the task was updated

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

class `pyfarm.master.api.jobs.`**`JobTasksIndexAPI`**

Bases: `flask.views.MethodView`

`get` (*job_name*)

A `GET` to this endpoint will return a list of all tasks in a job.

`GET /api/v1/jobs/[<str:name>|<int:id>]/tasks HTTP/1.1`

Request

```
GET /api/v1/jobs/Test%20Job%202/tasks/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "hidden": false,
        "id": 3,
        "attempts": 0,
        "priority": 0,
        "time_started": null,
        "time_submitted": "2014-03-06T15:49:51.892228",
        "frame": 1.0,
        "time_finished": null,
        "job_id": 2,
        "project_id": null,
        "state": "queued",
        "agent_id": null
    },
    {
        "hidden": false,
        "id": 4,
        "attempts": 0,
        "priority": 0,
        "time_started": null,
        "time_submitted": "2014-03-06T15:49:51.892925",
        "frame": 2.0,
        "time_finished": null,
        "job_id": 2,
        "project_id": null,
        "state": "queued",
        "agent_id": null
    }
]
```

Statuscode 200 no error

methods = ['GET']

exception `pyfarm.master.api.jobs.`**`ObjectNotFound`**

Bases: `Exception`

class `pyfarm.master.api.jobs.`**`SingleJobAPI`**

Bases: `flask.views.MethodView`

`delete`(*job_name*)

A `DELETE` to this endpoint will mark the specified job for deletion and remove it after stopping and removing all of its tasks.

`DELETE /api/v1/jobs/[<str:name>|<int:id>] HTTP/1.1`

Request

```
DELETE /api/v1/jobs/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO_CONTENT
```

Statuscode 204 the specified job was marked for deletion

Statuscode 404 the job does not exist

get (*job_name*)

A `GET` to this endpoint will return the specified job, by name or id.

GET /api/v1/jobs/[<str:name>|<int:id>] HTTP/1.1

Request

```
GET /api/v1/jobs/Test%20Job%202 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "ram_warning": null,
    "title": "Test Job",
    "state": "queued",
    "jobtype_version": 1,
    "jobtype": "TestJobType",
    "environ": null,
    "user": null,
    "priority": 0,
    "time_finished": null,
    "start": 2.0,
    "id": 1,
    "notes": "",
    "notified_users": []
    "ram": 32,
    "tags": [],
    "hidden": false,
    "data": {
        "foo": "bar"
    },
    "software_requirements": [
        {
            "software": "blender",
            "software_id": 1,
            "min_version": null,
            "max_version": null,
            "min_version_id": null,
            "max_version_id": null
        }
    ],
    "batch": 1,
    "time_started": null,
    "time_submitted": "2014-03-06T15:40:58.335259",
    "requeue": 3,
    "end": 4.0,
    "parents": [],
    "cpus": 1,
    "ram_max": null,
    "children": [],
    "by": 1.0,
    "project_id": null
}
```

Statuscode 200 no error

Statuscode 404 job not found

methods = ['DELETE', 'GET', 'POST']

post (*job_name*)

A POST to this endpoint will update the specified job with the data in the request. Columns not specified in the request will be left as they are. If the "start", "end" or "by" columns are updated, tasks will be created or deleted as required.

POST /api/v1/jobs/[<str:name>|<int:id>] HTTP/1.1

Request

```
PUT /api/v1/jobs/Test%20Job HTTP/1.1
Accept: application/json

{
    "start": 2.0
}
```

Response

```
HTTP/1.1 201 OK
Content-Type: application/json

{
    "end": 4.0,
    "children": [],
    "jobtype_version": 1,
    "jobtype": "TestJobType",
    "time_started": null,
    "tasks_failed": [],
    "project_id": null,
    "id": 1,
    "software_requirements": [
        {
            "software": "blender",
            "min_version": null,
            "max_version_id": null,
            "software_id": 1,
            "max_version": null,
            "min_version_id": null
        }
    ],
    "tags": [],
    "environ": null,
    "requeue": 3,
    "start": 2.0,
    "ram_warning": null,
    "title": "Test Job",
    "batch": 1,
    "time_submitted": "2014-03-06T15:40:58.335259",
    "ram_max": null,
    "user": null,
    "notes": "",
    "data": {
        "foo": "bar"
    },
    "ram": 32,
    "parents": [],
    "hidden": false,
    "priority": 0,
    "cpus": 1,
    "state": "queued",
    "by": 1.0,
    "time_finished": null
}
```

Statuscode 200 the job was updated

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

`pyfarm.master.api.jobs.`**`and_`**`(`*`*clauses`*`)`

Produce a conjunction of expressions joined by `AND`.

E.g.:

```
from sqlalchemy import and_

stmt = select([users_table]).where(
                and_(
                    users_table.c.name == 'wendy',
                    users_table.c.enrolled == True
                )
            )
```

The `and_()` conjunction is also available using the Python `&` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') &
                (users_table.c.enrolled == True)
            )
```

The `and_()` operation is also implicit in some cases; the `Select.where()` method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using `and_()`:

```
stmt = select([users_table]).\
            where(users_table.c.name == 'wendy').\
            where(users_table.c.enrolled == True)
```

See also:

`or_()`

`pyfarm.master.api.jobs.`**`or_`**`(`*`*clauses`*`)`

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

`pyfarm.master.api.jobs.`**`parse_requirements`**`(`*requirements*`)`

Takes a list dicts specifying a software and optional min- and max-versions and returns a list of `JobRequirement` objects.

Raises TypeError if the input was not as expected or ObjectNotFound if a referenced software of or version was not found.

Parameters **`requirements`** (*list*) – A list of of dicts specifying a software and optionally min_version and/or max_version.

Raises

- **`TypeError`** – Raised if `requirements` is not a list or if an entry in `requirements` is not a dictionary.
- **`ValueError`** – Raised if there's a problem with the content of at least one of the requirement dictionaries.
- **`ObjectNotFound`** – Raised if the referenced software or version was not found

`pyfarm.master.api.jobs.`**`schema`**`()`

Returns the basic schema of `Job`

`GET /api/v1/jobs/schema HTTP/1.1`

Request

```
GET /api/v1/jobs/schema HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "batch": "INTEGER",
    "by": "NUMERIC(10, 4)",
    "cpus": "INTEGER",
    "data": "JSONDict",
    "end": "NUMERIC(10,4)",
    "environ": "JSONDict",
    "hidden": "BOOLEAN",
    "id": "INTEGER",
    "jobtype": "VARCHAR(64)",
    "jobtype_version": "INTEGER",
    "jobqueue": "VARCHAR(255)",
    "notes": "TEXT",
    "priority": "INTEGER",
    "project_id": "INTEGER",
    "ram": "INTEGER",
    "ram_max": "INTEGER",
    "ram_warning": "INTEGER",
    "requeue": "INTEGER",
    "start": "NUMERIC(10,4)",
    "state": "WorkStateEnum",
    "time_finished": "DATETIME",
    "time_started": "DATETIME",
    "time_submitted": "DATETIME",
    "title": "VARCHAR(255)",
    "user": "VARCHAR(255)"
}
```

Statuscode 200 no error

pyfarm.master.api.jobtypes module

Jobtypes This module defines an API for managing and querying jobtypes

class `pyfarm.master.api.jobtypes.`**`JobTypeCodeAPI`**
Bases: `flask.views.MethodView`

`get`(*jobtype_name*, *version*)
A `GET` to this endpoint will return just the python code for this version of the specified jobtype.

`GET /api/v1/jobtypes/[<str:name>|<int:id>]/versions/<int:version>/code HTTP/1.1`

Request

```
GET /api/v1/jobtypes/TestJobType/versions/1/code HTTP/1.1
Accept: text/x-python
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/x-python

from pyfarm.jobtypes.core.jobtype import JobType

class TestJobType(JobType):
    def get_command(self):
        return "/usr/bin/touch"
    def get_arguments(self):
        return [os.path.join(
            self.assignment_data["job"]["data"]["path"], "%04d" %
            self.assignment_data["tasks"][0]["frame"])]
```

Statuscode 200 no error

Statuscode 404 jobtype or version not found

methods = ['GET']

class `pyfarm.master.api.jobtypes.`**`JobTypeIndexAPI`**

Bases: `flask.views.MethodView`

`get``()`

A `GET` to this endpoint will return a list of registered jobtypes.

`GET /api/v1/jobtypes/ HTTP/1.1`

Request

```
GET /api/v1/jobtypes/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "id": 1,
        "name": "TestJobType"
    }
]
```

Statuscode 200 no error

methods = ['GET', 'POST']

`post``()`

A `POST` to this endpoint will create a new jobtype.

`POST /api/v1/jobtypes/ HTTP/1.1`

Request

```
POST /api/v1/jobtypes/ HTTP/1.1
Accept: application/json

{
    "name": "TestJobType",
    "classname": "TestJobType",
    "description": "Jobtype for testing inserts and queries",
    "code": "\nfrom pyfarm.jobtypes.core.jobtype import "
            "JobType\n\nclass TestJobType(JobType):\n"
            "    def get_command(self):\n"
```

```
                "        return "/usr/bin/touch"\n\n"
                "    def get_arguments(self):\n"
                "           return [os.path.join("
                "self.assignment_data["job"]["data"]["path"], "
                ""%04d" % self.assignment_data["tasks"]"
                "[0]["frame"])]\n"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "batch_contiguous": true,
    "software_requirements": [],
    "version": 1,
    "max_batch": 1,
    "name": "TestJobType",
    "classname": "TestJobType",
    "description": "Jobtype for testing inserts and queries",
    "code": "\nfrom pyfarm.jobtypes.core.jobtype import "
            "JobType\n\nclass TestJobType(JobType):\n"
            "    def get_command(self):\n"
            "        return "/usr/bin/touch"\n\n"
            "    def get_arguments(self):\n"
            "           return [os.path.join("
            "self.assignment_data["job"]["data"]["path"], "
            ""%04d" % self.assignment_data["tasks"]"
            "[0]["frame"])]\n"
}
```

Statuscode 201 a new jobtype item was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 409 a conflicting jobtype already exists

class `pyfarm.master.api.jobtypes.`**`JobTypeSoftwareRequirementAPI`**

Bases: `flask.views.MethodView`

`delete`(*jobtype_name*, *software*)

A `DELETE` to this endpoint will delete the requested software requirement from the specified jobtype, creating a new version of the jobtype in the process

`DELETE /api/v1/jobtypes/[<str:name>|<int:id>]/software_requirements/<int:id> HTTP/1`

Request

```
DELETE /api/v1/jobtypes/TestJobType/software_requirements/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO CONTENT
```

Statuscode 204 the software requirement was deleted or didn't exist

get (*jobtype_name*, *software*)

A GET to this endpoint will return the specified software requirement from the newest version of the requested jobtype.

GET /api/v1/jobtypes/[<str:name>|<int:id>]/software_requirements/<int:id> HTTP/1.1

Request

```
GET /api/v1/jobtypes/TestJobType/software_requirements/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "software": {
        "software": "/bin/touch",
        "id": 1
    },
    "max_version": null,
    "min_version": {
        "version": "8.21",
        "id": 1
    },
    "jobtype_version": {
        "version": 7,
        "jobtype": "TestJobType"
    }
}
```

Statuscode 200 no error

Statuscode 404 jobtype or software requirement not found

methods = ['DELETE', 'GET']

class pyfarm.master.api.jobtypes.**JobTypeSoftwareRequirementsIndexAPI**

Bases: flask.views.MethodView

get (*jobtype_name*, *version=None*)

A GET to this endpoint will return a list of all the software requirements of the specified jobtype

GET /api/v1/jobtypes/[<str:name>|<int:id>]/software_requirements/ HTTP/1.1

Request

```
GET /api/v1/jobtypes/TestJobType/software_requirements/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "software": {
            "software": "/bin/touch",
            "id": 1
        },
        "max_version": null,
        "min_version": {
            "version": "8.21",
            "id": 1
        },
        "jobtype_version": {
            "version": 7,
            "jobtype": "TestJobType"
        }
    }
]
```

Statuscode 200 no error

Statuscode 404 jobtype or version not found

methods = ['GET', 'POST']

post (*jobtype_name*, *version=None*)

A `POST` to this endpoint will create a new software_requirement for the specified jobtype. This will transparently create a new jobtype version

POST /api/v1/jobtypes/[<str:name>|<int:id>]/software_requirements/ HTTP/1.1

Request

```
POST /api/v1/jobtypes/TestJobType/software_requirements/ HTTP/1.1
Accept: application/json

{
    "software": "blender",
    "min_version": "2.69"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "jobtype_version": {
        "id": 8,
        "jobtype": "TestJobType",
        "version": 7
    },
    "max_version": null,
    "min_version": {
        "id": 2,
        "version": "1.69"
    },
    "software": {
        "id": 2,
        "software": "blender"
    }
}
```

Statuscode 201 a new software requirement was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 405 you tried calling this method on a specific version

Statuscode 409 a conflicting software requirement already exists

class `pyfarm.master.api.jobtypes.`**`JobTypeVersionsIndexAPI`**

Bases: `flask.views.MethodView`

`get` (*jobtype_name*)

A `GET` to this endpoint will return a sorted list of of all known versions of the specified jobtype.

`GET /api/v1/jobtypes/[<str:name>|<int:id>]/versions/ HTTP/1.1`

Request

```
GET /api/v1/jobtypes/TestJobType/versions/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[1, 2]
```

Statuscode 200 no error

Statuscode 404 jobtype not found

`methods` = ['GET']

exception `pyfarm.master.api.jobtypes.`**`ObjectNotFound`**

Bases: `Exception`

class `pyfarm.master.api.jobtypes.`**`SingleJobTypeAPI`**

Bases: `flask.views.MethodView`

`delete` (*jobtype_name*)

A `DELETE` to this endpoint will delete the requested jobtype

`DELETE /api/v1/jobtypes/[<str:name>|<int:id>] HTTP/1.1`

Request

```
DELETE /api/v1/jobtypes/TestJobType HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO CONTENT
```

Statuscode 204 the jobtype was deleted or didn't exist

get (*jobtype_name*)

A GET to this endpoint will return the most recent version of the referenced jobtype, by name or id.

GET /api/v1/jobtypes/<str:tagname> HTTP/1.1

Request

```
GET /api/v1/jobtypes/TestJobType HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "batch_contiguous": true,
    "classname": null,
    "code": "\nfrom pyfarm.jobtypes.core.jobtype import "
            "JobType\n\nclass TestJobType(JobType):\n"
            "    def get_command(self):\n"
            "        return "/usr/bin/touch"\n\n"
            "    def get_arguments(self):\n"
            "           return [os.path.join("
            "self.assignment_data["job"]["data"]["path"], "
            ""%04d" % self.assignment_data["tasks"]"
            "[0]["frame"])]\n",
    "id": 1,
    "version": 1,
    "max_batch": 1,
    "name": "TestJobType",
    "software_requirements": [
        {
            "max_version": null,
            "max_version_id": null,
            "min_version": "8.21",
            "min_version_id": 1,
            "software": "/bin/touch",
            "software_id": 1
        }
    ]
}
```

Statuscode 200 no error

Statuscode 404 jobtype or version not found

methods = ['DELETE', 'GET', 'PUT']

put (*jobtype_name*)

A PUT to this endpoint will create a new jobtype under the given URI. If a jobtype already exists under that URI, a new version will be created with the given data.

You should only call this by id for updating an existing jobtype or if you have a reserved jobtype id. There is currently no way to reserve a jobtype id.

PUT /api/v1/jobtypes/[<str:name>|<int:id>] HTTP/1.1

Request

```
PUT /api/v1/jobtypes/TestJobType HTTP/1.1
Accept: application/json

{
    "name": "TestJobType",
    "description": "Jobtype for testing inserts and queries",
    "code": "\nfrom pyfarm.jobtypes.core.jobtype import "
            "JobType\n\nclass TestJobType(JobType):\n"
            "    def get_command(self):\n"
            "        return "/usr/bin/touch"\n\n"
            "    def get_arguments(self):\n"
            "           return [os.path.join("
            "self.assignment_data["job"]["data"]["path"], "
            ""%04d" % self.assignment_data["tasks"]"
            "[0]["frame"])]\n"
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "batch_contiguous": true,
    "classname": null,
    "code": "\nfrom pyfarm.jobtypes.core.jobtype import "
            "JobType\n\nclass TestJobType(JobType):\n"
            "    def get_command(self):\n"
            "        return "/usr/bin/touch"\n\n"
            "    def get_arguments(self):\n"
            "           return [os.path.join("
            "self.assignment_data["job"]["data"]["path"], "
            ""%04d" % self.assignment_data["tasks"]"
            "[0]["frame"])]\n",
    "id": 1,
    "max_batch": 1,
    "name": "TestJobType",
    "description": "Jobtype for testing inserts and queries",
    "software_requirements": []
}
```

Statuscode 201 a new jobtype was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

class `pyfarm.master.api.jobtypes.`**`VersionedJobTypeAPI`**

Bases: `flask.views.MethodView`

`delete` (*jobtype_name*, *version*)

A `DELETE` to this endpoint will delete the requested version of the specified jobtype.

`DELETE /api/v1/jobtypes/[<str:name>|<int:id>]/versions/<int:version> HTTP/1.1`

Request

```
DELETE /api/v1/jobtypes/TestJobType/versions/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO CONTENT
```

Statuscode 204 the version was deleted or didn't exist

`get` (*jobtype_name*, *version*)

A `GET` to this endpoint will return the specified version of the referenced jobtype, by name or id.

`GET /api/v1/jobtypes/[<str:name>|<int:id>]/versions/<int:version> HTTP/1.1`

Request

```
GET /api/v1/jobtypes/TestJobType/versions/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "batch_contiguous": true,
    "classname": null,
    "name": "TestJobType",
    "code": "\nfrom pyfarm.jobtypes.core.jobtype import "
            "JobType\n\nclass TestJobType(JobType):\n"
            "    def get_command(self):\n"
            "        return "/usr/bin/touch"\n\n"
            "    def get_arguments(self):\n"
            "           return [os.path.join("
            "self.assignment_data["job"]["data"]["path"], "
            ""%04d" % self.assignment_data["tasks"]"
            "[0]["frame"])]\n",
    "id": 1,
    "version": 1,
    "max_batch": 1,
    "software_requirements": [
        {
            "max_version": null,
            "max_version_id": null,
            "min_version": "8.21",
            "min_version_id": 1,
            "software": "/bin/touch",
            "software_id": 1
        }
    ]
}
```

Statuscode 200 no error

Statuscode 404 jobtype or version not found

methods = ['DELETE', 'GET']

`pyfarm.master.api.jobtypes.`**`or_`**`(`**clauses*`)`

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

`pyfarm.master.api.jobtypes.`**`parse_requirements`**`(`*requirements*`)`

Takes a list dicts specifying a software and optional min- and max-versions and returns a list of `JobRequirement` objects.

Raises TypeError if the input was not as expected or ObjectNotFound if a referenced software of or version was not found.

Parameters **`requirements`** (*list*) – A list of of dicts specifying a software and optionally min_version and/or max_version.

Raises

- **`TypeError`** – Raised if `requirements` is not a list or if an entry in `requirements` is not a dictionary.
- **`ValueError`** – Raised if there's a problem with the content of at least one of the requirement dictionaries.

- **ObjectNotFound** – Raised if the referenced software or version was not found

pyfarm.master.api.jobtypes.**schema**()

Returns the basic schema of JobType

GET /api/v1/jobtypes/schema HTTP/1.1

Request

```
GET /api/v1/jobtypes/schema HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "batch_contiguous": "BOOLEAN",
    "classname": "VARCHAR(64)",
    "code": "TEXT",
    "description": "TEXT",
    "id": "INTEGER",
    "version": "INTEGER",
    "max_batch": "INTEGER",
    "name": "VARCHAR(64)"
}
```

Statuscode 200 no error

pyfarm.master.api.pathmaps module

Path Maps API endpoints for viewing and managing path maps

class pyfarm.master.api.pathmaps.**PathMapIndexAPI**

Bases: flask.views.MethodView

get()

A GET to this endpoint will return a list of all registered path maps, with id. It can be made with a for_agent query parameter, in which case it will return only those path maps that apply to that agent.

GET /api/v1/pathmaps/ HTTP/1.1

Request

```
GET /api/v1/pathmaps/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "id": 1,
        "path_osx": "/mnt/nfs",
        "path_windows": "\\domains\cifs_server",
        "path_linux": "/mnt/nfs"
    },
    {
        "id": 7,
        "path_osx": "/renderout",
        "path_windows": "c:\renderout",
        "path_linux": "/renderout"
        "tag": "usual",
    }
]
```

Statuscode 200 no error

methods = ['GET', 'POST']

post ()

A POST to this endpoint will create a new path map.

A path map will list the equivalent path prefixes for all three supported families of operating systems, Linux, Windows and OS X. A path map can optionally be restricted to one tag, in which case it will only apply to agents with that tag. If a tag is specified that does not exist yet, that tag will be transparently created.

POST /api/v1/pathmaps/ HTTP/1.1

Request

```
POST /api/v1/pathmaps/ HTTP/1.1
Accept: application/json

{
    "path_linux": "/mnt/nfs",
    "path_windows": "\domain\cifs_server",
    "path_osx": "/mnt/nfs",
    "tag": "production"
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 1,
    "path_linux": "/mnt/nfs",
    "path_windows": "\domain\cifs_server",
    "path_osx": "/mnt/nfs",
    "tag": "production"
}
```

Statuscode 201 a new pathmap was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

class `pyfarm.master.api.pathmaps.`**`SinglePathMapAPI`**

Bases: `flask.views.MethodView`

`delete`(*pathmap_id*)

A `DELETE` to this endpoint will remove the specified pathmap

`DELETE /api/v1/pathmaps/<int:pathmap_id> HTTP/1.1`

Request

```
DELETE /api/v1/pathmaps/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO_CONTENT
```

Statuscode 204 the path map was deleted or did not exist in the first place

`get`(*pathmap_id*)

A `GET` to this endpoint will return a single path map specified by pathmap_id

`GET /api/v1/pathmaps/<int:pathmap_id> HTTP/1.1`

Request

```
GET /api/v1/pathmaps/1 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "path_osx": "/mnt/nfs",
    "path_windows": "\\domains\cifs_server",
    "path_linux": "/mnt/nfs"
}
```

Statuscode 200 no error

`methods` **= ['DELETE', 'GET', 'POST']**

`post`(*pathmap_id*)

A `POST` to this endpoint will update an existing path map with new values.

Only the values included in the request will be updated. The rest will be left unchanged. The id column cannot be changed. Including it in the request will lead to an error.

`POST /api/v1/pathmaps/<int:pathmap_id> HTTP/1.1`

Request

```
POST /api/v1/pathmaps/1 HTTP/1.1
Accept: application/json

{
    "path_linux": "/mnt/smb"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "path_linux": "/mnt/smb",
    "path_windows": "\domain\cifs_server",
    "path_osx": "/mnt/nfs",
    "tag": "production"
}
```

Statuscode 200 the specified pathmap was updated

Statuscode 404 the specified pathmap does not exist

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

`pyfarm.master.api.pathmaps.`**`or_`**`(`*`*clauses`*`)`

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

`pyfarm.master.api.pathmaps.`**`schema`**`()`

Returns the basic schema of `Agent`

GET /api/v1/pathmaps/schema HTTP/1.1

Request

```
GET /api/v1/pathmaps/schema HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": "INTEGER",
    "path_linux": "VARCHAR(512)",
    "path_windows": "VARCHAR(512)",
    "path_osx": "VARCHAR(512)",
    "tag": "VARCHAR(64)"
}
```

Statuscode 200 no error

pyfarm.master.api.software module

Software Contained within this module are an API handling functions which can manage or query software items using JSON.

class `pyfarm.master.api.software.`**`SingleSoftwareAPI`**

Bases: `flask.views.MethodView`

`delete`(*software_rq*)

A `DELETE` to this endpoint will delete the requested software tag

DELETE /api/v1/software/<str:softwarename> HTTP/1.1

Request

```
DELETE /api/v1/software/Autodesk%20Maya HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO_CONTENT
```

Statuscode 204 the software tag was deleted or didn't exist

`get`(*software_rq*)

A `GET` to this endpoint will return the requested software tag

GET /api/v1/software/<str:softwarename> HTTP/1.1

Request

```
GET /api/v1/software/Autodesk%20Maya HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "software": "Autodesk Maya",
    "id": 1,
    "versions": [
        {
            "version": "2013",
            "id": 1,
            "rank": 100
        },
        {
            "version": "2014",
            "id": 2,
            "rank": 200
        }
    ]
}
```

Statuscode 200 no error

Statuscode 404 the requested software tag was not found

methods = ['DELETE', 'GET', 'PUT']

put (*software_rq*)

A PUT to this endpoint will create a new software tag under the given URI or update an existing software tag if one exists. Renaming existing software tags via this call is supported, but when creating new ones, the included software name must be equal to the one in the URI.

You should only call this by id for overwriting an existing software tag or if you have a reserved software id. There is currently no way to reserve a tag id.

PUT /api/v1/software/<str:softwarename> HTTP/1.1

Request

```
PUT /api/v1/software/blender HTTP/1.1
Accept: application/json

{
    "software": "blender"
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 4,
    "software": "blender",
    "versions": []
}
```

Request

```
PUT /api/v1/software/blender HTTP/1.1
Accept: application/json

{
    "software": "blender",
    "version": [
        {
            "version": "1.69"
        }
    ]
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 4,
    "software": "blender",
    "versions": [
        {
            "version": "1.69",
            "id": 1,
            "rank": 100
        }
    ]
}
```

Statuscode 200 an existing software tag was updated

Statuscode 201 a new software tag was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

class `pyfarm.master.api.software.`**`SingleSoftwareVersionAPI`**

Bases: `flask.views.MethodView`

`delete`(*software_rq*, *version_name*)

A `DELETE` to this endpoint will delete the requested software version

`DELETE /api/v1/software/<str:softwarename>/versions/<str:version> HTTP/1.1`

Request

```
DELETE /api/v1/software/Autodesk%20Maya/versions/2013 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 204 NO_CONTENT
```

Statuscode 204 the software version was deleted or didn't exist

Statuscode 404 the software specified does not exist

get (*software_rq*, *version_name*)
A GET to this endpoint will return the specified version

GET /api/v1/software/<str:softwarename>/versions/<str:version> HTTP/1.1
Request

```
GET /api/v1/software/Autodesk%20Maya/versions/2014 HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "version": "2013",
    "id": 1,
    "rank": 100
}
```

Statuscode 200 no error

Statuscode 404 the requested software tag or version was not found

methods = ['DELETE', 'GET']

class pyfarm.master.api.software.**SoftwareIndexAPI**
Bases: flask.views.MethodView

get ()
A GET to this endpoint will return a list of known software, with all known versions.

GET /api/v1/software/ HTTP/1.1
Request

```
GET /api/v1/software/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "software": "Houdini",
        "id": 1,
        "versions": [
            {
                "version": "13.0.1",
                "id": 1,
                "rank": 100
            }
        ]
    }
]
```

Statuscode 200 no error

methods = ['GET', 'POST']

post()

A POST to this endpoint will create a new software tag.

A list of versions can be included. If the software item already exists the listed versions will be added to the existing ones. Versions with no explicit rank are assumed to be the newest version available. Users should not mix versions with an explicit rank with versions without one.

POST /api/v1/software/ HTTP/1.1

Request

```
POST /api/v1/software/ HTTP/1.1
Accept: application/json

{
    "software": "blender"
}
```

Response (new software item create)

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 4,
    "software": "blender",
    "versions": []
}
```

Statuscode 201 a new software item was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 409 a software tag with that name already exists

class pyfarm.master.api.software.**SoftwareVersionsIndexAPI**

Bases: flask.views.MethodView

get(*software_rq*)

A GET to this endpoint will list all known versions for this software

GET /api/v1/software/<str:softwarename>/versions/ HTTP/1.1

Request

```
GET /api/v1/software/Autodesk%20Maya/versions/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "version": "2013",
        "id": 1,
        "rank": 100
    },
    {
        "version": "2014",
        "id": 2,
        "rank": 200
    }
]
```

Statuscode 200 no error

Statuscode 404 the requested software tag was not found

methods = ['GET', 'POST']

post (*software_rq*)

A POST to this endpoint will create a new version for this software.

A rank can optionally be included. If it isn't, it is assumed that this is the newest version for this software

POST /api/v1/software/versions/ HTTP/1.1

Request

```
POST /api/v1/software/blender/versions/ HTTP/1.1
Accept: application/json

{
    "version": "1.70"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 4,
    "version": "1.70",
    "rank": "100"
}
```

Statuscode 201 a new software version was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 409 a software version with that name already exists

exception `pyfarm.master.api.software.`**`VersionParseError`**

Bases: `Exception`

Raised by `extract_version_dicts()` when the function is unable to parse a version.

`pyfarm.master.api.software.`**`extract_version_dicts`**`(`*json_in*`)`

Extracts and returns a list of versions from `json_in`.

`pyfarm.master.api.software.`**`schema`**`()`

Returns the basic schema of `Software`

`GET /api/v1/software/schema HTTP/1.1`

Request

```
GET /api/v1/software/schema HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": "INTEGER",
    "software": "VARCHAR(64)"
}
```

Statuscode 200 no error

pyfarm.master.api.tags module

Tag Contained within this module are an API handling functions which can manage or query tags using JSON.

class `pyfarm.master.api.tags.`**`AgentsInTagIndexAPI`**

Bases: `flask.views.MethodView`

`get``(`*tagname=None*`)`

A `GET` to this endpoint will list all agents associated with this tag.

`GET /api/v1/tags/<str:tagname>/agents/ HTTP/1.1`

Request

```
GET /api/v1/tags/interesting/agents/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

[
    {
        "hostname": "agent3",
        "id": 1,
        "href": "/api/v1/agents/1
    }
]
```

Statuscode 200 the list of agents associated with this tag is returned

Statuscode 404 the tag specified does not exist

methods = ['GET', 'POST']

post (*tagname=None*)

A POST will add an agent to the list of agents tagged with this tag The tag can be given as a string or as an integer (its id).

POST /api/v1/tags/<str:tagname>/agents/ HTTP/1.1

Request

```
POST /api/v1/tags/interesting/agents/ HTTP/1.1
Accept: application/json

{
    "agent_id": "dd0c6da2-0c91-42cf-a82f-6d503aae43d3"
}
```

Response (agent newly tagged)

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "href": "/api/v1/agents/1",
    "id": 1
}
```

Request

```
POST /api/v1/tags/interesting/agents/ HTTP/1.1
Accept: application/json

{
    "agent_id": "dd0c6da2-0c91-42cf-a82f-6d503aae43d3"
}
```

Response (agent already had that tag)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "href": "/api/v1/agents/1",
    "id": 1
}
```

Statuscode 200 an existing tag was found and returned

Statuscode 201 a new tag was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 404 either the tag or the referenced agent does not exist

class `pyfarm.master.api.tags.`**`SingleTagAPI`**

Bases: `flask.views.MethodView`

`delete`(*tagname=None*)

A `DELETE` to this endpoint will delete the tag under this URI, including all relations to tags or jobs.

`DELETE /api/v1/tags/<str:tagname> HTTP/1.1`

Request

```
DELETE /api/v1/tags/interesting HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 1,
    "tag": "interesting"
}
```

Statuscode 204 the tag was deleted or did not exist in the first place

`get`(*tagname=None*)

A `GET` to this endpoint will return the referenced tag, either by name or id, including a list of agents and jobs associated with it.

`GET /api/v1/tags/<str:tagname> HTTP/1.1`

Request

```
GET /api/v1/tags/interesting HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "agents": [{
        "hostname": "agent3",
        "href": "/api/v1/agents/94522b7e-817b-4358-95da-670b31aad624",
        "id": 1
    }],
    "id": 1,
    "jobs": [],
    "tag": "interesting"
}
```

Statuscode 200 no error

Statuscode 404 tag not found

methods = ['DELETE', 'GET', 'PUT']

put (*tagname=None*)

A PUT to this endpoint will create a new tag under the given URI. If a tag already exists under that URI, it will be deleted, then recreated. Note that when overwriting a tag like that, all relations that are not explicitly specified here will be deleted You can optionally specify a list of agents or jobs relations as integers in the request data.

You should only call this by id for overwriting an existing tag or if you have a reserved tag id. There is currently no way to reserve a tag id.

PUT /api/v1/tags/<str:tagname> HTTP/1.1

Request

```
PUT /api/v1/tags/interesting HTTP/1.1
Accept: application/json

{
    "tag": "interesting"
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 1,
    "tag": "interesting"
}
```

Request

```
PUT /api/v1/tags/interesting HTTP/1.1
Accept: application/json

```

```
{
    "tag": "interesting",
    "agents": [1]
    "jobs": []
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 1,
    "tag": "interesting"
}
```

Statuscode 201 a new tag was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 404 a referenced agent or job does not exist

class `pyfarm.master.api.tags.`**`TagIndexAPI`**

Bases: `flask.views.MethodView`

`get``()`

A `GET` to this endpoint will return a list of known tags, with id. Associated agents and jobs are included for every tag

:rtype : object .. http:get:: /api/v1/tags/ HTTP/1.1

Request

```
GET /api/v1/tags/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {
        "agents": [
            1
        ],
        "jobs": [],
        "id": 1,
        "tag": "interesting"
    },
    {
        "agents": [],
        "jobs": [],
        "id": 2,
        "tag": "boring"
    }
]
```

Statuscode 200 no error

methods = ['GET', 'POST']

post ()

A POST to this endpoint will do one of two things:

- create a new tag and return the row
- return the row for an existing tag

Tags only have one column, the tag name. Two tags are automatically considered equal if the tag names are equal.

POST /api/v1/tags/ HTTP/1.1

Request

```
POST /api/v1/tags/ HTTP/1.1
Accept: application/json

{
    "tag": "interesting"
}
```

Response (new tag create)

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "id": 1,
    "tag": "interesting"
}
```

Request

```
POST /api/v1/tags/ HTTP/1.1
Accept: application/json

{
    "tag": "interesting"
}
```

Response (existing tag returned)

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 1,
    "tag": "interesting"
}
```

Statuscode 200 an existing tag was found and returned

Statuscode 201 a new tag was created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

`pyfarm.master.api.tags.`**`schema`**`()`

Returns the basic schema of `Tag`

`GET /api/v1/tags/schema/ HTTP/1.1`

Request

```
GET /api/v1/tags/schema/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": "INTEGER",
    "tag": "VARCHAR(64)"
}
```

Statuscode 200 no error

pyfarm.master.api.tasklogs module

Task Logs This module defines an API for managing and querying logs belonging to tasks

class `pyfarm.master.api.tasklogs.`**`LogsInTaskAttemptsIndexAPI`**

Bases: `flask.views.MethodView`

`get` (*job_id*, *task_id*, *attempt*)

A `GET` to this endpoint will return a list of all known logs that are associated with this attempt at running this task

`GET /api/v1/jobs/<job_id>/tasks/<task_id>/attempts/<attempt>/logs/ HTTP/1.1`

Request

```
GET /api/v1/jobs/4/tasks/1300/attempts/5/logs/ HTTP/1.1
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

```

```
[
    {
        "agent_id": "3087ada4-290a-45b0-8c1a-21db4cd284fc",
        "created_on": "2014-09-03T10:58:59.754880",
        "identifier": "2014-09-03_10-58-59_4_4ee02475335911e4a935c86000cbf5fb.csv"
    }
]
```

Statuscode 200 no error

Statuscode 404 the specified task was not found

methods = ['GET', 'POST']

post (*job_id*, *task_id*, *attempt*)

A POST to this endpoint will register a new logfile with the given attempt at running the given task

A logfile has an identifier which must be unique in the system. If two tasks get assigned a logfile with the same id, it is considered to be the same log.

POST /api/v1/jobs/<job_id>/tasks/<task_id>/attempts/<attempt>/logs/ HTTP/1.1

Request

```
POST /api/v1/jobs/4/tasks/1300/attempts/5/logs/ HTTP/1.1
Content-Type: application/json

{
    "identifier": "2014-09-03_10-58-59_4_4ee02475335911e4a935c86000cbf5fb.csv",
    "agent_id": "2dc2cb5a-35da-41d6-8864-329c0d7d5391"
}
```

Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "identifier": "2014-09-03_10-58-59_4_4ee02475335911e4a935c86000cbf5fb.csv",
    "agent_id": "2dc2cb5a-35da-41d6-8864-329c0d7d5391",
    "created_on": "2014-09-03T10:59:05.103005",
    "id": 148
}
```

Statuscode 201 the association between this task attempt and logfile has been created

Statuscode 400 there was something wrong with the request (such as invalid columns being included)

Statuscode 404 the specified task does not exist

Statuscode 409 the specified log was already registered on the specified task

class pyfarm.master.api.tasklogs.**SingleLogInTaskAttempt**

Bases: flask.views.MethodView

get (*job_id*, *task_id*, *attempt*, *log_identifier*)

A GET to this endpoint will return metadata about the specified logfile

GET /api/v1/jobs/<job_id>/tasks/<task_id>/attempts/<attempt>/logs/<log_identifier>

Request

```
GET /api/v1/jobs/4/tasks/1300/attempts/5/logs/2014-09-03_10-58-59_4_4ee02475335911e4a935
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 147,
    "identifier": "2014-09-03_10-58-59_4_4ee02475335911e4a935c86000cbf5fb.csv",
    "created_on": "2014-09-03T10:58:59.754880",
    "agent_id": "836ce137-6ad4-443f-abb9-94c4465ff87c"
}
```

Statuscode 200 no error

Statuscode 404 task or logfile not found

methods = ['GET']

class pyfarm.master.api.tasklogs.**TaskLogfileAPI**

Bases: flask.views.MethodView

get (*job_id*, *task_id*, *attempt*, *log_identifier*)

A GET to this endpoint will return the actual logfile or a redirect to it.

GET /api/v1/jobs/<job_id>/tasks/<task_id>/attempts/<attempt>/logs/<log_identifier>/

Request

```
GET /api/v1/jobs/4/tasks/1300/attempts/5/logs/2014-09-03_10-58-59_4_4ee02475335911e4a935
Accept: text/csv
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/csv

<Content of the logfile>
```

Statuscode 200 no error

Statuscode 307 The logfile can be found in another location at this point in time. Independent future requests for the same logfile should continue using the original URL

Statuscode 400 the specified logfile identifier is not acceptable

Statuscode 404 task or logfile not found

methods = ['GET', 'PUT']

put (*job_id*, *task_id*, *attempt*, *log_identifier*)

A `PUT` to this endpoint will upload the request's body as the specified logfile

PUT /api/v1/jobs/<job_id>/tasks/<task_id>/attempts/<attempt>/logs/<log_identifier>/

Request

```
PUT /api/v1/jobs/4/tasks/1300/attempts/5/logs/2014-09-03_10-58-59_4_4ee02475335911e4a935

<content of the logfile>
```

Response

```
HTTP/1.1 201 CREATED
```

Statuscode 201 lofile was uploaded

Statuscode 400 the specified logfile identifier is not acceptable

Statuscode 404 task or logfile not found

Module contents

3.1.3 pyfarm.master.user_interface package

Submodules

pyfarm.master.user_interface.agents module

`pyfarm.master.user_interface.agents.`**agent_add_software** (*agent_id*)

`pyfarm.master.user_interface.agents.`**agent_delete_software** (*agent_id*, *version_id*)

`pyfarm.master.user_interface.agents.`**agents** ()

`pyfarm.master.user_interface.agents.`**delete_single_agent** (*agent_id*)

`pyfarm.master.user_interface.agents.`**or_** (**clauses*)

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

`pyfarm.master.user_interface.agents.`**`restart_single_agent`**`(`*agent_id*`)`

`pyfarm.master.user_interface.agents.`**`single_agent`**`(`*agent_id*`)`

pyfarm.master.user_interface.jobqueues module

`pyfarm.master.user_interface.jobqueues.`**`delete_jobqueue`**`(`*queue_id*`)`

`pyfarm.master.user_interface.jobqueues.`**`delete_subqueue`**`(`*queue*`)`

`pyfarm.master.user_interface.jobqueues.`**`jobqueue`**`(`*queue_id*`)`

`pyfarm.master.user_interface.jobqueues.`**`jobqueue_create`**`()`

`pyfarm.master.user_interface.jobqueues.`**`jobqueues`**`()`

pyfarm.master.user_interface.jobs module

`pyfarm.master.user_interface.jobs.`**`add_notified_user_to_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`alter_autodeletion_for_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`alter_frames_in_single_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`alter_scheduling_parameters_for_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`asc`**`(`*column*`)`

Produce an ascending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import asc
stmt = select([users_table]).order_by(asc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name ASC
```

The `asc()` function is a standalone version of the `ColumnElement.asc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.asc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `asc()` operation.

See also:

`desc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

`pyfarm.master.user_interface.jobs.`**`delete_multiple_jobs`**`()`

`pyfarm.master.user_interface.jobs.`**`delete_single_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`desc`**`(`*column*`)`

Produce a descending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import desc

stmt = select([users_table]).order_by(desc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name DESC
```

The `desc()` function is a standalone version of the `ColumnElement.desc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.desc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `desc()` operation.

See also:

`asc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

`pyfarm.master.user_interface.jobs.`**`distinct`**`(`*expr*`)`

Produce an column-expression-level unary `DISTINCT` clause.

This applies the `DISTINCT` keyword to an individual column expression, and is typically contained within an aggregate function, as in:

```
from sqlalchemy import distinct, func
stmt = select([func.count(distinct(users_table.c.name))])
```

The above would produce an expression resembling:

```
SELECT COUNT(DISTINCT name) FROM user
```

The `distinct()` function is also available as a column-level method, e.g. `ColumnElement.distinct()`, as in:

```
stmt = select([func.count(users_table.c.name.distinct())])
```

The `distinct()` operator is different from the `Select.distinct()` method of `Select`, which produces a `SELECT` statement with `DISTINCT` applied to the result set as a whole, e.g. a `SELECT DISTINCT` expression. See that method for further information.

See also:

`ColumnElement.distinct()`

`Select.distinct()`

`func`

`pyfarm.master.user_interface.jobs.`**`jobs`**`()`

`pyfarm.master.user_interface.jobs.`**`or_`**`(`**clauses*`)`

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

`pyfarm.master.user_interface.jobs.`**`pause_multiple_jobs`**`()`

`pyfarm.master.user_interface.jobs.`**`pause_single_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`remove_notified_user_from_job`**`(`*job_id*, *user_id*`)`

`pyfarm.master.user_interface.jobs.`**`rerun_failed_in_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`rerun_failed_in_multiple_jobs`**`()`

`pyfarm.master.user_interface.jobs.`**`rerun_multiple_jobs`**`()`

`pyfarm.master.user_interface.jobs.`**`rerun_single_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`rerun_single_task`**`(`*job_id*, *task_id*`)`

`pyfarm.master.user_interface.jobs.`**`single_job`**`(`*job_id*`)` `pyfarm.master.user_interface.jobs.`**`unpause_multiple_jobs`**`()`

`pyfarm.master.user_interface.jobs.`**`unpause_single_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`update_notes_for_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`update_tags_in_job`**`(`*job_id*`)`

`pyfarm.master.user_interface.jobs.`**`upgrade_job_to_latest_jobtype_version`**`(`*job_id*`)`

pyfarm.master.user_interface.jobtypes module

Jobtypes UI endpoints allowing seeing and manipulating jobtypes via the web interface

`pyfarm.master.user_interface.jobtypes.`**`add_jobtype_software_requirement`**`(`*jobtype_id*`)`

`pyfarm.master.user_interface.jobtypes.`**`create_jobtype`**`()`

`pyfarm.master.user_interface.jobtypes.`**`desc`**`(`*column*`)`

Produce a descending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import desc

stmt = select([users_table]).order_by(desc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name DESC
```

The `desc()` function is a standalone version of the `ColumnElement.desc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.desc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `desc()` operation.

See also:

`asc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

`pyfarm.master.user_interface.jobtypes.`**`jobtype`**`(`*jobtype_id*`)`

UI endpoint for a single jobtype. Allows showing and updating the jobtype

`pyfarm.master.user_interface.jobtypes.`**`jobtypes`**`()`

`pyfarm.master.user_interface.jobtypes.`**`remove_jobtype`**`(`*jobtype_id*`)`

`pyfarm.master.user_interface.jobtypes.`**`remove_jobtype_software_requirement`**`(`*jobtype_id, software_id*`)`

Module contents

3.2 Submodules

3.2.1 pyfarm.master.application module

Application

Contains the functions necessary to construct the application layer classes necessary to run the master.

class pyfarm.master.application.**SessionMixin**

Bases: object

Mixin which adds a _session attribute. This class is provided mainly to limit issues with circular imports.

class pyfarm.master.application.**UUIDConverter**(*map*)

Bases: werkzeug.routing.BaseConverter

A URL converter for UUIDs. This class is loaded as part of the Flask application setup and may be used in url routing:

```
@app.route('/foo/<uuid:value>')
def foobar(value):
    pass
```

When a request such as GET /foo/F9A63B47-66BF-4E2B-A545-879986BB7CA9 is made UUIDConverter will receive value to to_python() which will then convert the string to an instance of UUID.

to_python(*value*)

to_url(*value*)

pyfarm.master.application.**before_request**()

Global before_request handler that will handle common problems when trying to accept json data to the api.

pyfarm.master.application.**get_api_blueprint**(*url_prefix=None*)

Constructs and returns an instance of Blueprint for routing api requests.

Parameters **url_prefix** (*string*) – The url prefix for the api such as /api/v1. If not provided then value will be derived from PYFARM_API_PREFIX and/or PYFARM_API_VERSION

pyfarm.master.application.**get_application**(***configuration_keywords*)

Returns a new application context. If keys and values are provided to config_values they will be used to override the default configuration values or create new ones

```
>>> app = get_application(TESTING=True)
>>> assert app.testing is True
```

Parameters **setup_appcontext** (*bool*) – If True then setup the flask.g variable to include the application level information (ex. g.db)

pyfarm.master.application.**get_login_manager**(***kwargs*)

Constructs and returns an instance of LoginManager. Any keyword arguments provided will be passed to the constructor of LoginManager

`pyfarm.master.application.`**`get_login_serializer`**`(`*secret_key*`)`
Constructs and returns and instance of `URLSafeTimedSerializer`

`pyfarm.master.application.`**`get_sqlalchemy`**`(`*app=None*, *use_native_unicode=True*, *session_options=None*`)`
Constructs and returns an instance of `SQLAlchemy`. Any keyword arguments provided will be passed to the constructor of `SQLAlchemy`

3.2.2 pyfarm.master.entrypoints module

Entry Points

Contains the code which operates the Python entry point scripts as well as serving as a central location for the construction of the web application.

`pyfarm.master.entrypoints.`**`create_app`**`()`
An entry point specifically for uWSGI or similar to use

`pyfarm.master.entrypoints.`**`load_api`**`(`*app_instance*, *api_instance*`)`
configures flask to serve the api endpoints

`pyfarm.master.entrypoints.`**`load_authentication`**`(`*app_instance*`)`
configures flask to serve the authentication endpoints

`pyfarm.master.entrypoints.`**`load_before_first`**`(`*app_instance*, *database_instance*`)`

`pyfarm.master.entrypoints.`**`load_error_handlers`**`(`*app_instance*`)`
loads the error handlers onto application instance

`pyfarm.master.entrypoints.`**`load_index`**`(`*app_instance*`)`
configures flask to serve the main index and favicon

`pyfarm.master.entrypoints.`**`load_master`**`(`*app*, *api*`)`
loads and attaches all endpoints needed to run the master

`pyfarm.master.entrypoints.`**`load_setup`**`(`*app_instance*`)`
configures flask to serve the endpoint used for setting up the system

`pyfarm.master.entrypoints.`**`load_user_interface`**`(`*app_instance*`)`

`pyfarm.master.entrypoints.`**`run_master`**`()`
Runs `load_master()` then runs the application

`pyfarm.master.entrypoints.`**`tables`**`()`
Small script for basic table management and, eventually, some introspection as well.

3.2.3 pyfarm.master.index module

Index

Contains the endpoints for master"s index ("/")

`pyfarm.master.index.`**`favicon`**`()`
Sends out the favicon from the static directory

Warning: On deployment, /favicon.ico should really be handled by the frontend server and **not** the application.

pyfarm.master.index.**index_page**()

3.2.4 pyfarm.master.initial module

Initial Setup

Entry points for the /setup/ target

class pyfarm.master.initial.**NewUserForm**(*formdata=None*, *obj=None*, *prefix=''*, *data=None*, *meta=None*, ***kwargs*)

Bases: wtforms.form.Form

email = <UnboundField(TextField, (), {'validators': [<wtforms.validators.Required object at 0x7f161f3565f8>]})>

password = <UnboundField(PasswordField, (), {'validators': [<wtforms.validators.Required object at 0x7f161f08e630>]

username = <UnboundField(TextField, (), {'validators': [<wtforms.validators.Required object at 0x7f161f356d30>]})>

validate_username(*field*)

pyfarm.master.initial.**setup_page**()

3.2.5 pyfarm.master.testutil module

Test Utilities

Functions and classes mainly used during the unittests.

class pyfarm.master.testutil.**BaseTestCase**(*methodName='runTest'*)

Bases: unittest.case.TestCase

ENVIRONMENT_SETUP = False

ORIGINAL_ENVIRONMENT = {'_MP_FORK_LOGFORMAT_': '[%(asctime)s: %(levelname)s/%(processName)s] %(me

assert_accepted(*response*)

assert_bad_request(*response*)

assert_conflict(*response*)

assert_contents_equal(*a_source*, *b_source*)

Explicitly check to see of the two iterable objects contain the same data. This method exists to check to make sure two iterables contain the same data without regards to order. This is mostly meant for cases where two lists contain unhashable types.

assert_created(*response*)

assert_forbidden(*response*)

assert_internal_server_error(*response*)

assert_method_not_allowed(*response*)

assert_no_content(*response*)

assert_not_acceptable(*response*)

assert_not_found(*response*)

assert_ok(*response*)

assert_status(*response*, *status_code=None*)

`assert_temporary_redirect`(*response*)

`assert_unauthorized`(*response*)

`assert_unsupported_media_type`(*response*)

classmethod `build_environment`()

Sets up the current environment with some values for unittesting. This must be used before any other code is imported otherwise

Warning: This classmethod should not be used outside of a testing context

`maxDiff` **= None**

`setUp`()

`setup_app`()

Constructs the application object and assigns the instance variables for tests. If you're testing the master your sublcass will probably need to extend this method.

`setup_client`(*app*)

returns the test client from the given application instance

`setup_database`()

`setup_warning_filter`()

`tearDown`()

`teardown_app`()

`teardown_database`()

`teardown_warning_filter`()

class `pyfarm.master.testutil.`**`JsonResponseMixin`**

Bases: `object`

Mixin with testing helper methods

`json`

`pyfarm.master.testutil.`**`get_test_environment`**(***environment*)

Returns a dictionary that can be used to simulate a working environment. Any key/value pairs passed in as keyword arguments will override the defaults.

`pyfarm.master.testutil.`**`make_test_response`**(*response_class=None*)

3.2.6 pyfarm.master.utility module

Utility

General utility which are not view or tool specific

class `pyfarm.master.utility.`**`JSONEncoder`**(*skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: `json.encoder.JSONEncoder`

`default`(*o*)

`pyfarm.master.utility.`**`assert_mimetypes`**(*flask_request*, *mimetypes*)

> **Warning:** This function will produce an unhandled error if you use it outside of a request.

Check to make sure that the request's mimetype is in `mimetypes`. If this is not true then call `flask.abort()` with `UNSUPPORTED_MEDIA_TYPE`

Parameters

- **`flask_request`** – The flask request object which we should check the `mimetype` attribute on.
- **`mimetypes`** (*list, tuple, set*) – The mimetypes which `flask_request` can be.

`pyfarm.master.utility.`**`default_json_encoder`**(*obj*)

`pyfarm.master.utility.`**`dumps`**(*obj*, ***kwargs*)

Wrapper for `json.dumps()` that ensures `JSONEncoder` is passed in.

`pyfarm.master.utility.`**`error_handler`**(*e*, *code=None*, *default=None*, *title=None*, *template=None*)

Constructor for http errors that respects the current mimetype. By default this function returns html however when `request.mimetype` is `application/json` it will return a json response. This function is typically used within a `functools.partial()` call:

```
>>> from functools import partial
>>> try:
...     from httplib import BAD_REQUEST
... except ImportError:
...     from http.client import BAD_REQUEST
...
>>> from flask import request
>>> error_400 = partial(
...     error_handler, BAD_REQUEST,
...     lambda: "bad request to %s" % request.url, "Bad Request")
```

Parameters

- **`e`** (*flask.Response*) – The response object which will be passed into `error_handler()`, this value is ignored by default.
- **`code`** (*int*) – The integer to use in the response. For the most consistent results you can use the `httplib` or `http.client` modules depending on your Python version.
- **`default`** (*callable*) – This will be the default error message if g.error does not contain anything. `default` may either be a callable function which will produce the string or it may be a string by itself.
- **`title`** (*str*) – The HTML title of the request being made. This is not used when dealing with json requests and if not provided at all will default to using the official status code's string representation.
- **`template`** (*str*) – A alternative template path for HTML responses

`pyfarm.master.utility.`**`get_g`**(*attribute*, *instance_types*, *unset=<object object at 0x7f1626d127a0>*)

Returns data from `flask.g` after checking to make sure the attribute was set and that it has the correct type.

This function does not check to see if you're already inside a request.

Parameters

- **attribute** (*str*) – The name of the attribute on the `flask.g` object
- **instance_types** (*tuple*) – A tuple of classes which the data we're looking for should be a part of

`pyfarm.master.utility.`**`get_request_argument`**(*argument*, *default=None*, *required=False*, *types=None*)

This is a function similar to Flask's `request.args.get` except it does type validation and it has the concept of required url arguments.

Parameters

- **argument** (*str*) – The name of the url argument we're trying to retrieve
- **default** – The value to return if `argument` is not present in the url and argument is not a required parameter.
- **required** (*bool*) – If True and the url argument provided by `argument` is not provided respond to the request with `BAD_REQUEST`
- **types** – A single or list of multiple callable objects which will be used to try and produce a result to return. This would function similarly to this:

```
value = "5"
types = (int, bool)

for type_callable in types:
    try:
        return type_callable(value)
    except Exception:
        continue
```

`pyfarm.master.utility.`**`inside_request`**()

Returns True if we're inside a request, False if not.

`pyfarm.master.utility.`**`isuuid`**(*value*)

Returns True if `value` is a `UUID` object or can be converted to one

`pyfarm.master.utility.`**`jsonify`**(**args*, ***kwargs*)

Drop in replacement for `flask.jsonify()` that also handles list objects as well as a few custom objects like Decimal or datetime. Flask does not support lists by default because it's considered a security risk in most cases but we do need it in certain cases. Since flask's jsonify does not allow passing arbitrary arguments to `json.dumps()`, we cannot use it if the output data contains custom types.

`pyfarm.master.utility.`**`validate_json`**(*validator*, *json_types=(<class 'dict'>,)*)

A decorator, similar to `validate_with_model()`, but greatly simplified and more flexible. Unlike `validate_with_model()` this decorator is meant to handle data which may not be structured for a model.

Parameters

- **mimetype** (*tuple*) – A tuple of mimetypes that are allowed to be handled by the decorated function.
- **json_types** (*tuple*) – The root type or types which the object on `g.json` should be an instance of.

`pyfarm.master.utility.`**`validate_with_model`**(*model*, *type_checks=None*, *ignore=None*, *ignore_missing=None*, *disallow=None*)

Decorator which will check the contents of the of the json request against a model for:

- missing fields which are required

- values which don't match their type(s) in the database
- inclusion of fields which do not exist

Parameters

- **`model`** – The model object that the decorated endpoint should use for testing the points above.
- **`type_checks`** (*dict*) – A dictionary containing a mapping of column names to special functions used for checking. If there's a key in the incoming request that needs a more detailed check than "isinstance(g.json[column_name], <Python type(s) from sql>)" then this is the place to add it.
- **`ignore_missing`** (*list*) – A list of fields to completely ignore in the incoming request. Typically this is used by `PUT` requests or other similar requests where part of the data is in the url.
- **`allow_missing`** (*list*) – A list of fields which are allowed to be missing in the request. These fields will still be checked for type however.
- **`disallow`** (*list*) – A list of columns which are never in the request to the decorated function

3.3 Module contents

Contains all the necessary code to operate an instance of the master.

CHAPTER 4

pyfarm.models package

4.1 Subpackages

4.1.1 pyfarm.models.core package

Submodules

pyfarm.models.core.cfg module

Configuration Variables Stores basic configuration data related to tables and models. Most of these variables have defaults defined in the configuration under *PYFARM_DB_<value>*

const string TABLE_PREFIX Prefix for all tables

const string TABLE_SOFTWARE Stores the name of the table for software items

const string TABLE_TAG Stores the name of the table for tags

const string TABLE_AGENT Stores the name of the table for agents

const string TABLE_AGENT_TAGS Stores the name of the table for agent tags

const string TABLE_JOB Stores the name of the table for jobs

const string TABLE_JOB_TAG Stores the name of the table for job tags

const string TABLE_TASK Stores the name of the table for job tasks

const string TABLE_USER Stores the registered users (both human and api)

const string TABLE_ROLE Stores roles in which a user can operate in

const string TABLE_USERS_USER_ROLE Stores relationships between `TABLE_USERS_USER` and `TABLE_ROLE`

const string TABLE_JOB_QUEUES Stores the name of the table for job queues

const string TABLE_PATH_MAP Stores the name of the table for path maps

const integer MAX_HOSTNAME_LENGTH the max length of a hostname

const integer MAX_JOBTYPE_LENGTH the max length of a jobtype

const integer MAX_COMMAND_LENGTH the max length of a command (ex. *bash* or *cmd.exe*)

const integer MAX_USERNAME_LENGTH the max length of a username

const integer MAX_TAG_LENGTH the max length of a tag

Note: this value is shared amongst all tag columns and may be split into multiple values at a later time

pyfarm.models.core.functions module

Functions Contains core functions and data for use by `pyfarm.models`

`pyfarm.models.core.functions.`**`getuuid`**`(`*value*, *table*, *table_attrib*, *error_tail*`)`

Returns the proper value for the given input. Depending on the type being provided this will return one of the following:

- None
- the value from an attribute
- string from a UUID
- the original value (after validating it's a UUID)

Parameters

- **`value`** (*string*) – the value to validate and returning data from
- **`table`** (*string*) – the table which the provided *value* belongs to
- **`table_attrib`** (*string*) – the attribute to use when attempting to pull data off of a model object
- **`error_tail`** (*string*) – added to the end of error messages
- **`error_text`** (*str*) – error text to render in the event of problems

Raises ValueError raised when the provided input is invalid, blank, or otherwise unexpected

`pyfarm.models.core.functions.`**`modelfor`**`(`*model*, *table*`)`

Returns True if the given *model* object is for the expected *table*.

```
>>> from pyfarm.models.core.cfg import TABLE_AGENT
>>> from pyfarm.models.agent import Agent
>>> modelfor(Agent("foo", "10.56.0.0", "255.0.0.0"), TABLE_AGENT)
True
```

`pyfarm.models.core.functions.`**`repr_enum`**`(`*value*, *enum=None*`)`

produces the string representation of an enum value

`pyfarm.models.core.functions.`**`repr_ip`**`(`*value*`)`

properly formats an `IPAddress` object

`pyfarm.models.core.functions.`**`split_and_extend`**`(`*items*`)`

Takes a list of input elements and splits them before producing an extended set.

Example

```
>>> split_and_extend(["root.admin", "admin"])
set(['admin', 'root.admin', 'root'])
```

`pyfarm.models.core.functions.`**`work_columns`**(*state_default*, *priority_default*)
Produces some default columns which are used by models which produce work.

pyfarm.models.core.mixins module

Mixin Classes Module containing mixins which can be used by multiple models.

class `pyfarm.models.core.mixins.`**`ModelTypes`**(*primary_keys*, *autoincrementing*, *columns*, *required*, *relationships*, *mappings*)
Bases: `tuple`

`autoincrementing`
Alias for field number 1

`columns`
Alias for field number 2

`mappings`
Alias for field number 5

`primary_keys`
Alias for field number 0

`relationships`
Alias for field number 4

`required`
Alias for field number 3

class `pyfarm.models.core.mixins.`**`ReprMixin`**
Bases: `object`

Mixin which allows model classes to to convert columns into a more easily read object format.

Variables

- **`REPR_COLUMNS`** (*tuple*) – the columns to convert
- **`REPR_CONVERT_COLUMN`** (*dict*) – optional dictionary containing columns names and functions for converting to a more readable string format

`REPR_COLUMNS` **= NotImplemented**

`REPR_CONVERT_COLUMN` **= {}**

class `pyfarm.models.core.mixins.`**`UtilityMixins`**
Bases: `object`

Mixins which can be used to produce dictionaries of existing data.

Const dict DICT_CONVERT_COLUMN A dictionary containing key value pairs of attribute names and a function to retrieve the attribute. The function should take a single input and return the value itself. Optionally, you can also use the `NotImplemented` object to exclude some columns from the results.

`DICT_CONVERT_COLUMN` **= {}**

`to_dict`(*unpack_relationships=True*)
Produce a dictionary of existing data in the table

Parameters **`unpack_relationships`** (*list, tuple, set, bool*) – If `True` then unpack all relationships. If `unpack_relationships` is an iterable such as a list or tuple object then only unpack those relationships.

classmethod to_schema()

Produce a dictionary which represents the table's schema in a basic format

classmethod types()

A classmethod that constructs a `namedtuple` object with four attributes:

- primary_keys - set of all primary key(s) names
- autoincrementing - set of all columns which have autoincrement set
- columns - set of all column names
- required - set of all required columns (non-nullable wo/defaults)
- relationships - not columns themselves but do store relationships
- mappings - contains a dictionary with each field mapping to a Python type

class pyfarm.models.core.mixins.**ValidatePriorityMixin**

Bases: `object`

Mixin that adds a *state* column and uses a class level *STATE_ENUM* attribute to assist in validation.

MAX_PRIORITY = 1000

MIN_PRIORITY = -1000

validate_attempts(*key*, *value*)

ensures the number of attempts provided is valid

validate_priority(*key*, *value*)

ensures the value provided to priority is valid

class pyfarm.models.core.mixins.**ValidateWorkStateMixin**

Bases: `object`

STATE_ENUM = NotImplemented

validate_state(*key*, *value*)

Ensures that `value` is a member of `STATE_ENUM`

validate_state_column(*key*, *value*)

validates the state column

class pyfarm.models.core.mixins.**WorkStateChangedMixin**

Bases: `object`

Mixin which adds a static method to be used when the model state changes

static state_changed(*target*, *new_value*, *old_value*, *initiator*)

update the datetime objects depending on the new value

pyfarm.models.core.types module

Custom Columns and Type Generators Special column types used by PyFarm's models.

class pyfarm.models.core.types.**AgentStateEnum**(**args*, ***kwargs*)

Bases: `pyfarm.models.core.types.EnumType`

custom column type for working with `AgentState`

enum = AgentState(OFFLINE=Values(201, 'offline'), ONLINE=Values(202, 'online'), DISABLED=Values(200, 'disabled'

class `pyfarm.models.core.types.`**`EnumType`**(**args*, ***kwargs*)

Bases: `sqlalchemy.sql.type_api.TypeDecorator`

Special column type which handles translation from a human readable enum into an integer that the database can use.

Variables **`enum`** – required class level variable which defines what enum this custom column handles

Raises AssertionError raised if `enum` is not set on the class

`enum` **= NotImplemented**

`impl`

alias of `Integer`

`json_types` **= (<class 'str'>, <class 'int'>)**

`process_bind_param`(*value*, *dialect*)

Takes `value` and maps it to the internal integer.

Raises ValueError raised if `value` is not part of the class level `enum` mapping

`process_result_value`(*value*, *dialect*)

`pyfarm.models.core.types.`**`IDTypeAgent`**

alias of `UUIDType`

class `pyfarm.models.core.types.`**`IPAddress`**(*addr*, *version=None*, *flags=0*)

Bases: `netaddr.ip.IPAddress`

Custom version of `netaddr.IPAddress` which can match itself against other instance of the same class, a string, or an integer.

class `pyfarm.models.core.types.`**`IPv4Address`**(**args*, ***kwargs*)

Bases: `sqlalchemy.sql.type_api.TypeDecorator`

Column type which can store and retrieve IPv4 addresses in a more efficient manner

`MAX_INT` **= 4294967295**

`checkInteger`(*value*)

`impl`

alias of `BigInteger`

`json_types` **= (<class 'str'>, <class 'int'>)**

`process_bind_param`(*value*, *dialect*)

`process_result_value`(*value*, *dialect*)

class `pyfarm.models.core.types.`**`JSONDict`**(**args*, ***kwargs*)

Bases: `pyfarm.models.core.types.JSONSerializable`

Column type for storing dictionary objects as json

`json_types`

alias of `dict`

`serialize_types` **= (<class 'dict'>, <class 'collections.UserDict'>)**

class `pyfarm.models.core.types.`**`JSONList`**(**args*, ***kwargs*)

Bases: `pyfarm.models.core.types.JSONSerializable`

Column type for storing list objects as json

json_types
alias of `list`

serialize_types = (<class 'list'>, <class 'tuple'>, <class 'collections.UserList'>)

class `pyfarm.models.core.types.`**JSONSerializable**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.type_api.TypeDecorator`

Base of all custom types which process json data to and from the database.

Variables

- **serialize_types** (*tuple*) – the kinds of objects we expect to serialize to and from the database
- **serialize_none** (*bool*) – if True then return None instead of converting it to its json value
- **allow_blank** (*bool*) – if True, do not raise a `ValueError` for empty data
- **allow_empty** (*bool*) – if True, do not raise `ValueError` if the input data itself is empty

dumps(*value*)
Performs the process of dumping *value* to json. For classes such as `UserDict` or `UserList` this will dump the underlying data instead of the object itself.

impl
alias of `UnicodeText`

process_bind_param(*value*, *dialect*)
Converts the value being assigned into a json blob

process_result_value(*value*, *dialect*)
Converts data from the database into a Python object

serialize_none = False

serialize_types = None

class `pyfarm.models.core.types.`**MACAddress**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.type_api.TypeDecorator`

Column type which can store and retrieve MAC addresses in a more efficient manner

MAX_INT = 281474976710655

impl
alias of `BigInteger`

json_types = (<class 'str'>, <class 'int'>)

process_bind_param(*value*, *dialect*)

process_result_value(*value*, *dialect*)

class `pyfarm.models.core.types.`**OperatingSystemEnum**(**args*, ***kwargs*)
Bases: `pyfarm.models.core.types.EnumType`

custom column type for working with `AgentState`

enum = OperatingSystem(OTHER=Values(303, 'other'), WINDOWS=Values(301, 'windows'), BSD=Values(304, 'bsd'), L

class `pyfarm.models.core.types.`**UUIDType**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.type_api.TypeDecorator`

Custom column type which handles UUIDs in the appropriate manner for various databases.

impl
alias of `TypeEngine`

json_types
alias of `UUID`

load_dialect_impl(*dialect*)

process_bind_param(*value*, *dialect*)

process_result_value(*value*, *dialect*)

class `pyfarm.models.core.types.`**UseAgentAddressEnum**(**args*, ***kwargs*)
Bases: `pyfarm.models.core.types.EnumType`

custom column type for working with `UseAgentAddress`

enum = UseAgentAddress(REMOTE=Values(311, 'remote'), LOCAL=Values(310, 'local'), HOSTNAME=Values(312, 'ho

class `pyfarm.models.core.types.`**WorkStateEnum**(**args*, ***kwargs*)
Bases: `pyfarm.models.core.types.EnumType`

custom column type for working with `WorkState`

enum = WorkState(PAUSED=Values(100, 'paused'), FAILED=Values(107, 'failed'), DONE=Values(106, 'done'), RUNNIN

`pyfarm.models.core.types.`**id_column**(*column_type=None*, ***kwargs*)
Produces a column used for *id* on each table. Typically this is done using a class in `pyfarm.models.mixins` however because of the ORM and the table relationships it's cleaner to have a function produce the column.

Module contents

4.2 Submodules

4.2.1 pyfarm.models.agent module

Agent Models

Models and interface classes related to the agent.

class `pyfarm.models.agent.`**Agent**(***kwargs*)
Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.ValidatePriorityMixin`, `pyfarm.models.core.mixins.ValidateWorkStateMixin`, `pyfarm.models.core.mixins.UtilityMixin`
`pyfarm.models.core.mixins.ReprMixin`

Stores information about an agent include its network address, state, allocation configuration, etc.

Note: This table enforces two forms of uniqueness. The `id` column must be unique and the combination of these columns must also be unique to limit the frequency of duplicate data:

- `hostname`
- `port`
- `id`

MAX_CPUS = 256

MAX_PORT = 65535

MAX_RAM = 262144

MIN_CPUS = 1

MIN_PORT = 1024

MIN_RAM = 16

REPR_COLUMNS = (‘id’, ‘hostname’, ‘port’, ‘state’, ‘remote_ip’, ‘cpus’, ‘ram’, ‘free_ram’)

REPR_CONVERT_COLUMN = {‘remote_ip’: <function repr_ip at 0x7f16205fb620>}

STATE_DEFAULT = ‘online’

STATE_ENUM = MappedEnum(OFFLINE=’offline’, ONLINE=’online’, DISABLED=’disabled’, RUNNING=’running’)

api_url(*scheme=’http’*, *version=1*)
Returns the base url which should be used to access the api of this specific agent.

Raises ValueError Raised if this function is called while the agent’s `use_address` column is set to `PASSIVE`

cpu_allocation
The total amount of cpu space an agent is allowed to process work in. A value of 1.0 would mean an agent can handle as much work as the system could handle given the requirements of a task. For example if an agent has 8 cpus, cpu_allocation is .5, and a task requires 4 cpus then only that task will run on the system.

cpu_name
The make and model of CPUs in this agents

cpus
The number of logical CPU cores installed on the agent

free_ram
The amount of ram which was last considered free

get_supported_types()

gpus
The graphics cards that are installed in this agent

hostname
The hostname we should use to talk to this host. Preferably this value will be the fully qualified name instead of the base hostname alone.

id
Provides an id for the current row. This value should never be directly relied upon and it’s intended for use by relationships.

is_offline()

last_heard_from
Time we last had contact with this agent

last_polled
Time we last tried to contact the agent

mac_addresses
The MAC addresses this agent has

os_class
The type of operating system running on the agent; “linux”, “windows”, or “mac”.

os_fullname
The full human-readable name of the agent’s OS, as returned by platform.platform()

`port`
The port the agent is currently running on

`ram`
The amount of ram installed on the agent in megabytes

`ram_allocation`
The amount of ram the agent is allowed to allocate towards work. A value of 1.0 would mean to let the agent use all of the memory installed on the system when assigning work.

`remote_ip`
the remote address which came in with the request

`restart_requested`
If True, the agent will be restarted

`satisfies_jobtype_requirements`(*jobtype_version*)

`software_versions`
software this agent has installed or is configured for

`state`
Stores the current state of the host. This value can be changed either by a master telling the host to do something with a task or from the host via REST api.

`tags`
Tags associated with this agent

`task_logs`

`tasks`
Relationship between an `Agent` and any `pyfarm.models.Task` objects

`time_offset`
The offset in seconds the agent is from an official time server

`upgrade_to`
The version this agent should upgrade to.

`use_address`
The address we should use when communicating with the agent

classmethod `validate_hostname`(*key*, *value*)
Ensures that the hostname provided by *value* matches a regular expression that expresses what a valid hostname is.

`validate_hostname_column`(*key*, *value*)
Validates the hostname column

classmethod `validate_ipv4_address`(_, *value*)
Ensures the `ip` address is valid. This checks to ensure that the value provided is:

- not a hostmask
- not link local (**RFC 3927**)
- not used for multicast (**RFC 1112**)
- not a netmask (**RFC 4632**)
- not reserved (**RFC 6052**)
- a private address (**RFC 1918**)

validate_numeric_column(*key*, *value*)
Validates several numerical columns. Columns such as ram, cpus and port a are validated with this method.

validate_remote_ip(*key*, *value*)
Validates the remote_ip column

classmethod validate_resource(*key*, *value*)
Ensure the `value` provided for `key` is within an expected range. This classmethod retrieves the min and max values from the `Agent` class directory using:

```
>>> min_value = getattr(Agent, "MIN_%s" % key.upper())
>>> max_value = getattr(Agent, "MAX_%s" % key.upper())
```

version
The pyfarm version number this agent is running.

4.2.2 pyfarm.models.gpu module

GPU

Model describing a given make and model of graphics card. Every agent can have zero or more GPUs associated with it.

class pyfarm.models.gpu.**GPU**(***kwargs*)
Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.UtilityMixins`, `pyfarm.models.core.mixins.ReprMixin`

agents

fullname
The full name of this graphics card model

id
Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

4.2.3 pyfarm.models.job module

Job Models

Models and interface classes related to jobs.

class pyfarm.models.job.**Job**(***kwargs*)
Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.ValidatePriorityMixin`, `pyfarm.models.core.mixins.ValidateWorkStateMixin`, `pyfarm.models.core.mixins.WorkStateCha`
`pyfarm.models.core.mixins.ReprMixin`, `pyfarm.models.core.mixins.UtilityMixins`

Defines the attributes and environment for a job. Individual commands are kept track of by `Task`

MAX_CPUS = 256

MAX_RAM = 262144

MIN_CPUS = 1

MIN_RAM = 16

REPR_COLUMNS = ('id', 'state', 'project')

REPR_CONVERT_COLUMN = {'state': <built-in function repr>}

SPECIAL_CPUS = [0]

SPECIAL_RAM = [0]

STATE_ENUM = ['paused', 'failed', 'done', 'running', None]

alter_frame_range(*start*, *end*, *by*)

autodelete_time
If not None, this job will be automatically deleted this number of seconds after it finishes.

batch
Number of tasks to run on a single agent at once. Depending on the capabilities of the software being run this will either cause a single process to execute on the agent or multiple processes one after the other.

configured by: *job.batch*

by
The number of frames to count by between *start* and *end*. This column may also sometimes be referred to as 'step' by other software.

can_use_more_agents()

children

cpus
Number of cpus or threads each task should consume on each agent. Depending on the job type being executed this may result in additional cpu consumption, longer wait times in the queue (2 cpus means 2 'fewer' cpus on an agent), or all of the above.

Table 4.1: Special Values

Value	Result
0	minimum number of cpu resources not required
-1	agent cpu is exclusive for a task from this job

configured by: *job.cpus*

data
Json blob containing additional data for a job

Note: Changes made directly to this object are **not** applied to the session.

environ
Dictionary containing information about the environment in which the job will execute.

Note: Changes made directly to this object are **not** applied to the session.

get_batch()

hidden
If True, keep the job hidden from the queue and web ui. This is typically set to True if you either want to save a job for later viewing or if the jobs data is being populated in a deferred manner.

id
Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

job_queue_id
The foreign key which stores `JobQueue.id`

jobtype_version

jobtype_version_id
The foreign key which stores `JobTypeVersion.id`

maximum_agents
The scheduler will never assign more than this number of agents to this job.

minimum_agents
The scheduler will try to assign at least this number of agents to this job as long as it can use them, before any other considerations.

notes
Notes that are provided on submission or added after the fact. This column is only provided for human consumption, is not scanned, index, or used when searching

notified_users

num_assigned_agents()

output_link
An optional link to a URI where this job's output can be viewed.

parents

paused()

priority
The priority of the job relative to others in the queue. This is not the same as task priority.

configured by: *job.priority*

queue
The queue for this job

ram
Amount of ram a task from this job will require to be free in order to run. A task exceeding this value will not result in any special behavior.

Table 4.2: Special Values

Value	Result
0	minimum amount of free ram not required
-1	agent ram is exclusive for a task from this job

configured by: *job.ram*

ram_max
Maximum amount of ram a task is allowed to consume on an agent.

Warning: If set, the task will be **terminated** if the ram in use by the process exceeds this value.

ram_warning
Amount of ram used by a task before a warning raised. A task exceeding this value will not cause any work stopping behavior.

requeue
Number of times to requeue failed tasks

Table 4.3: Special Values

Value	Result
0	never requeue failed tasks
-1	requeue failed tasks indefinitely

configured by: *job.requeue*

software_requirements

state
The state of the job with a value provided by `WorkState`

tags
Relationship between this job and `Tag` objects

tasks

tasks_done
Relationship between this job and any `Task` objects which are done.

tasks_failed
Relationship between this job and any `Task` objects which have failed.

tasks_queued
Relationship between this job and any `Task` objects which are queued.

tasks_running
Relationship between this job and any `Task` objects which are running.

time_finished
Time the job was finished. This will be set when the last task finishes and reset if a job is requeued.

time_started
The time this job was started. By default this value is set when `state` is changed to an appropriate value or when a job is requeued.

time_submitted
The time the job was submitted. By default this defaults to using `datetime.datetime.utcnow()` as the source of submission time. This value will not be set more than once and will not change even after a job is requeued.

title
The title of this job

to_be_deleted
If true, the master will stop all running tasks for this job and then delete it.

update_state()

user
The owner of this job

user_id
The id of the user who owns this job

validate_resource(*key*, *value*)
Validation that ensures that the value provided for either `ram` or `cpus` is a valid value with a given range

weight
The weight of this job. The scheduler will distribute available agents between jobs and job queues in the same queue in proportion to their weights.

4.2.4 pyfarm.models.jobqueue module

Job Queue Model

Model for job queues

class `pyfarm.models.jobqueue.`**`JobQueue`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.UtilityMixins`, `pyfarm.models.core.mixins.ReprMixin`

Stores information about a job queue. Used for flexible, configurable distribution of computing capacity to jobs.

`REPR_COLUMNS` **= ('id', 'name')**

`children`

`fullpath`
The path of this jobqueue. This column is a database denormalization. It is technically redundant, but faster to access than recursively querying all parent queues. If set to NULL, the path must be computed by recursively querying the parent queues.

`get_job_for_agent`(*agent*)

`id`
Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

`jobs`

`maximum_agents`
The scheduler will never assign more than this number of agents to jobs in or below this queue.

`minimum_agents`
The scheduler will try to assign at least this number of agents to jobs in or below this queue as long as it can use them, before any other considerations.

`name`

`num_assigned_agents`()

`parent`
Relationship between this queue its parent

`parent_jobqueue_id`
The parent queue of this queue. If NULL, this is a top level queue.

`path`()

`priority`
The priority of this job queue. The scheduler will not assign any nodes to other job queues or jobs with the same parent and a lower priority as long as this one can still use nodes. The minimum_agents column takes precedence over this.

static **`top_level_unique_check`**(*mapper*, *connection*, *target*)

`weight`
The weight of this job queue. The scheduler will distribute available agents between jobs and job queues in the same queue in proportion to their weights.

`pyfarm.models.jobqueue.`**`asc`**(*column*)
Produce an ascending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import asc
stmt = select([users_table]).order_by(asc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name ASC
```

The `asc()` function is a standalone version of the `ColumnElement.asc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.asc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `asc()` operation.

See also:

`desc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

`pyfarm.models.jobqueue.`**`desc`**`(`*column*`)`

Produce a descending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import desc

stmt = select([users_table]).order_by(desc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name DESC
```

The `desc()` function is a standalone version of the `ColumnElement.desc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.desc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `desc()` operation.

See also:

`asc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

pyfarm.models.jobqueue.**distinct**(*expr*)

Produce an column-expression-level unary `DISTINCT` clause.

This applies the `DISTINCT` keyword to an individual column expression, and is typically contained within an aggregate function, as in:

```
from sqlalchemy import distinct, func
stmt = select([func.count(distinct(users_table.c.name))])
```

The above would produce an expression resembling:

```
SELECT COUNT(DISTINCT name) FROM user
```

The `distinct()` function is also available as a column-level method, e.g. `ColumnElement.distinct()`, as in:

```
stmt = select([func.count(users_table.c.name.distinct())])
```

The `distinct()` operator is different from the `Select.distinct()` method of `Select`, which produces a `SELECT` statement with `DISTINCT` applied to the result set as a whole, e.g. a `SELECT DISTINCT` expression. See that method for further information.

See also:

`ColumnElement.distinct()`

`Select.distinct()`

`func`

pyfarm.models.jobqueue.**or_**(**clauses*)

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

4.2.5 pyfarm.models.jobtype module

Job Type Models

Models and objects dedicated to handling information which is specific to an individual job. See `pyfarm.models.job` for more the more general implementation.

class `pyfarm.models.jobtype.`**`JobType`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.UtilityMixins`, `pyfarm.models.core.mixins.ReprMixin`

Stores the unique information necessary to execute a task

`REPR_COLUMNS` = **('id', 'name')**

`description`
Human readable description of the job type. This field is not required and is not directly relied upon anywhere.

`id`
Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

`name`
The name of the job type. This can be either a human readable name or the name of the job type class itself.

`validate_name`(*key*, *value*)

`versions`

4.2.6 pyfarm.models.pathmap module

Path Map Model

Model for path maps, allowing for OS-dependent mapping of path prefixes to other path prefixes.

class `pyfarm.models.pathmap.`**`PathMap`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.ReprMixin`, `pyfarm.models.core.mixins.UtilityMixins`

`id`
Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

`path_linux`
The path on linux platforms

`path_osx`
The path on Mac OS X platforms

`path_windows`
The path on Windows platforms

`tag`
Relationship attribute for the tag this path map applies to.

`tag_id`
The tag an agent needs to have for this path map to apply to it. If this is NULL, this path map applies to all agents, but is overridden by applying path maps that do specify a tag.

4.2.7 pyfarm.models.project module

4.2.8 pyfarm.models.software module

Software

Table of software items. Agents can reference this table to show that they provide a given software. Jobs or jobtypes can depend on a software via the SoftwareRequirement table

class `pyfarm.models.software.`**`Software`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.UtilityMixins`

Model to represent a versioned piece of software that can be present on an agent and may be depended on by a job and/or jobtype through the appropriate SoftwareRequirement table

`id`

Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

`software`

The name of the software

`versions`

All known versions of this software

4.2.9 pyfarm.models.tag module

Tag

Table with tags for both jobs and agents

class `pyfarm.models.tag.`**`Tag`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.UtilityMixins`

Model which provides tagging for `Job` and class:*.Agent* objects

`agents`

`id`

Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

`jobs`

`tag`

The actual value of the tag

4.2.10 pyfarm.models.task module

Task Models

Models and interface classes related to tasks

class `pyfarm.models.task.`**`Task`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.ValidatePriorityMixin`, `pyfarm.models.core.mixins.ValidateWorkStateMixin`, `pyfarm.models.core.mixins.WorkStateCha` `pyfarm.models.core.mixins.UtilityMixins`, `pyfarm.models.core.mixins.ReprMixin`

Defines a task which a child of a `Job`. This table represents rows which contain the individual work unit(s) for a job.

`REPR_COLUMNS` = ('id', 'state', 'frame', 'project')

`REPR_CONVERT_COLUMN` = {'state': functools.partial(<function repr_enum at 0x7f16205fb6a8>, enum=['paused', 'failed

`STATE_DEFAULT` = None

`STATE_ENUM` = ['paused', 'failed', 'done', 'running', None]

`agent`

`agent_id`
: Foreign key which stores `Job.id`

`attempts`
: The number of attempts which have been made on this task. This value is auto incremented when `state` changes to a value synonymous with a running state.

static `clear_error_state`(*target*, *new_value*, *old_value*, *initiator*)
: Sets `last_error` column to `None` if the task's state is 'done'

`failures`
: The number of times this task has failed. This value is auto incremented when `state` changes to a value synonymous with a failed state.

`frame`
: The frame this `Task` will be executing.

`hidden`
: hides the task from queue and web ui

`id`
: Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

static `increment_attempts`(*target*, *new_value*, *old_value*, *initiator*)

`job`
: relationship attribute which retrieves the associated job for this task

`job_id`
: Foreign key which stores `Job.id`

`last_error`
: This column may be set when an error is present. The agent typically sets this column when the job type either can't or won't run a given task. This column will be cleared whenever the task's state is returned to a non-error state.

`log_associations`

`priority`
: The priority of the job relative to others in the queue. This is not the same as task priority.

: **configured by**: *job.priority*

static `reset_agent_if_failed_and_retry`(*target*, *new_value*, *old_value*, *initiator*)

`sent_to_agent`
: Whether this task was already sent to the assigned agent

`state`
: The state of the job with a value provided by `WorkState`

time_finished
Time the job was finished. This will be set when the last task finishes and reset if a job is requeued.

time_started
The time this job was started. By default this value is set when `state` is changed to an appropriate value or when a job is requeued.

time_submitted
The time the job was submitted. By default this defaults to using `datetime.datetime.utcnow()` as the source of submission time. This value will not be set more than once and will not change even after a job is requeued.

static **update_failures**(*target*, *new_value*, *old_value*, *initiator*)

4.2.11 pyfarm.models.tasklog module

TaskLog

Model describing a log file for a task or batch of tasks. A task can be associated with more than one log file, for example because it needed to be retried and there are logs for every attempt or because the jobtype used uses more than one process to execute a batch. A log file can belong to more than one task if tasks have been batched together for execution.

class `pyfarm.models.tasklog.`**TaskLog**(***kwargs*)
Bases: `flask_sqlalchemy.Model`, `pyfarm.models.core.mixins.UtilityMixins`, `pyfarm.models.core.mixins.ReprMixin`

agent
Relationship between an `TaskLog`and the :class:`pyfarm.models.Agent` it was created on

agent_id
The agent this log was created on

created_on
The time when this log was created

id
Provides an id for the current row. This value should never be directly relied upon and it's intended for use by relationships.

identifier
The identifier for this log

task_associations

class `pyfarm.models.tasklog.`**TaskTaskLogAssociation**(***kwargs*)
Bases: `flask_sqlalchemy.Model`

attempt

log

task

task_id

task_log_id

4.2.12 pyfarm.models.user module

Permissions

Stores users and their roles in the database.

class `pyfarm.models.user.`**`User`**(***kwargs*)

Bases: `flask_sqlalchemy.Model`, `flask_login.UserMixin`, `pyfarm.models.core.mixins.ReprMixin`

Stores information about a user including the roles they belong to

`REPR_COLUMNS` = **('id', 'username')**

`active`

Enables or disables a particular user across the entire system

`check_password`(*password*)

checks the password provided against the stored password

classmethod `create`(*username*, *password*, *email=None*, *roles=None*)

`email`

Contact email for registration and possible notifications

`expiration`

User expiration. If this value is set then the user will no longer be able to access PyFarm past the expiration.

classmethod `get`(*id_or_username*)

Get a user model either by id or by the user's username

`get_auth_token`()

`get_id`()

`has_roles`(*allowed=None*, *required=None*)

checks the provided arguments against the roles assigned

classmethod `hash_password`(*value*)

`id`

`is_active`()

returns true if the user and the roles it belongs to are active

`jobs`

`last_login`

The last date that this user was logged in.

`onetime_code`

SHA256 one time use code which can be used for unique urls such as for password resets.

`password`

The password used to login

`roles`

`subscribed_jobs`

`username`

The username used to login.

4.3 Module contents

Contains all the models used for database communication and object relational management.

CHAPTER 5

pyfarm.scheduler package

5.1 Submodules

5.1.1 pyfarm.scheduler.celery_app module

Celery Application

Creates the base instance of `Celery` which is used by components of PyFarm's master that require interaction with a task queue. This module also configures Celery's beat scheduler for other tasks such as agent polling and task assignment.

5.1.2 pyfarm.scheduler.tasks module

Tasks

This module is responsible for finding and allocating tasks on agents.

`pyfarm.scheduler.tasks.`**`and_`**`(`*`*clauses`*`)`

Produce a conjunction of expressions joined by `AND`.

E.g.:

```
from sqlalchemy import and_

stmt = select([users_table]).where(
                and_(
                    users_table.c.name == 'wendy',
                    users_table.c.enrolled == True
                )
            )
```

The `and_()` conjunction is also available using the Python `&` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') &
                (users_table.c.enrolled == True)
            )
```

The `and_()` operation is also implicit in some cases; the `Select.where()` method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using `and_()`:

```
stmt = select([users_table]).\
            where(users_table.c.name == 'wendy').\
            where(users_table.c.enrolled == True)
```

See also:

`or_()`

`pyfarm.scheduler.tasks.`**`asc`**`(`*column*`)`

Produce an ascending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import asc
stmt = select([users_table]).order_by(asc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name ASC
```

The `asc()` function is a standalone version of the `ColumnElement.asc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.asc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `asc()` operation.

See also:

`desc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

`pyfarm.scheduler.tasks.`**`desc`**`(`*column*`)`

Produce a descending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import desc

stmt = select([users_table]).order_by(desc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name DESC
```

The `desc()` function is a standalone version of the `ColumnElement.desc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.desc())
```

Parameters **`column`** – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `desc()` operation.

See also:

`asc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

`pyfarm.scheduler.tasks.`**`distinct`**`(`*expr*`)`

Produce an column-expression-level unary `DISTINCT` clause.

This applies the `DISTINCT` keyword to an individual column expression, and is typically contained within an aggregate function, as in:

```
from sqlalchemy import distinct, func
stmt = select([func.count(distinct(users_table.c.name))])
```

The above would produce an expression resembling:

```
SELECT COUNT(DISTINCT name) FROM user
```

The `distinct()` function is also available as a column-level method, e.g. `ColumnElement.distinct()`, as in:

```
stmt = select([func.count(users_table.c.name.distinct())])
```

The `distinct()` operator is different from the `Select.distinct()` method of `Select`, which produces a `SELECT` statement with `DISTINCT` applied to the result set as a whole, e.g. a `SELECT DISTINCT` expression. See that method for further information.

See also:

`ColumnElement.distinct()`

`Select.distinct()`

`func`

`pyfarm.scheduler.tasks.`**`or_`**`(`**clauses*`)`

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

5.2 Module contents

5.2.1 Scheduler

This package contains the components used by PyFarm's master to schedule tasks, assign work to agents, and other periodic tasks that can't be performed inside of the web application.

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

/api