NailGun Documentation

Release 0.32.0

Jeremy Audet

Sep 19, 2023

Contents

1 Examples 3

2 API Documentation 15

3 Quick Start 47

4 Why NailGun? 49

5 Scope and Limitations 51

6 Resources 53

7 Contributing 55

Python Module Index 59

Index 61

NailGun is a GPL-licensed Python library that facilitates easy usage of the Satellite 6 API. It lets you write code like this:

```
>>> org = Organization(id=1).read()
```

This page provides a summary of information about NailGun.

Contents

- *NailGun*
  - *Quick Start*
  - *Why NailGun?*
  - *Scope and Limitations*
  - *Resources*
  - *Contributing*
    - *Nailgun Review Process*
    - *Nailgun Release Process*

More in-depth coverage is provided in other sections.

CHAPTER 1

Examples

This page contains several examples of how to use NailGun. The examples progress from simple to more advanced.

You can run any of the scripts presented in this document. This is the set-up procedure for scripts that use NailGun:

```
python3 -m venv env
source env/bin/activate
pip install nailgun
./some_script.py  # some script of your choice
```

This is the set-up procedure for scripts that do not use NailGun:

```
python3 -m venv env
source env/bin/activate
pip install requests
./some_script.py  # some script of your choice
```

Additionally, a video demonstration entitled NailGun Hands On is available.

Contents

- *Examples*
  - *Video Demonstration*
  - *Getting Started*
  - *Managing Server Configurations*
  - *Using More Methods*
    - *get_fields*
    - *create*
    - *read*

        * *update*
        * *search*
    - *Helper Functions*
        * *to_json_serializable*
    - *Using Lower Layers*

1.1 Video Demonstration

Note that this video does not touch on features that were added after it was recorded on May 26 2015, such as the *update* method.

1.2 Getting Started

This script demonstrates how to create an organization, print out its attributes and delete it using NailGun:

```
#!/usr/bin/env python3
"""Create an organization, print out its attributes and delete it.

Use NailGun to accomplish this task.

"""
from pprint import pprint

from nailgun.config import ServerConfig
from nailgun.entities import Organization


def main():
    """Create an organization, print out its attributes and delete it."""
    server_config = ServerConfig(
        auth=('admin', 'changeme'),  # Use these credentials...
        url='https://sat1.example.com',  # ...to talk to this server.
    )
    org = Organization(server_config, name='junk org').create()
    pprint(org.get_values())  # e.g. {'name': 'junk org', ...}
    org.delete()


if __name__ == '__main__':
    main()
```

This script demonstrates how to do the same *without* NailGun:

```
#!/usr/bin/env python3
"""Create an organization, print out its attributes and delete it.

Use Requests and standard library modules to accomplish this task.

"""
import json
```

(continues on next page)

(continued from previous page)

```
from pprint import pprint

import requests


def main():
    """Create an organization, print out its attributes and delete it."""
    auth = ('admin', 'changeme')
    base_url = 'https://sat1.example.com'
    organization_name = 'junk org'
    args = {'auth': auth, 'headers': {'content-type': 'application/json'}}

    response = requests.post(
        f'{base_url}/katello/api/v2/organizations',
        json.dumps(
            {
                'name': organization_name,
                'organization': {'name': organization_name},
            }
        ),
        **args,
    )
    response.raise_for_status()
    pprint(response.json())
    response = requests.delete(
        f"{base_url}/katello/api/v2/organizations/{response.json()['id']}", **args
    )
    response.raise_for_status()


if __name__ == '__main__':
    main()
```

1.3 Managing Server Configurations

In the example shown above, a *nailgun.config.ServerConfig* object was created in the body of the script. However, inter-mixing configuration data and program logic in this manner is problematic:

- Placing sensitive information in to a code-base puts that information at risk of becoming public, especially when the code-base is version-controlled.
- Server-specific configuration information is likely to change frequently. Placing that information in to a code-base means subjecting that code-base to unnecessary churn, making it harder for developers to find useful information in a repository's change log.

NailGun addresses this issue by providing full support for configuration files. Here's a simple example of how to create a pair of configuration objects, save them to disk, and read them back again:

```
>>> from nailgun.config import ServerConfig
>>> ServerConfig('http://sat1.example.com').save('sat1')
>>> ServerConfig('http://sat2.example.com').save('sat2')
>>> set(ServerConfig.get_labels()) == set(('sat1', 'sat2'))
True
>>> sat1_cfg = ServerConfig.get('sat1')
>>> sat2_cfg = ServerConfig.get('sat2')
```

A label of "default" is used when saving or reading configuration objects if no explicit label is given. As a result, this is valid:

```
>>> from nailgun.config import ServerConfig
>>> ServerConfig('bogus url').save()
>>> ServerConfig.get().url == 'bogus url'
True
```

The use of "default" is especially useful if you have created numerous server configurations, but only want to work with one at a time:

```
>>> from nailgun.config import ServerConfig
>>> ServerConfig.get('sat1').save()  # same as .save(label='default')
```

In addition, if no server configuration object is specified when instantiating an *nailgun.entity_mixins.Entity* object, the server configuration labeled "default" is used. With this in mind, here's a revised version of the first script in section *Getting Started*:

```
#!/usr/bin/env python3
"""Create an organization, print out its attributes and delete it."""
from pprint import pprint

from nailgun.entities import Organization


def main():
    """Create an organization, print out its attributes and delete it."""
    org = Organization(name='junk org').create()
    pprint(org.get_values())  # e.g. {'name': 'junk org', ...}
    org.delete()


if __name__ == '__main__':
    main()
```

This works just fine in many use cases. But what if you do not want to save your server configuration to disk? This might be the case if multiple processes are using NailGun and each process should default to communicating with a different default server, or if you are working with a read-only file system. In this case, you can use *nailgun.entity_mixins.DEFAULT_SERVER_CONFIG*.

NailGun handles other use cases, too. For example, the XDG base directory specification is obeyed, meaning that you can do things like provide a system-wide configuration file or place user configuration data in an alternate location. Read *nailgun.config* for full details.

1.4 Using More Methods

The examples so far have only made use of a small set of classes and methods:

- The `ServerConfig` class and several of its methods.
- The `Organization` class and its `create`, `get_values` and `delete` methods.

However, there are several more very useful high-level methods that you should be aware of. In addition, there are aspects to the `create` method that have not been touched on.

- *get_fields*
- *create*
- *read*
- *update*
- *search*

1.4.1 get_fields

The `get_fields` method is closely related to the `get_values` method. The former tells you which values *may* be assigned to an entity, and the latter tells you what values *are* assigned to an entity. For example:

```
>>> from nailgun.entities import Product
>>> product = Product(name='junk product')
>>> product.get_values()
{'name': 'junk product'}
>>> product.get_fields()
{
    'description': <nailgun.entity_fields.StringField object at 0x7fb5bf25ee10>,
    'gpg_key': <nailgun.entity_fields.OneToOneField object at 0x7fb5bf1f1128>,
    'id': <nailgun.entity_fields.IntegerField object at 0x7fb5bd4bd748>,
    'label': <nailgun.entity_fields.StringField object at 0x7fb5bd48b7f0>,
    'name': <nailgun.entity_fields.StringField object at 0x7fb5bd48b828>,
    'organization': <nailgun.entity_fields.OneToOneField object at 0x7fb5bd498f60>,
    'sync_plan': <nailgun.entity_fields.OneToOneField object at 0x7fb5bd49eac8>,
}
```

Fields serve two purposes. First, they provide typing information mixins. For example, a server expects this JSON payload when creating a product:

```
{
    "name": "junk product",
    "organization_id": 5,
    ...
}
```

And a server will return this JSON payload when reading a product:

```
{
    "name": "junk product",
    "organization": {
        'id': 3,
        'label': 'c5f2646f-5975-48c4-b2a3-bf8398b44510',
        'name': 'junk org',
    },
    ...
}
```

Notice how the "organization" field is named and structured differently in the above two cases. NailGun can deal with this irregularity due to the presence of the `StringField` and `OneToOneField`. If you are ever fiddling with an entity's definition, be careful to use the right field types. Otherwise, you may get some strange and hard-to-troubleshoot bugs.

Secondly, fields can generate random values for unit testing purposes. (This does *not* normally happen!) See the `create_missing` method for more information.

1.4.2 create

So far, we have only used brand new objects:

```
>>> from nailgun.entities import Organization
>>> org = entities.Organization(name='junk org').create()
```

However, we can also use existing objects:

```
>>> from nailgun.entities import Organization
>>> org = entities.Organization()
>>> org.name = 'junk org'
>>> org = org.create()
```

Note that the `create` method is side-effect free. As a result, the `org = org.create()` idiom is advisable. (The next section discusses this more.)

1.4.3 read

The `read` method fetches information about an entity. Typical usages of this method have already been shown, so this example goes in to more detail:

```
>>> from nailgun.entities import Organization
>>> org = Organization(id=418)
>>> response = org.read()
>>> for obj in (org, response):
...     type(obj)
...
<class 'nailgun.entities.Organization'>
<class 'nailgun.entities.Organization'>
>>> for obj in (org, response):
...     obj.get_values()
...
{'id': 418}
{
    'description': None,
    'id': 418,
    'label': 'junk_org',
    'name': 'junk org',
    'title': 'junk org',
}
```

Some notes on the above:

- The `read` method requires that an `id` attribute be present. Running `Organization().read()` will throw an exception.
- The `read` method is side-effect free. Rather than altering the object it is called on, it creates a new object, populates that object with attributes and returns the object. As a result, idioms like `org = org.read()` are advisable.

So far, we have only used brand new objects:

```
>>> from nailgun.entities import Organization
>>> org = Organization(id=418).read()
```

However, we can also use existing objects:

```
>>> from nailgun.entities import Organization
>>> org = Organziation()
>>> org.id = 418
>>> org = org.read()
```

1.4.4 update

The `update` method updates an entity's values. For example:

```
>>> from nailgun.entities import Organization
>>> org = Organization(id=418).read()
>>> org.get_values()
{
    'description': None,
    'id': 418,
    'label': 'junk_org',
    'name': 'junk org',
    'title': 'junk org',
}
>>> org.name = 'junkier org'
>>> org.description = 'supercalifragilisticexpialidocious'
>>> org = org.update()  # update all fields by default
>>> org.get_values()
{
    'description': 'supercalifragilisticexpialidocious',
    'id': 418,
    'label': 'junk_org',
    'name': 'junkier org',
    'title': 'junkier org',
}
>>> org.description = None
>>> org = org.update(['description'])  # update only named fields
>>> org.get_values()
{
    'description': None,
    'id': 418,
    'label': 'junk_org',
    'name': 'junkier org',
    'title': 'junkier org',
}
```

Some notes on the above:

- By default, the `update` method updates all fields. However, it is also possible to update a subset of fields.
- The `update` method is side-effect free. As a result, idioms like `org = org.update()` are advisable.

So far, we have only called `update` on existing objects. However, we can also call `update` on brand new objects:

```
>>> from nailgun.entities import Organization
>>> Organization(
...     id=418,
```

(continues on next page)

(continued from previous page)

```
...     name='junkier org',
...     description='supercalifragilisticexpialidocious',
... ).update(['name', 'description'])
```

1.4.5 search

The `search` method searches for entities. By default, it searches for all entities of a given kind:

```
lc_envs = LifecycleEnvironment().search()
```

If any attributes have been set, they are used. This finds all lifecycle environments that have a name of "foo" and that belong to organization 1:

```
lc_envs = LifecycleEnvironment(name='foo', organization=1).search()
```

You can choose to use only some fields in a search. This finds all lifecycle environments that have a name of "foo":

```
lc_envs = LifecycleEnvironment(name='foo', organization=1).search({'name'})
```

Other options are available, too. You can hard-code query parameters (especially useful for pagination), filter results locally and more. For examples of how to search, see *nailgun.entity_mixins.EntitySearchMixin.search()*. For examples of how search queries are generated, see *nailgun.entity_mixins.EntitySearchMixin.search_payload()*.

1.5 Helper Functions

Nailgun has also some helper functions for common operations.

1.5.1 to_json_serializable

This function parses nested nailgun entities, date, datetime, numbers, dict and list so the result can be parsed by json module:

```
>>> from nailgun import entities
>>> from nailgun.config import ServerConfig
>>> from datetime import date, datetime
>>> cfg=ServerConfig('https://foo.bar')
>>> dct = {'dict': {'objs':
[
    1, 'str', 2.5, date(2016, 12 , 13),
    datetime(2016, 12, 14, 1, 2, 3)
]}}
>>> entities.to_json(dct)
{'dict':
    {'objs': [1, 'str', 2.5, '2016-12-13', '2016-12-14 01:02:03']}
}
>>> env = entities.Environment(cfg, id=1, name='env')
>>> entities.to_json(env)
{'id': 1, 'name': 'env'}
>>> location =  entities.Location(cfg, name='loc')
>>> hostgroup = entities.HostGroup(
```

(continues on next page)

(continued from previous page)

```
        cfg, name='hgroup', id=2, location=[location])
>>> entities.to_json_serializable(hostgroup)
{'location': [{'name': 'loc'}], 'name': 'hgroup', 'id': 2}
>>> mixed = [regular_object, env, hostgroup]
>>> entities.to_json_serializable(mixed)
[
    {'dict': {'objs': [1, 'str', 2.5, '2016-12-13', '2016-12-13']}},
    {'id': 1, 'name': 'env'},
    {'location': [{'name': 'loc'}], 'name': 'hgroup', 'id': 2}
]
>>> import json
>>> json.dumps(entities.to_json_serializable(mixed))
'[{"dict": {"objs": [1, "str", 2.5, "2016-12-13", "2016-12-13"]}}, {"id": 1, "name":
↪"env"}, {"location": [{"name": "loc"}], "name": "hgroup", "id": 2}]'
```

1.6 Using Lower Layers

This section demonstrates how to create a user account. To make things interesting, there are some extra considerations:

- The user account must belong to the organization labeled "Default_Organization".
- The user account must be named "Alice" and have the password "hackme".
- The user account must be created on a pair of satellites.

Two sets of code that accomplish this task are listed. The first body of code shows how to accomplish the task with NailGun. The second body of code does not make use of NailGun, and instead relies entirely on Requests and standard library modules.

```
#!/usr/bin/env python3
"""Create an identical user account on a pair of satellites."""
from pprint import pprint

from nailgun.config import ServerConfig
from nailgun.entities import Organization, User


def main():
    """Create an identical user account on a pair of satellites."""
    server_configs = ServerConfig.get('sat1'), ServerConfig.get('sat2')
    for server_config in server_configs:
        org = Organization(server_config).search(query={'search': 'name="Default_
↪Organization"'})[0]
        # The LDAP authentication source with an ID of 1 is internal. It is
        # nearly guaranteed to exist and be functioning.
        user = User(
            server_config,
            auth_source=1,  # or: AuthSourceLDAP(server_config, id=1),
            login='Alice',
            mail='alice@example.com',
            organization=[org],
            password='hackme',
        ).create()
        pprint(user.get_values())  # e.g. {'login': 'Alice', ...}
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

The code above makes use of NailGun. The code below makes use of Requests and standard library modules.

```
#!/usr/bin/env python3
"""Create an identical user account on a pair of satellites.

If you'd like to test out this script, you can quickly set up an environment
like so::

    python3 -m venv env
    source env/bin/activate
    pip install requests
    ./create_user_plain.py  # copy this script to the current directory

"""
import json
from pprint import pprint
import sys

import requests


def main():
    """Create an identical user account on a pair of satellites."""
    server_configs = (
        {'url': url, 'auth': ('admin', 'changeme'), 'verify': False}
        for url in ('https://sat1.example.com', 'https://sat2.example.com')
    )
    for server_config in server_configs:
        response = requests.post(
            f'{server_config["url"]}/api/v2/users',
            json.dumps(
                {
                    'user': {
                        'auth_source_id': 1,
                        'login': 'Alice',
                        'mail': 'alice@example.com',
                        'organization_ids': [
                            get_organization_id(server_config, 'Default_Organization')
                        ],
                        'password': 'hackme',
                    }
                }
            ),
            auth=server_config['auth'],
            headers={'content-type': 'application/json'},
            verify=server_config['verify'],
        )
        response.raise_for_status()
        pprint(response.json())


def get_organization_id(server_config, label):
```

(continues on next page)

(continued from previous page)

```
    """Return the ID of the organization with label ``label``.

    :param server_config: A dict of information about the server being talked
        to. The dict should include the keys "url", "auth" and "verify".
    :param label: A string label that will be used when searching. Every
        organization should have a unique label.
    :returns: An organization ID. (Typically an integer.)

    """
    response = requests.get(
        f'{server_config["url"]}/katello/api/v2/organizations',
        data=json.dumps({"search": f"label={label}"}),
        auth=server_config["auth"],
        headers={"content-type": "application/json"},
        verify=server_config["verify"],
    )
    response.raise_for_status()
    decoded = response.json()
    if decoded['subtotal'] != 1:
        pprint(
            f'Expected to find one organization, but instead found {decoded["subtotal
↪"]}. '
            f'Search results: {decoded["results"]}'
        )
        sys.exit(1)
    return decoded['results'][0]['id']


if __name__ == '__main__':
    main()
```

What's different between the two scripts?

First, both scripts pass around `server_config` objects (see *nailgun.config.ServerConfig*). However, the NailGun script does not include any hard-coded parameters. Instead, configurations are read from disk. This makes the script more secure (it can be published publicly without any information leakage) and maintainable (server details can change independent of programming logic).

Second, the sans-NailGun script relies entirely on convention when placing values in to and retrieving values from the `server_config` objects. This is easy to get wrong. For example, one piece of code might place a value named `'verify_ssl'` in to a dictionary and a second piece of code might retrieve a value named `'verify'`. This is a mistake, but you won't know about it until runtime. In contrast, the `ServerConfig` objects have an explicit set of possible instance attributes, and tools such as Flake can use this information when linting code. (Similarly, NailGun's entity objects such as `Organization` and `User` have an explicit set of possible instance attributes.) Thus, NailGun allows for more effective static analysis.

Third, NailGun automatically checks HTTP status codes for you when you call methods such as `create`. In contrast, the sans-NailGun script requires that the user call `raise_for_status` or some equivalent every time a response is received. Thus, NailGun makes it harder for undetected errors to creep in to code and cause trouble.

Fourth, there are several hard-coded paths present in the sans-NailGun script: `'/katello/api/v2/organizations'` and `'/api/v2/users'`. This is a hassle. Developers need to look up a path every time they write an API call, and it's easy to make a mistake and waste time troubleshooting the resultant error. NailGun shields the developer from this issue — not a single path is present!

Fifth, the NailGun script shields developers from idiosyncrasies in JSON request formats. Notice how no nested dict is necessary when issuing a GET request for organizations, but a nested dict is necessary when issuing a POST request for users. Differences like this abound. NailGun packages data for you.

Sixth, and perhaps most obviously, the NailGun script is *significantly* shorter! This makes it easier to focus on high-level business logic instead of worrying about implementation details.

CHAPTER 2

API Documentation

This is the NailGun API documentation. It is mostly auto generated from the source code. A good place to start reading is *nailgun*.

The `nailgun` namespace is the public API. The `tests` is not part of the public API, and it is documented here for easy reference for developers.

2.1 nailgun

The root of the NailGun namespace.

NailGun's modules are organized in to a tree of dependencies, where each module only knows about the modules below it in the tree and no module knows about others at the same level in the tree. The modules can be visualized like this:

```
nailgun.entities
└── nailgun.entity_mixins
    ├── nailgun.entity_fields
    ├── nailgun.config
    └── nailgun.client
```

If this is your first time working with NailGun, please read several of the *Examples* before the documentation here.

2.1.1 nailgun.entities

2.1.2 nailgun.entity_mixins

Defines a set of mixins that provide tools for interacting with entities.

exception nailgun.entity_mixins.**BadValueError**
Indicates that an inappropriate value was assigned to an entity.

nailgun.entity_mixins.**CREATE_MISSING = False**
Used by *nailgun.entity_mixins.EntityCreateMixin.create_raw()*.

This is the default value for the `create_missing` argument to *nailgun.entity_mixins.EntityCreateMixin.create_raw()*. Keep in mind that this variable also affects methods which call `create_raw`, such as *nailgun.entity_mixins.EntityCreateMixin.create_json()*.

nailgun.entity_mixins.**DEFAULT_SERVER_CONFIG = None**
A *nailgun.config.ServerConfig* object.

Used by *nailgun.entity_mixins.Entity*.

class nailgun.entity_mixins.**Entity**(*server_config=None*, ***kwargs*)
A representation of a logically related set of API paths.

This class is rather useless as is, and it is intended to be subclassed. Subclasses can specify two useful types of information:

- fields
- metadata

Fields and metadata are represented by the `_fields` and `_meta` instance attributes, respectively. Here is an example of how to define and instantiate an entity:

```
>>> class User(Entity):
...     def __init__(self, server_config=None, **kwargs):
...         self._fields = {
...             'name': StringField(),
...             'supervisor': OneToOneField('User'),
...             'subordinate': OneToManyField('User'),
...         }
...         self._meta = {'api_path': 'api/users'}
...         return super(User, self).__init__(server_config, **kwargs)
...
>>> user = User(
...     name='Alice',
...     supervisor=User(id=1),
...     subordinate=[User(id=3), User(id=4)],
... )
>>> user.name == 'Alice'
True
>>> user.supervisor.id = 1
True
```

The canonical procedure for initializing foreign key fields, shown above, is powerful but verbose. It is tiresome to write statements such as `[User(id=3), User(id=4)]`. As a convenience, entity IDs may be given:

```
>>> User(name='Alice', supervisor=1, subordinate=[3, 4])
>>> user.name == 'Alice'
True
>>> user.supervisor.id = 1
True
```

An entity object is useless if you are unable to use it to communicate with a server. The solution is to provide a *nailgun.config.ServerConfig* when instantiating a new entity.

1. If the `server_config` argument is specified, then that is used.
2. Otherwise, if *nailgun.entity_mixins.DEFAULT_SERVER_CONFIG* is set, then that is used.
3. Otherwise, call *nailgun.config.ServerConfig.get()*.

An entity's server configuration is stored as a private instance variaable and is used by mixin methods, such as *nailgun.entity_mixins.Entity.path()*. For more information on server configuration objects, see *nailgun.config.BaseServerConfig*.

Raises

- ***nailgun.entity_mixins.NoSuchFieldError*** – If a value is assigned to a non-existent field.
- ***nailgun.entity_mixins.BadValueError*** – If an inappropriate value is assigned to a field.

compare(*other*, *filter_fcn=None*)

Returns True if properties can be compared in terms of eq. Entity's Fields can be filtered accordingly to 'filter_fcn'. This callable receives field's name as first parameter and field itself as second parameter. It must return True if field's value should be included on comparison and False otherwise. If not provided field's marked as unique will not be compared by default. 'id' and 'name' are examples of unique fields commonly ignored. Check Entities fields for fields marked with 'unique=True'

Parameters

- **other** – entity to compare
- **filter_fcn** – callable

Returns boolean

entity_with_parent(***parent*)

Returns modified entity by adding parent entity

Parent dict optional, The key/value pair of base entity else fetch from the fields

get_fields()

Return a copy of the fields on the current object.

Returns A dict mapping field names to :class'nailgun.entity_fields.Field' objects.

get_values()

Return a copy of field values on the current object.

This method is almost identical to `vars(self).copy()`. However, only instance attributes that correspond to a field are included in the returned dict.

Returns A dict mapping field names to user-provided values.

path(*which=None*)

Return the path to the current entity.

Return the path to base entities of this entity's type if:

- `which` is `'base'`, or
- `which` is `None` and instance attribute `id` is unset.

Return the path to this exact entity if instance attribute `id` is set and:

- `which` is `'self'`, or
- `which` is `None`.

Raise *NoSuchPathError* otherwise.

Child classes may choose to extend this method, especially if a child entity offers more than the two URLs supported by default. If extended, then the extending class should check for custom parameters before calling `super`:

```
def path(self, which):
    if which == 'custom':
        return urljoin(...)
    super(ChildEntity, self).__init__(which)
```

This will allow the extending method to accept a custom parameter without accidentally raising a *NoSuchPathError*.

Parameters **which** – A string. Optional. Valid arguments are 'self' and 'base'.

Returns A string. A fully qualified URL.

Raises ***nailgun.entity_mixins.NoSuchPathError*** – If no path can be built.

to_json()

Create a JSON encoded string with Entity properties. Ex:

```
>>> from nailgun import entities, config
>>> kwargs = {
...          'id': 1,
...          'name': 'Nailgun Org',
...      }
>>> org = entities.Organization(config.ServerConfig('foo'), \*\*kwargs)
>>> org.to_json()
'{"id": 1, "name": "Nailgun Org"}'
```

Returns str

to_json_dict(*filter_fcn=None*)

Create a dict with Entity properties for json encoding. It can be overridden by subclasses for each standard serialization doesn't work. By default it call _to_json_dict on OneToOne fields and build a list calling the same method on each OneToMany object's fields.

Fields can be filtered accordingly to 'filter_fcn'. This callable receives field's name as first parameter and fields itself as second parameter. It must return True if field's value should be included on dict and False otherwise. If not provided field will not be filtered.

Returns dct

class nailgun.entity_mixins.**EntityCreateMixin**

This mixin provides the ability to create an entity.

The methods provided by this class work together. The call tree looks like this:

```
create
└── create_json
    └── create_raw
        ├── create_missing
        └── create_payload
```

In short, here is what the methods do:

create_missing() Populate required fields with random values. Required fields that already have a value are not populated. This method is not called by default.

create_payload() Assemble a payload of data that can be encoded and sent to the server.

create_raw() Make an HTTP POST request to the server, including the payload.

create_json() Check the server's response for errors and decode the response.

create() Create a *nailgun.entity_mixins.Entity* object representing the created entity and populate its fields with data returned from the server.

See the individual methods for more detailed information.

create(*create_missing=None*)
Create an entity.

Call *create_json()*, use the response to populate a new object of type `type(self)` and return that object.

This method requires that a method named "read" be available on the current object. A method named "read" will be available if *EntityReadMixin* is present in the inheritance tree, and using the method provided by that mixin is the recommended technique for making a "read" method available.

This method makes use of *EntityReadMixin.read()* for two reasons. First, calling that method is simply convenient. Second, the server frequently returns weirdly structured, inconsistently named or straight-up broken responses, and quite a bit of effort has gone in to decoding server responses so *EntityReadMixin.read()* can function correctly. Calling `read` allows this method to re-use the decoding work that has been done for that method.

Returns An instance of type `type(self)`.

Return type *nailgun.entity_mixins.Entity*

Raises `AttributeError` if a method named "read" is not available on the current object.

create_json(*create_missing=None*)
Create an entity.

Call *create_raw()*. Check the response status code, decode JSON and return the decoded JSON as a dict.

Returns A dict. The server's response, with all JSON decoded.

Raises `requests.exceptions.HTTPError` if the response has an HTTP 4XX or 5XX status code.

Raises `ValueError` If the response JSON can not be decoded.

create_missing()
Automagically populate all required instance attributes.

Iterate through the set of all required class *nailgun.entity_fields.Field* defined on `type(self)` and create a corresponding instance attribute if none exists. Subclasses should override this method if there is some relationship between two required fields.

Returns Nothing. This method relies on side-effects.

create_payload()
Create a payload of values that can be sent to the server.

See *_payload()*.

create_raw(*create_missing=None*)
Create an entity.

Possibly call *create_missing()*. Then make an HTTP POST call to `self.path('base')`. The request payload consists of whatever is returned by *create_payload()*. Return the response.

Parameters **create_missing** – Should *create_missing()* be called? In other words, should values be generated for required, empty fields? Defaults to *nailgun.entity_mixins.CREATE_MISSING*.

Returns A `requests.response` object.

class nailgun.entity_mixins.**EntityDeleteMixin**

This mixin provides the ability to delete an entity.

The methods provided by this class work together. The call tree looks like this:

```
delete → delete_raw
```

In short, here is what the methods do:

delete_raw() Make an HTTP DELETE request to the server.

delete() Check the server's response for errors and decode the response.

delete(*synchronous=True*, *timeout=None*)

Delete the current entity.

Call *delete_raw()* and check for an HTTP 4XX or 5XX response. Return either the JSON-decoded response or information about a completed foreman task.

Parameters **synchronous** – A boolean. What should happen if the server returns an HTTP 202 (accepted) status code? Wait for the task to complete if `True`. Immediately return a response otherwise.

Returns A dict. Either the JSON-decoded response or information about a foreman task.

Raises `requests.exceptions.HTTPError` if the response has an HTTP 4XX or 5XX status code.

Raises `ValueError` If an HTTP 202 response is received and the response JSON can not be decoded.

Raises ***nailgun.entity_mixins.TaskTimedOutError*** – If an HTTP 202 response is received, `synchronous is True` and the task times out.

delete_raw()

Delete the current entity.

Make an HTTP DELETE call to `self.path('base')`. Return the response.

Returns A `requests.response` object.

class nailgun.entity_mixins.**EntityReadMixin**

This mixin provides the ability to read an entity.

The methods provided by this class work together. The call tree looks like this:

```
read → read_json → read_raw
```

In short, here is what the methods do:

read_raw() Make an HTTP GET request to the server.

read_json() Check the server's response for errors and decode the response.

read() Create a *nailgun.entity_mixins.Entity* object representing the created entity and populate its fields with data returned from the server.

See the individual methods for more detailed information.

read(*entity=None*, *attrs=None*, *ignore=None*, *params=None*)

Get information about the current entity.

1. Create a new entity of type `type(self)`.
2. Call *read_json()* and capture the response.

3. Populate the entity with the response.
4. Return the entity.

Step one is skipped if the `entity` argument is specified. Step two is skipped if the `attrs` argument is specified. Step three is modified by the `ignore` argument.

All of an entity's one-to-one and one-to-many relationships are populated with objects of the correct type. For example, if `SomeEntity.other_entity` is a one-to-one relationship, this should return `True`:

```
isinstance(
    SomeEntity(id=N).read().other_entity,
    nailgun.entity_mixins.Entity
)
```

Additionally, both of these commands should succeed:

```
SomeEntity(id=N).read().other_entity.id
SomeEntity(id=N).read().other_entity.read().other_attr
```

In the example above, `other_entity.id` is the **only** attribute with a meaningful value. Calling `other_entity.read` populates the remaining entity attributes.

Parameters

- **entity** (`nailgun.entity_mixins.Entity`) – The object to be populated and returned. An object of type `type(self)` by default.
- **attrs** – A dict. Data used to populate the object's attributes. The response from *nailgun.entity_mixins.EntityReadMixin.read_json()* by default.
- **ignore** – A set of attributes which should not be read from the server. This is mainly useful for attributes like a password which are not returned.

Returns An instance of type `type(self)`.

Return type *nailgun.entity_mixins.Entity*

read_json(*params=None*)

Get information about the current entity.

Call *read_raw()*. Check the response status code, decode JSON and return the decoded JSON as a dict.

Returns A dict. The server's response, with all JSON decoded.

Raises `requests.exceptions.HTTPError` if the response has an HTTP 4XX or 5XX status code.

Raises `ValueError` If the response JSON can not be decoded.

read_raw(*params=None*)

Get information about the current entity.

Make an HTTP GET call to `self.path('self')`. Return the response.

Returns A `requests.response` object.

class nailgun.entity_mixins.**EntitySearchMixin**

This mixin provides the ability to search for entities.

The methods provided by this class work together. The call tree looks like this:

```
search
├── search_json
│   └── search_raw
│       └── search_payload
├── search_normalize
└── search_filter
```

In short, here is what the methods do:

search_payload() Assemble a search query that can be encoded and sent to the server.

search_raw() Make an HTTP GET request to the server, including the payload.

search_json() Check the server's response for errors and decode the response.

search_normalize() Normalize search results so they can be used to create new entities.

search() Create one or more *nailgun.entity_mixins.Entity* objects representing the found entities and populate their fields.

search_filter() Read all `entities` and locally filter them.

See the individual methods for more detailed information.

search (*fields=None*, *query=None*, *filters=None*, *path_fields={}*)
Search for entities.

At its simplest, this method searches for all entities of a given kind. For example, to ask for all `nailgun.entities.LifecycleEnvironment` entities:

```
LifecycleEnvironment().search()
```

Values on an entity are used to generate a search query, and the `fields` argument can be used to specify which fields should be used when generating a search query:

```
lc_env = LifecycleEnvironment(name='foo', organization=1)
results = lc_env.search()  # Search by name and organization.
results = lc_env.search({'name', 'organization'})  # Same.
results = lc_env.search({'name'})  # Search by name.
results = lc_env.search({'organization'})  # Search by organization
results = lc_env.search(set())  # Search for all lifecycle envs.
results = lc_env.search({'library'})  # Error!
```

In some cases, the simple search queries that can be generated by NailGun are not sufficient. In this case, you can pass in a raw search query instead. For example, to search for all lifecycle environments with a name of 'foo':

```
LifecycleEnvironment().search(query={'search': 'name="foo"'})
```

The example above is rather pointless: it is easier and more concise to use a generated query. But — and this is a **very** important "but" — the manual search query is melded in to the generated query. This can be used to great effect:

```
LifecycleEnvironment(name='foo').search(query={'per_page': 50})
```

For examples of what the final search queries look like, see *search_payload()*. (That method also accepts the `fields` and `query` arguments.)

In some cases, the server's search facilities may be insufficient, or it may be inordinately difficult to craft a search query. In this case, you can filter search results locally. For example, to ask the server for a list of

all lifecycle environments and then locally search through the results for the lifecycle environment named "foo":

```
LifecycleEnvironment().search(filters={'name': 'foo'})
```

Be warned that filtering locally can be **very** slow. NailGun must `read()` every single entity returned by the server before filtering results. This is because the values used in the filtering process may not have been returned by the server in the initial response to the search.

The fact that all entities are read when `filters` is specified can be used to great effect. For example, this search returns a fully populated list of every single lifecycle environment:

```
LifecycleEnvironment().search(filters={})
```

Parameters

- **fields** – A set naming which fields should be used when generating a search query. If `None`, all values on the entity are used. If an empty set, no values are used.
- **query** – A dict containing a raw search query. This is melded in to the generated search query like so: `{generated: query}.update({manual: query})`.
- **filters** – A dict. Used to filter search results locally.

Returns A list of entities, all of type `type(self)`.

static search_filter(*entities*, *filters*)

Read all `entities` and locally filter them.

This method can be used like so:

```
entities = EntitySearchMixin(entities, {'name': 'foo'})
```

In this example, only entities where `entity.name == 'foo'` holds true are returned. An arbitrary number of field names and values may be provided as filters.

Note: This method calls *`EntityReadMixin.read()`*. As a result, this method only works when called on a class that also inherits from *`EntityReadMixin`*.

Parameters

- **entities** – A list of *`Entity`* objects. All list items should be of the same type.
- **filters** – A dict in the form `{field_name: field_value, ...}`.

Raises ***nailgun.entity_mixins.NoSuchFieldError*** – If any of the fields named in `filters` do not exist on the entities being filtered.

Raises `NotImplementedError` If any of the fields named in `filters` are a *`nailgun.entity_fields.OneToOneField`* or *`nailgun.entity_fields.OneToManyField`*.

search_json(*fields=None*, *query=None*)

Search for entities.

Call *`search_raw()`*. Check the response status code, decode JSON and return the decoded JSON as a dict.

> **Warning:** Subclasses that override this method should not alter the `fields` or `query` arguments. (However, subclasses that override this method may still alter the server's response.) See *search_normalize()* for details.

Parameters

- **fields** – See *search()*.
- **query** – See *search()*.

Returns A dict. The server's response, with all JSON decoded.

Raises `requests.exceptions.HTTPError` if the response has an HTTP 4XX or 5XX status code.

Raises `ValueError` If the response JSON can not be decoded.

search_normalize(*results*)

Normalize search results so they can be used to create new entities.

See *search()* for an example of how to use this method. Here's a simplified example:

```
results = self.search_json()
results = self.search_normalize(results)
entity = SomeEntity(some_cfg, **results[0])
```

At this time, it is possible to parse all search results without knowing what search query was sent to the server. However, it is possible that certain responses can only be parsed if the search query is known. If that is the case, this method will be given a new `payload` argument, where `payload` is the query sent to the server.

As a precaution, the following is highly recommended:

- *search()* may alter `fields` and `query` at will.
- *search_payload()* may alter `fields` and `query` in an idempotent manner.
- No other method should alter `fields` or `query`.

Parameters **results** – A list of dicts, where each dict is a set of attributes for one entity. The contents of these dicts are as is returned from the server.

Returns A list of dicts, where each dict is a set of attributes for one entity. The contents of these dicts have been normalized and can be used to instantiate entities.

search_payload(*fields=None*, *query=None*)

Create a search query.

Do the following:

1. Generate a search query. By default, all values returned by *nailgun.entity_mixins.Entity.get_values()* are used. If `fields` is specified, only the named values are used.
2. Merge `query` in to the generated search query.
3. Return the result.

The rules for generating a search query can be illustrated by example. Let's say that we have an entity with an *nailgun.entity_fields.IntegerField*, a *nailgun.entity_fields.OneToOneField* and a *nailgun.entity_fields.OneToManyField*:

```
>>> some_entity = SomeEntity(id=1, one=2, many=[3, 4])
>>> fields = some_entity.get_fields()
>>> isinstance(fields['id'], IntegerField)
True
>>> isinstance(fields['one'], OneToOneField)
True
>>> isinstance(fields['many'], OneToManyField)
True
```

This method appends "_id" and "_ids" on to the names of each `OneToOneField` and `OneToManyField`, respectively:

```
>>> some_entity.search_payload()
{'id': 1, 'one_id': 2, 'many_ids': [3, 4]}
```

By default, all fields are used. But you can specify a set of field names to use:

```
>>> some_entity.search_payload({'id'})
{'id': 1}
>>> some_entity.search_payload({'one'})
{'one_id': 2}
>>> some_entity.search_payload({'id', 'one'})
{'id': 1, 'one_id': 2}
```

If a `query` is specified, it is merged in to the generated query:

```
>>> some_entity.search_payload(query={'id': 5})
{'id': 5, 'one_id': 2, 'many_ids': [3, 4]}
>>> some_entity.search_payload(query={'per_page': 1000})
{'id': 1, 'one_id': 2, 'many_ids': [3, 4], 'per_page': 1000}
```

Warning: This method currently generates an extremely naive search query that will be wrong in many cases. In addition, Satellite currently accepts invalid search queries without complaint. Make sure to check the API documentation for your version of Satellite against what this method produces.

Parameters

- **fields** – See *search()*.
- **query** – See *search()*.

Returns A dict that can be encoded as JSON and used in a search.

search_raw(*fields=None*, *query=None*)

Search for entities.

Make an HTTP GET call to `self.path('base')`. Return the response.

Warning: Subclasses that override this method should not alter the `fields` or `query` arguments. (However, subclasses that override this method may still alter the server's response.) See *search_normalize()* for details.

Parameters

- **fields** – See *search()*.

- **query** – See *search()*.

Returns A `requests.response` object.

class nailgun.entity_mixins.**EntityUpdateMixin**

This mixin provides the ability to update an entity.

The methods provided by this class work together. The call tree looks like this:

```
update → update_json → update_raw → update_payload
```

In short, here is what the methods do:

update_payload() Assemble a payload of data that can be encoded and sent to the server. Set self._updatable_fields (list of strings) to limit the fields that can be updated.

update_raw() Make an HTTP PUT request to the server, including the payload.

update_json() Check the server's response for errors and decode the response.

update() Create a *nailgun.entity_mixins.Entity* object representing the created entity and populate its fields.

See the individual methods for more detailed information.

update(*fields=None*)

Update the current entity.

Call *update_json()*, use the response to populate a new object of type `type(self)` and return that object.

This method requires that *nailgun.entity_mixins.EntityReadMixin.read()* or some other identical method be available on the current object. A more thorough explanation is available at *nailgun.entity_mixins.EntityCreateMixin.create()*.

Parameters **fields** – An iterable of field names. Only the fields named in this iterable will be updated. No fields are updated if an empty iterable is passed in. All fields are updated if `None` is passed in.

Raises `KeyError` if asked to update a field but no value is available for that field on the current entity.

update_json(*fields=None*)

Update the current entity.

Call *update_raw()*. Check the response status code, decode JSON and return the decoded JSON as a dict.

Parameters **fields** – See *update()*.

Returns A dict consisting of the decoded JSON in the server's response.

Raises `requests.exceptions.HTTPError` if the response has an HTTP 4XX or 5XX status code.

Raises `ValueError` If the response JSON can not be decoded.

update_payload(*fields=None*)

Create a payload of values that can be sent to the server.

By default, this method behaves just like *_payload()*. However, one can also specify a certain set of fields that should be returned. For more information, see *update()*.

update_raw(*fields=None*)
Update the current entity.

Make an HTTP PUT call to `self.path('base')`. The request payload consists of whatever is returned by *update_payload()*. Return the response.

Parameters **fields** – See *update()*.

Returns A `requests.response` object.

exception nailgun.entity_mixins.**MissingValueError**
Indicates that no value can be found for a field.

exception nailgun.entity_mixins.**NoSuchFieldError**
Indicates that the referenced field does not exist.

exception nailgun.entity_mixins.**NoSuchPathError**
Indicates that the requested path cannot be constructed.

nailgun.entity_mixins.**TASK_POLL_RATE = 5**
Default for `poll_rate` argument to *nailgun.entity_mixins._poll_task()*.

nailgun.entity_mixins.**TASK_TIMEOUT = 300**
Default for `timeout` argument to *nailgun.entity_mixins._poll_task()*.

exception nailgun.entity_mixins.**TaskFailedError**(*message*, *task_id*)
Indicates that a task finished with a result other than "success".

exception nailgun.entity_mixins.**TaskTimedOutError**(*message*, *task_id*)
Indicates that a task did not finish before the timout limit.

nailgun.entity_mixins.**_get_entity_id**(*field_name*, *attrs*)
Find the ID for a one to one relationship.

The server may return JSON data in the following forms for a *nailgun.entity_fields.OneToOneField*:

```
'user': None
'user': {'name': 'Alice Hayes', 'login': 'ahayes', 'id': 1}
'user_id': 1
'user_id': None
```

Search `attrs` for a one to one `field_name` and return its ID.

Parameters

- **field_name** – A string. The name of a field.
- **attrs** – A dict. A JSON payload as returned from a server.

Returns Either an entity ID or None.

nailgun.entity_mixins.**_get_entity_ids**(*field_name*, *attrs*)
Find the IDs for a one to many relationship.

The server may return JSON data in the following forms for a *nailgun.entity_fields.OneToManyField*:

```
'user': [{'id': 1, ...}, {'id': 42, ...}]
'users': [{'id': 1, ...}, {'id': 42, ...}]
'user_ids': [1, 42]
```

Search `attrs` for a one to many `field_name` and return its ID.

Parameters

- **field_name** – A string. The name of a field.
- **attrs** – A dict. A JSON payload as returned from a server.

Returns An iterable of entity IDs.

nailgun.entity_mixins.**_get_server_config**()
Search for a *nailgun.config.ServerConfig*.

Returns *nailgun.entity_mixins.DEFAULT_SERVER_CONFIG* if it is not `None`, or whatever is returned by *nailgun.config.ServerConfig.get()* otherwise.

Return type *nailgun.config.ServerConfig*

nailgun.entity_mixins.**_make_entities_from_ids**(*entity_cls*, *entity_objs_and_ids*, *server_config*)
Given an iterable of entities and/or IDs, return a list of entities.

Parameters

- **entity_cls** – An *Entity* subclass.
- **entity_obj_or_id** – An iterable of *nailgun.entity_mixins.Entity* objects and/or entity IDs. All of the entities in this iterable should be of type `entity_cls`.

Returns A list of `entity_cls` objects.

nailgun.entity_mixins.**_make_entity_from_id**(*entity_cls*, *entity_obj_or_id*, *server_config*)
Given an entity object or an ID, return an entity object.

If the value passed in is an object that is a subclass of *Entity*, return that value. Otherwise, create an object of the type that `field` references, give that object an ID of `field_value`, and return that object.

Parameters

- **entity_cls** – An *Entity* subclass.
- **entity_obj_or_id** – Either a *nailgun.entity_mixins.Entity* object or an entity ID.

Returns An `entity_cls` object.

Return type *nailgun.entity_mixins.Entity*

nailgun.entity_mixins.**_payload**(*fields*, *values*)
Implement the `*_payload` methods.

It's frequently useful to create a dict of values that can be encoded to JSON and sent to the server. Unfortunately, there are mismatches between the field names used by NailGun and the field names the server expects. This method provides a default translation that works in many cases. For example:

```
>>> from nailgun.entities import Product
>>> product = Product(name='foo', organization=1)
>>> set(product.get_fields())
{
    'description',
    'gpg_key',
    'id',
    'label',
    'name',
    'organization',
    'sync_plan',
```

(continues on next page)

(continued from previous page)

```
}
>>> set(product.get_values())
{'name', 'organization'}
>>> product.create_payload()
{'organization_id': 1, 'name': 'foo'}
```

Parameters

- **fields** – A value like what is returned by *nailgun.entity_mixins.Entity.get_fields()*.
- **values** – A value like what is returned by *nailgun.entity_mixins.Entity.get_values()*.

Returns A dict mapping field names to field values.

nailgun.entity_mixins.**_poll_task**(*task_id*, *server_config*, *poll_rate=None*, *timeout=None*, *must_succeed=True*)

Implement `nailgun.entities.ForemanTask.poll()`.

See `nailgun.entities.ForemanTask.poll()` for a full description of how this method acts. Other methods may also call this method, such as *nailgun.entity_mixins.EntityDeleteMixin.delete()*.

Certain mixins benefit from being able to poll the server after performing an operation. However, this module cannot use `nailgun.entities.ForemanTask.poll()`, as that would be a circular import. Placing the implementation of `nailgun.entities.ForemanTask.poll()` here allows both that method and the mixins in this module to use the same logic.

nailgun.entity_mixins.**call_entity_method_with_timeout**(*entity_callable*, *timeout=300*, ***kwargs*)

Call Entity callable with a custom timeout

:param entity_callable, the entity method object to call :param timeout: the time to wait for the method call to finish :param kwargs: the kwargs to pass to the entity callable

Usage:

call_entity_method_with_timeout(entities.Repository(id=repo_id).sync, timeout=1500)

nailgun.entity_mixins.**to_json_serializable**(*obj*)

Transforms obj into a json serializable object.

Parameters **obj** – entity or any json serializable object

Returns serializable object

2.1.3 nailgun.entity_fields

The basic components of the NailGun ORM.

Each of the fields in this module corresponds to some type of information that Satellite tracks. When paired the classes in *nailgun.entity_mixins*, it is possible to represent the entities that Satellite manages. For a concrete example of how this works, see *nailgun.entity_mixins.Entity*.

Fields are typically used declaratively in an entity's `__init__` function and are otherwise left untouched, except by the mixin methods. For example, *nailgun.entity_mixins.EntityReadMixin.read()* looks at the fields on an entity to determine what information it should expect the server to return.

A secondary use of fields is to generate random data. For example, you could call `User.get_fields()['login'].gen_value()` to generate a random login. (`gen_value` is implemented at *StringField.gen_value()*) Beware that the `gen_value` methods strive to produce the most outrageous values that are still legal, so they will often return nonsense UTF-8 values, which is unpleasant to work with manually.

class nailgun.entity_fields.**BooleanField**(*required=False*, *choices=None*, *default=<object object>*, *unique=False*, *parent=False*)

Field that represents a boolean

gen_value()

Return a value suitable for a *BooleanField*.

class nailgun.entity_fields.**DateField**(*min_date=None*, *max_date=None*, **args*, ***kwargs*)

Field that represents a date

gen_value()

Return a value suitable for a *DateField*.

class nailgun.entity_fields.**DateTimeField**(*min_date=None*, *max_date=None*, **args*, ***kwargs*)

Field that represents a datetime

gen_value()

Return a value suitable for a *DateTimeField*.

class nailgun.entity_fields.**DictField**(*required=False*, *choices=None*, *default=<object object>*, *unique=False*, *parent=False*)

Field that represents a set of key-value pairs.

gen_value()

Return a value suitable for a *DictField*.

class nailgun.entity_fields.**EmailField**(*required=False*, *choices=None*, *default=<object object>*, *unique=False*, *parent=False*)

Field that represents an email

gen_value()

Return a value suitable for a *EmailField*.

class nailgun.entity_fields.**Field**(*required=False*, *choices=None*, *default=<object object>*, *unique=False*, *parent=False*)

Base class to implement other fields

Record this field's attributes.

Parameters

- **required** – A boolean. Determines whether a value must be submitted to the server when creating or updating an entity.
- **choices** – A tuple of values that this field may be populated with.
- **default** – Entity classes that inherit from *nailgun.entity_mixins.EntityCreateMixin* use this field.
- **unique** – A boolean. Determines if the entity should be unique with its name.
- **parent** – A boolean. Determines if the Entity is a parent entity to one_to_one mapped entity

class nailgun.entity_fields.**FloatField**(*required=False*, *choices=None*, *default=<object object>*, *unique=False*, *parent=False*)

Field that represents a float

gen_value()
Return a value suitable for a *FloatField*.

class nailgun.entity_fields.**IPAddressField**(*length=(1, 30), str_type=('utf8',), *args, **kwargs*)
Field that represents an IP address

gen_value()
Return a value suitable for a *IPAddressField*.

class nailgun.entity_fields.**IntegerField**(*min_val=None, max_val=None, *args, **kwargs*)
Field that represents an integer.

gen_value()
Return a value suitable for a *IntegerField*.

class nailgun.entity_fields.**ListField**(*required=False, choices=None, default=<object object>, unique=False, parent=False*)
Field that represents a list of strings

class nailgun.entity_fields.**MACAddressField**(*length=(1, 30), str_type=('utf8',), *args, **kwargs*)
Field that represents a MAC address

gen_value()
Return a value suitable for a *MACAddressField*.

class nailgun.entity_fields.**NetmaskField**(*length=(1, 30), str_type=('utf8',), *args, **kwargs*)
Field that represents an netmask

gen_value()
Return a value suitable for a *NetmaskField*.

class nailgun.entity_fields.**OneToManyField**(*entity, *args, **kwargs*)
Field that represents a reference to zero or more other entities.

Parameters **entity** (nailgun.entity_mixins.Entity) – The entities to which this field points.

gen_value()
Return the class that this field references.

class nailgun.entity_fields.**OneToOneField**(*entity, *args, **kwargs*)
Field that represents a reference to another entity.

All parameters not documented here are passed to *Field*.

Parameters **entity** (nailgun.entity_mixins.Entity) – The entity to which this field points.

gen_value()
Return the class that this field references.

class nailgun.entity_fields.**StringField**(*length=(1, 30), str_type=('utf8',), *args, **kwargs*)
Field that represents a string.

The default `length` of string fields is short for two reasons:

1. Foreman's database backend limits many fields to 255 bytes in length. As a result, `length` should be no longer than 85 characters long, as 85 unicode characters may be up to 255 bytes long.

2. Humans have to read through the error messages produced by this library. Long error messages are hard to read through, and that hurts productivity. Thus, a `length` even shorter than 85 chars is desirable.

Parameters

- **length** – Either a `(min_len, max_len)` tuple or an `exact_len` integer.
- **str_type** – The types of characters to generate when *`StringField.gen_value()`* is called. May be a single string type (e.g. `'utf8'`) or a tuple of string types. This argument is passed through to FauxFactory's `gen_string` method, so this method accepts all string types which that method does.

gen_value()
Return a value suitable for a *`StringField`*.

class nailgun.entity_fields.**URLField**(*scheme=None*, **args*, ***kwargs*)
Field that represents an URL

Parameters **scheme** (*str*) – The URL scheme can be one of ['http', 'https', 'ftp']

gen_value()
Return a value suitable for a *`URLField`*.

2.1.4 nailgun.config

Tools for managing and presenting server connection configurations.

NailGun needs to know certain facts about the remote server in order to do anything useful. For example, NailGun needs to know the URL of the remote server (e.g. 'https://example.com:250') and how to authenticate with the remote server. *`nailgun.config.ServerConfig`* eases the task of managing and presenting that information.

class nailgun.config.**BaseServerConfig**(*url*, *auth=None*, *version=None*)
A set of facts for communicating with a Satellite server.

This object stores a set of facts that can be used when communicating with a Satellite server, regardless of whether that communication takes place via the API, CLI or UI. *`nailgun.config.ServerConfig`* is more specialized and adds attributes that are useful when communicating with the API.

Parameters

- **url** – A string. The URL of a server. For example: *'https://example.com:250'*.
- **auth** – Credentials to use when communicating with the server. For example: *('username', 'password')*. No instance attribute is created if no value is provided.
- **version** – A string, such as `'6.0'` or `'6.1'`, indicating the Satellite version the server is running. This version number is parsed by `packaging.version.parse` before being stored locally. This allows for version comparisons:

```
>>> from nailgun.config import ServerConfig
>>> from packaging.version import parse
>>> cfg = ServerConfig('http://sat.example.com', version='6.0')
>>> cfg.version == parse('6.0')
True
>>> cfg.version == parse('6.0.0')
True
>>> cfg.version < parse('10.0')
True
>>> '6.0' < '10.0'
False
```

If no version number is provided, then no instance attribute is created, and it is assumed that the server is running an up-to-date nightly build.

Warning: This class will likely be moved to a separate Python package in a future release of NailGun. Be careful about making references to this class, as those references will likely need to be changed.

classmethod delete(*label='default'*, *path=None*)
Delete a server configuration.

This method is thread safe.

Parameters

- **label** – A string. The configuration identified by `label` is deleted.
- **path** – A string. The configuration file to be manipulated. Defaults to what is returned by *nailgun.config._get_config_file_path()*.

Returns `None`

classmethod get(*label='default'*, *path=None*)
Read a server configuration from a configuration file.

Parameters

- **label** – A string. The configuration identified by `label` is read.
- **path** – A string. The configuration file to be manipulated. Defaults to what is returned by *nailgun.config._get_config_file_path()*.

Returns A brand new *nailgun.config.BaseServerConfig* object whose attributes have been populated as appropriate.

Return type *BaseServerConfig*

classmethod get_labels(*path=None*)
Get all server configuration labels.

Parameters **path** – A string. The configuration file to be manipulated. Defaults to what is returned by *nailgun.config._get_config_file_path()*.

Returns Server configuration labels, where each label is a string.

save(*label='default'*, *path=None*)
Save the current connection configuration to a file.

This method is thread safe.

Parameters

- **label** – A string. An identifier for the current configuration. This allows multiple configurations with unique labels to be saved in a single file. If a configuration identified by `label` already exists in the destination configuration file, it is replaced.
- **path** – A string. The configuration file to be manipulated. By default, an XDG-compliant configuration file is used. A configuration file is created if one does not exist already.

Returns `None`

exception nailgun.config.**ConfigFileError**
Indicates an error occurred when locating a configuration file.

> **Warning:** This class will likely be moved to a separate Python package in a future release of NailGun. Be careful about making references to this class, as those references will likely need to be changed.

class nailgun.config.**ServerConfig**(*url*, *auth=None*, *version=None*, *verify=None*)

Extend *nailgun.config.BaseServerConfig*.

This class adds functionality that is useful specifically when working with the API. For example, it stores the additional `verify` instance attribute and adds logic useful for presenting information to the methods in *nailgun.client*.

Parameters **verify** – A boolean. Should SSL be verified when communicating with the server? No instance attribute is created if no value is provided.

classmethod get(*label='default'*, *path=None*)

Read a server configuration from a configuration file.

This method extends *nailgun.config.BaseServerConfig.get()*. Please read up on that method before trying to understand this one.

The entity classes rely on the requests library to be a transport mechanism. The methods provided by that library, such as `get` and `post`, accept an `auth` argument. That argument must be a tuple:

> Auth tuple to enable Basic/Digest/Custom HTTP Auth.

However, the JSON decoder does not recognize a tuple as a type, and represents sequences of elements as a tuple. Compensate for that by converting `auth` to a two element tuple if it is a two element list.

This override is done here, and not in the base class, because the base class may be extracted out into a separate library and used in other contexts. In those contexts, the presence of a list may not matter or may be desirable.

get_client_kwargs()

Get kwargs for use with the methods in *nailgun.client*.

This method returns a dict of attributes that can be unpacked and used as kwargs via the `**` operator. For example:

```
cfg = ServerConfig.get()
client.get(f'{cfg.url}/api/v2', **cfg.get_client_kwargs())
```

This method is useful because client code may not know which attributes should be passed from a `ServerConfig` object to one of the `nailgun.client` functions. Consider that the example above could also be written like this:

```
cfg = ServerConfig.get()
client.get(f'{cfg.url}/api/v2', auth=cfg.auth, verify=cfg.verify)
```

But this latter approach is more fragile. It will break if `cfg` does not have an `auth` or `verify` attribute.

nailgun.config.**_get_config_file_path**(*xdg_config_dir*, *xdg_config_file*)

Search `XDG_CONFIG_DIRS` for a config file and return the first found.

Search each of the standard XDG configuration directories for a configuration file. Return as soon as a configuration file is found. Beware that by the time client code attempts to open the file, it may be gone or otherwise inaccessible.

Parameters

- **xdg_config_dir** – A string. The name of the directory that is suffixed to the end of each of the `XDG_CONFIG_DIRS` paths.

- **xdg_config_file** – A string. The name of the configuration file that is being searched for.

Returns A `str` path to a configuration file.

Raises *nailgun.config.ConfigFileError* – When no configuration file can be found.

2.1.5 nailgun.client

Wrappers for methods in the Requests module.

The functions in this module wrap functions from the Requests module. Each function is modified with the following behaviours:

1. It sets the 'content-type' header to 'application/json', so long as no content-type is already set.
2. It encodes its `data` argument as JSON (using the `json` module) if the 'content-type' header is 'application/json'.
3. It logs information about the request before it is sent.
4. It logs information about the response when it is received.

`nailgun.client.`**`delete`**(*url*, ***kwargs*)
A wrapper for `requests.delete`. Sends a DELETE request.

`nailgun.client.`**`get`**(*url*, *params=None*, ***kwargs*)
A wrapper for `requests.get`.

`nailgun.client.`**`head`**(*url*, ***kwargs*)
A wrapper for `requests.head`.

`nailgun.client.`**`patch`**(*url*, *data=None*, ***kwargs*)
A wrapper for `requests.patch`. Sends a PATCH request.

`nailgun.client.`**`post`**(*url*, *data=None*, *json=None*, ***kwargs*)
A wrapper for `requests.post`.

`nailgun.client.`**`put`**(*url*, *data=None*, ***kwargs*)
A wrapper for `requests.put`. Sends a PUT request.

`nailgun.client.`**`request`**(*method*, *url*, ***kwargs*)
A wrapper for `requests.request`.

2.2 tests

Unit tests for NailGun.

2.2.1 tests.test_client

Unit tests for *nailgun.client*.

class `tests.test_client.`**`ClientTestCase`**(*methodName='runTest'*)
Tests for functions in *nailgun.client*.

`setUp`()
Hook method for setting up the test fixture before exercising it.

test_client_request()
Test *nailgun.client.request()*.

Make the same assertions as *tests.test_client.ClientTestCase.test_clients()*.

test_clients()
Test all the wrappers except *nailgun.client.request()*.

The following functions are tested:

- *nailgun.client.delete()*
- *nailgun.client.get()*
- *nailgun.client.head()*
- *nailgun.client.patch()*
- *nailgun.client.post()*
- *nailgun.client.put()*

Assert that:

- The wrapper function passes the correct parameters to requests.
- The wrapper function returns whatever requests returns.

test_identical_args()
Check that the wrapper functions have the correct signatures.

For example, *nailgun.client.delete()* should have the same signature as `requests.delete`.

class tests.test_client.**ContentTypeIsJsonTestCase**(*methodName='runTest'*)
Tests for function `_content_type_is_json`.

test_false()
Assert `True` is returned when content-type is not JSON.

test_false_with_no_headers()
If no headers passed should return None

test_true()
Assert `True` is returned when content-type is JSON.

class tests.test_client.**SetContentTypeTestCase**(*methodName='runTest'*)
Tests for function `_set_content_type`.

test_existing_value()
Assert that no content-type is provided if one is set.

test_files_in_kwargs()
Assert that no content-type is provided if files are given.

test_no_value()
Assert that a content-type is provided if none is set.

2.2.2 tests.test_config

Unit tests for *nailgun.config*.

class tests.test_config.**BaseServerConfigTestCase**(*methodName='runTest'*)
Tests for *nailgun.config.BaseServerConfig*.

test_delete()
Test *nailgun.config.BaseServerConfig.delete()*.

Assert that the method reads the config file before writing, and that it writes out a correct config file.

test_get()
Test *nailgun.config.BaseServerConfig.get()*.

Assert that the method extracts the asked-for section from a configuration file and correctly populates a new `BaseServerConfig` object. Also assert that the `auth` attribute is a list. (See the docstring for *nailgun.config.ServerConfig.get()*.)

test_get_labels()
Test *nailgun.config.BaseServerConfig.get_labels()*.

Assert that the method returns the correct labels.

test_init()
Test instantiating *nailgun.config.BaseServerConfig*.

Assert that only provided values become object attributes.

test_init_invalid()
Test instantiating :class: *nailgun.config.BaseServerConfig*. Assert that configs with invalid versions do not load.

test_save()
Test *nailgun.config.BaseServerConfig.save()*.

Assert that the method reads the config file before writing, and that it writes out a correct config file.

class tests.test_config.**ReprTestCase**(*methodName='runTest'*)
Test method `nailgun.config.BaseServerConfig.__repr__`.

test_bsc_v1()
Test *nailgun.config.BaseServerConfig*.

Assert that `__repr__` works correctly when `url` is specified.

test_bsc_v2()
Test *nailgun.config.BaseServerConfig*.

Assert that `__repr__` works correctly when `url` and `auth` are specified.

test_bsc_v3()
Test *nailgun.config.BaseServerConfig*.

Assert that `__repr__` works correctly when `url` and `version` are specified.

test_sc_v1()
Test *nailgun.config.ServerConfig*.

Assert that `__repr__` works correctly when only a URL is passed in.

test_sc_v2()
Test *nailgun.config.ServerConfig*.

Assert that `__repr__` works correctly when `url` and `auth` are specified.

test_sc_v3()
Test *nailgun.config.ServerConfig*.

Assert that `__repr__` works correctly when `url` and `version` are specified.

test_sc_v4()
Test *nailgun.config.ServerConfig*.

Assert that __repr__ works correctly when `url` and `verify` are specified.

class tests.test_config.**ServerConfigTestCase**(*methodName='runTest'*)
Tests for *nailgun.config.ServerConfig*.

test_get()
Test *nailgun.config.ServerConfig.get()*.

Assert that the `auth` attribute is a tuple.

test_get_client_kwargs()
Test *nailgun.config.ServerConfig.get_client_kwargs()*.

Assert that:

- `get_client_kwargs` returns all of the instance attributes from its object except the "url" attribute, and
- no instance attributes from the object are removed.

test_init()
Test instantiating *nailgun.config.ServerConfig*.

Assert that only provided values become object attributes.

test_raise_config_file_error()
Should raise error if path not found

2.2.3 tests.test_entities

2.2.4 tests.test_entity_fields

Unit tests for *nailgun.entity_fields*.

class tests.test_entity_fields.**GenValueTestCase**(*methodName='runTest'*)
Tests for the `gen_value` method on various `*Field` classes.

Classes with complex `gen_value` implementations are broken out into separate test cases.

test_boolean_field()
Test *nailgun.entity_fields.BooleanField.gen_value()*.

test_date_field()
Test *nailgun.entity_fields.DateField.gen_value()*.

test_datetime_field()
Test *nailgun.entity_fields.DateTimeField.gen_value()*.

test_dict_field()
Test *nailgun.entity_fields.DictField.gen_value()*.

Assert that an empty dict is returned by default. There are very few occurrences of dict fields in the entity classes, so it is hard to intelligently produce a randomized value that will be of use in a wide variety of entities. Instead, those few entities override or extend this method.

test_email_field()
Test *nailgun.entity_fields.EmailField.gen_value()*.

Ensure *nailgun.entity_fields.EmailField.gen_value()* returns a unicode string containing the character '@'.

test_float_field()
Test *nailgun.entity_fields.FloatField.gen_value()*.

test_gen_netmask()
Test *nailgun.entity_fields.NetmaskField.gen_value()*.

Assert that the result is in `fauxfactory.constants.VALID_NETMASKS`.

test_ip_address_field()
Test *nailgun.entity_fields.IPAddressField.gen_value()*.

Ensure the value returned is acceptable to `socket.inet_aton`.

test_mac_address_field()
Test *nailgun.entity_fields.MACAddressField.gen_value()*.

Ensure the value returned is a string containing 12 hex digits (either upper or lower case), grouped into pairs of digits and separated by colon characters. For example: `'01:23:45:FE:dc:BA'`

The regex used in this test is inspired by this Q&A: http://stackoverflow.com/questions/7629643/how-do-i-validate-the-format-of-a-mac-address

test_one_to_many_field()
Test *nailgun.entity_fields.OneToManyField.gen_value()*.

test_one_to_one_field()
Test *nailgun.entity_fields.OneToOneField.gen_value()*.

test_url_field()
Test *nailgun.entity_fields.URLField.gen_value()*.

Check that the result can be parsed by the urlparse/urllib.parse module and that the resultant object has a `netloc` attribute.

class tests.test_entity_fields.**IntegerFieldTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_fields.IntegerField*.

test_int_is_returned()
Enture the value returned is an `int`.

test_max_val_arg()
Ensure that the `max_val` argument is respected.

test_min_val_arg()
Ensure that the `min_val` argument is respected.

test_min_val_max_val_args()
Ensure that the `min_val` and `max_val` args are respected.

class tests.test_entity_fields.**StringFieldTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_fields.StringField*.

test_length_arg()
Ensure that the `length` argument is respected.

test_str_is_returned()
Ensure a unicode string at least 1 char long is returned.

test_str_type_arg()
Ensure that the `str_type` argument is respected.

class tests.test_entity_fields.**TestClass**
A class that is used when testing the OneTo{One,Many}Field classes.

2.2.5 tests.test_entity_mixins

Tests for *nailgun.entity_mixins*.

class tests.test_entity_mixins.**EntityCreateMixinTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins.EntityCreateMixin*.

setUp()
Set `self.entity = EntityWithCreate(...)`.

test_create()
Test *nailgun.entity_mixins.EntityCreateMixin.create()*.

test_create_json()
Test *nailgun.entity_mixins.EntityCreateMixin.create_json()*.

test_create_missing()
Call method `create_missing`.

test_create_raw_v1()
What happens if the `create_missing` arg is not specified?

nailgun.entity_mixins.EntityCreateMixin.create_raw() should default to *nailgun.entity_mixins.CREATE_MISSING*. We do not set `CREATE_MISSING` in this test. It is a process-wide variable, and setting it may prevent tests from being run in parallel.

test_create_raw_v2()
What happens if the `create_missing` arg is `True`?

test_create_raw_v3()
What happens if the `create_missing` arg is `False`?

class tests.test_entity_mixins.**EntityDeleteMixinTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins.EntityDeleteMixin*.

setUp()
Set `self.entity = EntityWithDelete(...)`.

test_delete_raw()
Call *nailgun.entity_mixins.EntityDeleteMixin.delete_raw()*.

test_delete_v1()
What happens if the server returns an error HTTP status code?

test_delete_v2()
What happens if the server returns an HTTP ACCEPTED status code?

test_delete_v3()
What happens if the server returns an HTTP NO_CONTENT status?

test_delete_v4()
What happens if the server returns some other stuccess status?

test_delete_v5()
What happens if the server returns an HTTP OK status and empty content?

test_delete_v6()
What happens if the server returns an HTTP OK status and blank only content?

class tests.test_entity_mixins.**EntityReadMixinTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins.EntityReadMixin*.

setUp()
Set `self.entity = EntityWithRead(...)`.

classmethod setUpClass()
Set `cls.test_entity`.

`test_entity` is a class having one to one and one to many fields.

test_missing_value_error()
Raise a *nailgun.entity_mixins.MissingValueError*.

test_read_json()
Call *nailgun.entity_mixins.EntityReadMixin.read_json()*.

test_read_raw()
Call *nailgun.entity_mixins.EntityReadMixin.read_raw()*.

test_read_v1()
Make `read_json` return hashes.

test_read_v2()
Make `read_json` return hashes, but with different field names.

test_read_v3()
Make `read_json` return IDs.

test_read_v4()
Do not ignore any fields.

class tests.test_entity_mixins.**EntitySearchMixinTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins.EntitySearchMixin*.

setUp()
Set `self.cfg` and `self.entity`.

test_search_filter_v1()
Test *nailgun.entity_mixins.EntitySearchMixin.search_filter()*.

Pass a zero-length list of entities.

test_search_filter_v2()
Test *nailgun.entity_mixins.EntitySearchMixin.search_filter()*.

Try to filter on a foreign key field.

test_search_filter_v3()
Test *nailgun.entity_mixins.EntitySearchMixin.search_filter()*.

Pass an invalid filter.

test_search_filter_v4()
Test *nailgun.entity_mixins.EntitySearchMixin.search_filter()*.

Pass in valid entities and filters.

test_search_json()
Call *nailgun.entity_mixins.EntitySearchMixin.search_json()*.

test_search_normalize_v1()
Call `search_normalize`.

Pretend the server returns values for all fields, and an extra value.

test_search_normalize_v2()
Call `search_normalize`.

Pretend the server returns no values for any fields.

test_search_payload_v1()
Call search_payload. Generate an empty query.

test_search_payload_v2()
Call search_payload. Pass in a query.

test_search_payload_v3()
Call search_payload. Include a variety of fields in a search.

test_search_raw()
Call *nailgun.entity_mixins.EntitySearchMixin.search_raw()*.

test_search_v1()
Test *nailgun.entity_mixins.EntitySearchMixin.search()*.

Pass no arguments.

test_search_v2()
Test *nailgun.entity_mixins.EntitySearchMixin.search()*.

Provide each possible argument.

class tests.test_entity_mixins.**EntityTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins.Entity*.

setUp()
Set self.cfg.

test_bad_value_error()
Try to raise a *nailgun.entity_mixins.BadValueError*.

test_compare()
Assert compare take only not unique fields into account

test_compare_to_null()
Assert entity comparison to None

test_compare_with_filter()
Assert compare can filter fields based on callable

test_entity_get_fields()
Test *nailgun.entity_mixins.Entity.get_fields()*.

test_entity_get_values()
Test *nailgun.entity_mixins.Entity.get_values()*.

test_entity_get_values_v2()
Test *nailgun.entity_mixins.Entity.get_values()*, ensure _path_fields are never returned.

test_eq()
Test method nailgun.entity_mixins.Entity.__eq__.

Assert that __eq__ works comparing all attributes, even from nested structures.

test_eq_none()
Test method nailgun.entity_mixins.Entity.__eq__ against None

Assert that __eq__ returns False when compared to None.

test_init_v1()
Provide no value for the server_config argument.

test_init_v2()
Provide a server config object via DEFAULT_SERVER_CONFIG.

test_no_such_field_error()
Try to raise a *nailgun.entity_mixins.NoSuchFieldError*.

test_path()
Test *nailgun.entity_mixins.Entity.path()*.

test_repr_v1()
Test method nailgun.entity_mixins.Entity.__repr__.

Assert that __repr__ works correctly when no arguments are passed to an entity.

test_repr_v2()
Test method nailgun.entity_mixins.Entity.__repr__.

Assert that __repr__ works correctly when an ID is passed to an entity.

test_repr_v3()
Test method nailgun.entity_mixins.Entity.__repr__.

Assert that __repr__ works correctly when one entity has a foreign key relationship to a second entity.

class tests.test_entity_mixins.**EntityUpdateMixinTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins.EntityUpdateMixin*.

setUp()
Set self.entity = EntityWithUpdate(...).

test_update()
Test *nailgun.entity_mixins.EntityUpdateMixin.update()*.

test_update_json()
Call *nailgun.entity_mixins.EntityUpdateMixin.update_json()*.

test_update_payload_v1()
Call *nailgun.entity_mixins.EntityUpdateMixin.update_payload()*.

Assert that the method behaves correctly given various values for the field argument.

test_update_payload_v2()
Call *nailgun.entity_mixins.EntityUpdateMixin.update_payload()*.

Assign None to a OneToOneField and call update_payload.

test_update_raw()
Call *nailgun.entity_mixins.EntityUpdateMixin.update_raw()*.

class tests.test_entity_mixins.**EntityWithCreate**(*server_config=None*, ***kwargs*)
Inherits from *nailgun.entity_mixins.EntityCreateMixin*.

class tests.test_entity_mixins.**EntityWithDelete**(*server_config=None*, ***kwargs*)
Inherits from *nailgun.entity_mixins.EntityDeleteMixin*.

class tests.test_entity_mixins.**EntityWithRead**(*server_config=None*, ***kwargs*)
Inherits from *nailgun.entity_mixins.EntityReadMixin*.

class tests.test_entity_mixins.**EntityWithSearch**(*server_config=None*, ***kwargs*)
Inherits from *nailgun.entity_mixins.EntitySearchMixin*.

class tests.test_entity_mixins.**EntityWithSearch2**(*server_config=None*, ***kwargs*)
An entity with integer, one to one and one to many fields.

class tests.test_entity_mixins.**EntityWithUpdate**(*server_config=None*, ***kwargs*)
Inherits from *nailgun.entity_mixins.EntityUpdateMixin*.

class tests.test_entity_mixins.**MakeEntitiesFromIdsTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins._make_entities_from_ids()*.

setUp()
Set `self.cfg`.

test_pass_in_both()
Let `entity_objs_and_ids` be an iterable of integers and IDs.

test_pass_in_emtpy_iterable()
Let the `entity_objs_and_ids` argument be an empty iterable.

test_pass_in_entity_ids()
Let the `entity_objs_and_ids` arg be an iterable of integers.

test_pass_in_entity_obj()
Let the `entity_objs_and_ids` arg be an iterable of entities.

class tests.test_entity_mixins.**MakeEntityFromIdTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins._make_entity_from_id()*.

setUp()
Set `self.cfg`.

test_pass_in_entity_id()
Let the `entity_obj_or_id` argument be an integer.

test_pass_in_entity_obj()
Let the `entity_obj_or_id` argument be an entity object.

class tests.test_entity_mixins.**PollTaskTestCase**(*methodName='runTest'*)
Tests for *nailgun.entity_mixins._poll_task()*.

setUp()
Create a bogus server configuration object.

test__poll_task_failure()
What happens when a foreman task completes but does not succeed?

Assert that a *nailgun.entity_mixins.TaskFailedError* exception is raised.

test__poll_task_success()
What happens when a foreman task completes and does succeed?

Assert that the server's response is returned.

test__poll_task_timeout()
What happens when a foreman task timesout? Assert that the task is still running.

class tests.test_entity_mixins.**SampleEntity**(*server_config=None*, ***kwargs*)
Sample entity to be used in the tests

class tests.test_entity_mixins.**SampleEntityThree**(*server_config=None*, ***kwargs*)
An entity with foreign key fields as One to One and ListField.

This class has a *nailgun.entity_fields.OneToOneField* called "one_to_one" pointing to *tests.test_entity_mixins.SampleEntityTwo*.

This class has a *nailgun.entity_fields.ListField* called "list" containing instances of *tests.test_entity_mixins.SampleEntity*.

class tests.test_entity_mixins.**SampleEntityTwo**(*server_config=None*, ***kwargs*)
An entity with foreign key fields.

This class has a *nailgun.entity_fields.OneToManyField* called "one_to_many" pointing to *tests.test_entity_mixins.SampleEntity*.

CHAPTER 3

Quick Start

This script demonstrates how to create and delete an organization, and how to save some of our work for later re-use:

```
>>> from nailgun.config import ServerConfig
>>> from nailgun.entities import Organization
>>> server_config = ServerConfig(
...     auth=('admin', 'changeme'),      # Use these credentials...
...     url='https://sat1.example.com',  # ...to talk to this server.
... )  # More options are available, e.g. disabling SSL verification.
>>> org = Organization(server_config, name='junk org').create()
>>> org.name == 'junk org'  # Access all attrs likewise, e.g. `org.label`
True
>>> org.delete()
>>> server_config.save()  # Save to disk w/label 'default'. Read with get()
```

This example glosses over *many* features. The *Examples* and *API documentation* sections provide more in-depth documentation.

CHAPTER 4

Why NailGun?

NailGun exists to make working with the Satellite 6 API easier. Here are some of the challenges developers face:

- Existing libraries, such as the Python Requests library, are general purpose tools. As a result, client code can easily become excessively verbose. See the *Examples* document for an example.
- The Satellite 6 API is not RESTful in its design. As a result, even experienced developers may find the API hard to work with.
- The Satellite 6 API is not consistent in its implementation. For example, see the "Payload Generation" section of this blog post.

All of the above issues are compounded by the size of the Satellite 6 API. As of this writing, there are 405 paths. This makes it tough to design compact and elegant client code.

NailGun addresses these issues. NailGun is specialized, it has a consistent design, it abstracts away many painful implementation details and it contains workarounds for certain bugs. Why use a hammer when you can use a nail gun?

CHAPTER 5

Scope and Limitations

NailGun is not an officially supported product. NailGun is a Python-only library, and integration with other languages such as Java or Ruby is not currently a consideration. Although NailGun is developed with a broad audience in mind, it targets Robottelo first and foremost.

NailGun was originally conceived as a set of helper routines internal to Robottelo. It has since been extracted from that code base and turned in to an independently useful library.

Warning: Until version 1.0 is released, functionality will be incomplete, and breaking changes may be introduced. Users are advised to read the release notes closely.

CHAPTER 6

Resources

The *Examples* and *API documentation* sections provide more in-depth documentation.

Join the #robottelo channel on the freenode IRC network to chat with a human. The Robottelo source code contains many real-world examples how NailGun is used, especially the tests/foreman/api/ directory. This blog post provides a glimpse in to the challenges that NailGun is designed to overcome.

CHAPTER 7

Contributing

Contributions are encouraged. The easiest way to contribute is to submit a pull request on GitHub, but patches are welcome no matter how they arrive.

You can use pip and make to quickly set up a development environment:

```
pip install -r requirements.txt -r requirements-dev.txt
pre-commit install-hooks
make test
make docs-html
```

Please adhere to the following guidelines:

- All PR's should follow the predetermined pull request template and explain the problem that is addressed. Issues should follow template and explain what the problem is.
- **Maintain Coding Standards**
  - Keep pep8 rules
  - **Follow the same stylistic and logical patterns used in the code**
    - All entity class names and class attributes have to be in the singular format
    - All required entity attributes have to have *required=True* parameter
    - It is preferable to use *alpha* data type for default string values for easier debug procedure
    - In case any workaround is introduced, it is necessary to provide corresponding BZ ID directly into the code docstring
    - All linting (flake8) and formatting/style checks would be enforced by Travis-CI and PR would be considered broken until checks are passed successfully.
    - Use of pre-commit configuration included with repo will ensure style compliance locally before commit, helping reduce travis failures.
- **Adhere to typical commit guidelines:**

  - Commits should not cause NailGun's unit test to fail. If it does, it will the responsibility of contributor to review those failures and fix them in the same PR's or raise another. The tracking of failures would be responsibility of contributor.
  - Commits should be small and coherent. One commit should address one issue.
  - Commits should have good commit messages.
  - Rebasing is encouraged. Rebasing produces a much nicer commit history than merging.
- To make the review process easy for all reviewers and anyone else interested in the new functionality, please provide some output making use of your changes. Having example of usage in docstring along with your code could really help others to build up on your code. You can add log from Python interactive shell or some tests results (from Robottelo / Foreman Ansible Modules) in PR message, or you can do something completely different - as long as it runs your code, it's fine!
- If PR is applicable for many branches (e.g. master and one of '6.X.z' branches), specify that information in PR message
- **Unit tests**
  - Unit tests are compulsory
  - Unit tests should cover all available actions, for the entity. For eg: Repository Sets, have enable, disable, list_available there should be unit tests exercising these actions.
- When in doubt, ask on IRC. Join #robottelo on freenode

Important to Note :

- **Define Foreman Version labels in Nailgun** if possible, the contributor should set the right version.
- **All PRs should be raised along with Unit tests** The unit tests should be added while adding a new entity or modifying the existing entity or modifying and adding to the core of Nailgun.
- **Test results from upstream devel or from upstream nightly** The API call results are required from PR author to make the review process more firm. Author can provide results from any library that uses the contributed code by running the changes on upstream nightly or from his/her devel box. The interactive python shell output would be acceptable as well.

7.1 Nailgun Review Process

- Travis CI is run, and any issues are resolved by contributor.
- If deemed necessary by contributors/reviewers, an automation run is triggered.
- At least two ACKs are required to merge a pull request.
- At least one ACK must be from a Tier 2 reviewer.
- If a PR requires changes to the CI environment, the "CI Owner" must also provide an ACK.
- Pull request can be merged only when all comments are in resolved state (Resolve conversation button is pressed)

Reviewers & Responsibilities :

- **Both Tiers**
  - Consistently check your projects for new pull requests.
  - Check code for consistency with project guidelines.
  - Pin code dependencies (external libraries), against the version it was tested.

  - Determine if CI and/or test infrastructure changes are required.
  - Provide helpful feedback.
  - Follow-up with any pending feedback, to ensure the PR is resolved quickly.

- **Tier 1 Reviewer**
  - Check the scenarios are valid for the feature or components
  - Suggestions on the feature that can be covered with minimal code additions/changes.
- **Tier 2 Reviewer**
  - Check for logical errors.
  - Guide the contributor on how to fix mistakes and any other improvements.
  - Ideally if not done by contributor, identify code that may impact third-party projects (e.g Nailgun -> Robottelo , FAM), file issues if PR causes breakages in relevant projects

7.2 Nailgun Release Process

Projects that require nailgun, would often rely on the released Nailgun from Pypi. We intended to make the release process more formal and standard to deliver timely and stable code base to consumer projects.

- Nailgun Releases should be performed against stable branches.
- No historical release support.
- Nailgun will follow request based minor releases.

Python Module Index

n

nailgun, 15
nailgun.client, 35
nailgun.config, 32
nailgun.entity_fields, 29
nailgun.entity_mixins, 15

t

tests, 35
tests.test_client, 35
tests.test_config, 36
tests.test_entity_fields, 38
tests.test_entity_mixins, 40

Index

Symbols

_get_config_file_path() (*in module nailgun.config*), 34
_get_entity_id() (*in module nailgun.entity_mixins*), 27
_get_entity_ids() (*in module nailgun.entity_mixins*), 27
_get_server_config() (*in module nailgun.entity_mixins*), 28
_make_entities_from_ids() (*in module nailgun.entity_mixins*), 28
_make_entity_from_id() (*in module nailgun.entity_mixins*), 28
_payload() (*in module nailgun.entity_mixins*), 28
_poll_task() (*in module nailgun.entity_mixins*), 29

B

BadValueError, 15
BaseServerConfig (*class in nailgun.config*), 32
BaseServerConfigTestCase (*class in tests.test_config*), 36
BooleanField (*class in nailgun.entity_fields*), 30

C

call_entity_method_with_timeout() (*in module nailgun.entity_mixins*), 29
ClientTestCase (*class in tests.test_client*), 35
compare() (*nailgun.entity_mixins.Entity method*), 17
ConfigFileError, 33
ContentTypeIsJsonTestCase (*class in tests.test_client*), 36
create() (*nailgun.entity_mixins.EntityCreateMixin method*), 19
create_json() (*nailgun.entity_mixins.EntityCreateMixin method*), 19
CREATE_MISSING (*in module nailgun.entity_mixins*), 15
create_missing() (*nailgun.entity_mixins.EntityCreateMixin method*), 19
create_payload() (*nailgun.entity_mixins.EntityCreateMixin method*), 19
create_raw() (*nailgun.entity_mixins.EntityCreateMixin method*), 19

D

DateField (*class in nailgun.entity_fields*), 30
DateTimeField (*class in nailgun.entity_fields*), 30
DEFAULT_SERVER_CONFIG (*in module nailgun.entity_mixins*), 16
delete() (*in module nailgun.client*), 35
delete() (*nailgun.config.BaseServerConfig class method*), 33
delete() (*nailgun.entity_mixins.EntityDeleteMixin method*), 20
delete_raw() (*nailgun.entity_mixins.EntityDeleteMixin method*), 20
DictField (*class in nailgun.entity_fields*), 30

E

EmailField (*class in nailgun.entity_fields*), 30
Entity (*class in nailgun.entity_mixins*), 16
entity_with_parent() (*nailgun.entity_mixins.Entity method*), 17
EntityCreateMixin (*class in nailgun.entity_mixins*), 18
EntityCreateMixinTestCase (*class in tests.test_entity_mixins*), 40
EntityDeleteMixin (*class in nailgun.entity_mixins*), 19
EntityDeleteMixinTestCase (*class in tests.test_entity_mixins*), 40
EntityReadMixin (*class in nailgun.entity_mixins*), 20

EntityReadMixinTestCase (*class in tests.test_entity_mixins*), 40
EntitySearchMixin (*class in nailgun.entity_mixins*), 21
EntitySearchMixinTestCase (*class in tests.test_entity_mixins*), 41
EntityTestCase (*class in tests.test_entity_mixins*), 42
EntityUpdateMixin (*class in nailgun.entity_mixins*), 26
EntityUpdateMixinTestCase (*class in tests.test_entity_mixins*), 43
EntityWithCreate (*class in tests.test_entity_mixins*), 43
EntityWithDelete (*class in tests.test_entity_mixins*), 43
EntityWithRead (*class in tests.test_entity_mixins*), 43
EntityWithSearch (*class in tests.test_entity_mixins*), 43
EntityWithSearch2 (*class in tests.test_entity_mixins*), 43
EntityWithUpdate (*class in tests.test_entity_mixins*), 43

F

Field (*class in nailgun.entity_fields*), 30
FloatField (*class in nailgun.entity_fields*), 30

G

gen_value() (*nailgun.entity_fields.BooleanField method*), 30
gen_value() (*nailgun.entity_fields.DateField method*), 30
gen_value() (*nailgun.entity_fields.DateTimeField method*), 30
gen_value() (*nailgun.entity_fields.DictField method*), 30
gen_value() (*nailgun.entity_fields.EmailField method*), 30
gen_value() (*nailgun.entity_fields.FloatField method*), 30
gen_value() (*nailgun.entity_fields.IntegerField method*), 31
gen_value() (*nailgun.entity_fields.IPAddressField method*), 31
gen_value() (*nailgun.entity_fields.MACAddressField method*), 31
gen_value() (*nailgun.entity_fields.NetmaskField method*), 31
gen_value() (*nailgun.entity_fields.OneToManyField method*), 31
gen_value() (*nailgun.entity_fields.OneToOneField method*), 31
gen_value() (*nailgun.entity_fields.StringField method*), 32
gen_value() (*nailgun.entity_fields.URLField method*), 32
GenValueTestCase (*class in tests.test_entity_fields*), 38
get() (*in module nailgun.client*), 35
get() (*nailgun.config.BaseServerConfig class method*), 33
get() (*nailgun.config.ServerConfig class method*), 34
get_client_kwargs() (*nailgun.config.ServerConfig method*), 34
get_fields() (*nailgun.entity_mixins.Entity method*), 17
get_labels() (*nailgun.config.BaseServerConfig class method*), 33
get_values() (*nailgun.entity_mixins.Entity method*), 17

H

head() (*in module nailgun.client*), 35

I

IntegerField (*class in nailgun.entity_fields*), 31
IntegerFieldTestCase (*class in tests.test_entity_fields*), 39
IPAddressField (*class in nailgun.entity_fields*), 31

L

ListField (*class in nailgun.entity_fields*), 31

M

MACAddressField (*class in nailgun.entity_fields*), 31
MakeEntitiesFromIdsTestCase (*class in tests.test_entity_mixins*), 43
MakeEntityFromIdTestCase (*class in tests.test_entity_mixins*), 44
MissingValueError, 27

N

nailgun (*module*), 15
nailgun.client (*module*), 35
nailgun.config (*module*), 32
nailgun.entity_fields (*module*), 29
nailgun.entity_mixins (*module*), 15
NetmaskField (*class in nailgun.entity_fields*), 31
NoSuchFieldError, 27
NoSuchPathError, 27

O

OneToManyField (*class in nailgun.entity_fields*), 31
OneToOneField (*class in nailgun.entity_fields*), 31

P

patch() (*in module nailgun.client*), 35
path() (*nailgun.entity_mixins.Entity method*), 17
PollTaskTestCase (*class in tests.test_entity_mixins*), 44
post() (*in module nailgun.client*), 35
put() (*in module nailgun.client*), 35

R

read() (*nailgun.entity_mixins.EntityReadMixin method*), 20
read_json() (*nailgun.entity_mixins.EntityReadMixin method*), 21
read_raw() (*nailgun.entity_mixins.EntityReadMixin method*), 21
ReprTestCase (*class in tests.test_config*), 37
request() (*in module nailgun.client*), 35

S

SampleEntity (*class in tests.test_entity_mixins*), 44
SampleEntityThree (*class in tests.test_entity_mixins*), 44
SampleEntityTwo (*class in tests.test_entity_mixins*), 44
save() (*nailgun.config.BaseServerConfig method*), 33
search() (*nailgun.entity_mixins.EntitySearchMixin method*), 22
search_filter() (*nailgun.entity_mixins.EntitySearchMixin static method*), 23
search_json() (*nailgun.entity_mixins.EntitySearchMixin method*), 23
search_normalize() (*nailgun.entity_mixins.EntitySearchMixin method*), 24
search_payload() (*nailgun.entity_mixins.EntitySearchMixin method*), 24
search_raw() (*nailgun.entity_mixins.EntitySearchMixin method*), 25
ServerConfig (*class in nailgun.config*), 34
ServerConfigTestCase (*class in tests.test_config*), 38
SetContentTypeTestCase (*class in tests.test_client*), 36
setUp() (*tests.test_client.ClientTestCase method*), 35
setUp() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
setUp() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
setUp() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 40
setUp() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
setUp() (*tests.test_entity_mixins.EntityTestCase method*), 42
setUp() (*tests.test_entity_mixins.EntityUpdateMixinTestCase method*), 43
setUp() (*tests.test_entity_mixins.MakeEntitiesFromIdsTestCase method*), 44
setUp() (*tests.test_entity_mixins.MakeEntityFromIdTestCase method*), 44
setUp() (*tests.test_entity_mixins.PollTaskTestCase method*), 44
setUpClass() (*tests.test_entity_mixins.EntityReadMixinTestCase class method*), 40
StringField (*class in nailgun.entity_fields*), 31
StringFieldTestCase (*class in tests.test_entity_fields*), 39

T

TASK_POLL_RATE (*in module nailgun.entity_mixins*), 27
TASK_TIMEOUT (*in module nailgun.entity_mixins*), 27
TaskFailedError, 27
TaskTimedOutError, 27
test__poll_task_failure() (*tests.test_entity_mixins.PollTaskTestCase method*), 44
test__poll_task_success() (*tests.test_entity_mixins.PollTaskTestCase method*), 44
test__poll_task_timeout() (*tests.test_entity_mixins.PollTaskTestCase method*), 44
test_bad_value_error() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_boolean_field() (*tests.test_entity_fields.GenValueTestCase method*), 38
test_bsc_v1() (*tests.test_config.ReprTestCase method*), 37
test_bsc_v2() (*tests.test_config.ReprTestCase method*), 37
test_bsc_v3() (*tests.test_config.ReprTestCase method*), 37
test_client_request() (*tests.test_client.ClientTestCase method*), 35
test_clients() (*tests.test_client.ClientTestCase method*), 36
test_compare() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_compare_to_null() (*tests.test_entity_mixins.EntityTestCase*

method), 42
test_compare_with_filter() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_create() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
test_create_json() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
test_create_missing() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
test_create_raw_v1() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
test_create_raw_v2() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
test_create_raw_v3() (*tests.test_entity_mixins.EntityCreateMixinTestCase method*), 40
test_date_field() (*tests.test_entity_fields.GenValueTestCase method*), 38
test_datetime_field() (*tests.test_entity_fields.GenValueTestCase method*), 38
test_delete() (*tests.test_config.BaseServerConfigTestCase method*), 36
test_delete_raw() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_delete_v1() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_delete_v2() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_delete_v3() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_delete_v4() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_delete_v5() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_delete_v6() (*tests.test_entity_mixins.EntityDeleteMixinTestCase method*), 40
test_dict_field() (*tests.test_entity_fields.GenValueTestCase method*), 38
test_email_field() (*tests.test_entity_fields.GenValueTestCase method*), 38
test_entity_get_fields() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_entity_get_values() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_entity_get_values_v2() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_eq() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_eq_none() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_existing_value() (*tests.test_client.SetContentTypeTestCase method*), 36
test_false() (*tests.test_client.ContentTypeIsJsonTestCase method*), 36
test_false_with_no_headers() (*tests.test_client.ContentTypeIsJsonTestCase method*), 36
test_files_in_kwargs() (*tests.test_client.SetContentTypeTestCase method*), 36
test_float_field() (*tests.test_entity_fields.GenValueTestCase method*), 38
test_gen_netmask() (*tests.test_entity_fields.GenValueTestCase method*), 39
test_get() (*tests.test_config.BaseServerConfigTestCase method*), 37
test_get() (*tests.test_config.ServerConfigTestCase method*), 38
test_get_client_kwargs() (*tests.test_config.ServerConfigTestCase method*), 38
test_get_labels() (*tests.test_config.BaseServerConfigTestCase method*), 37
test_get_logical_args() (*tests.test_client.ClientTestCase method*), 36
test_init() (*tests.test_config.BaseServerConfigTestCase method*), 37
test_init() (*tests.test_config.ServerConfigTestCase method*), 38
test_init_invalid() (*tests.test_config.BaseServerConfigTestCase method*), 37
test_init_v1() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_init_v2() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_int_is_returned() (*tests.test_entity_fields.IntegerFieldTestCase method*), 39
test_ip_address_field()

(*tests.test_entity_fields.GenValueTestCase method*), 39
test_length_arg() (*tests.test_entity_fields.StringFieldTestCase method*), 39
test_mac_address_field() (*tests.test_entity_fields.GenValueTestCase method*), 39
test_max_val_arg() (*tests.test_entity_fields.IntegerFieldTestCase method*), 39
test_min_val_arg() (*tests.test_entity_fields.IntegerFieldTestCase method*), 39
test_min_val_max_val_args() (*tests.test_entity_fields.IntegerFieldTestCase method*), 39
test_missing_value_error() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_no_such_field_error() (*tests.test_entity_mixins.EntityTestCase method*), 42
test_no_value() (*tests.test_client.SetContentTypeTestCase method*), 36
test_one_to_many_field() (*tests.test_entity_fields.GenValueTestCase method*), 39
test_one_to_one_field() (*tests.test_entity_fields.GenValueTestCase method*), 39
test_pass_in_both() (*tests.test_entity_mixins.MakeEntitiesFromIdsTestCase method*), 44
test_pass_in_emtpy_iterable() (*tests.test_entity_mixins.MakeEntitiesFromIdsTestCase method*), 44
test_pass_in_entity_id() (*tests.test_entity_mixins.MakeEntityFromIdTestCase method*), 44
test_pass_in_entity_ids() (*tests.test_entity_mixins.MakeEntitiesFromIdsTestCase method*), 44
test_pass_in_entity_obj() (*tests.test_entity_mixins.MakeEntitiesFromIdsTestCase method*), 44
test_pass_in_entity_obj() (*tests.test_entity_mixins.MakeEntityFromIdTestCase method*), 44
test_path() (*tests.test_entity_mixins.EntityTestCase method*), 43
test_raise_config_file_error() (*tests.test_config.ServerConfigTestCase method*), 38
test_read_json() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_read_raw() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_read_v1() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_read_v2() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_read_v3() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_read_v4() (*tests.test_entity_mixins.EntityReadMixinTestCase method*), 41
test_repr_v1() (*tests.test_entity_mixins.EntityTestCase method*), 43
test_repr_v2() (*tests.test_entity_mixins.EntityTestCase method*), 43
test_repr_v3() (*tests.test_entity_mixins.EntityTestCase method*), 43
test_save() (*tests.test_config.BaseServerConfigTestCase method*), 37
test_sc_v1() (*tests.test_config.ReprTestCase method*), 37
test_sc_v2() (*tests.test_config.ReprTestCase method*), 37
test_sc_v3() (*tests.test_config.ReprTestCase method*), 37
test_sc_v4() (*tests.test_config.ReprTestCase method*), 37
test_search_filter_v1() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_filter_v2() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_filter_v3() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_filter_v4() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_json() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_normalize_v1() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_normalize_v2() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_payload_v1() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 41
test_search_payload_v2() (*tests.test_entity_mixins.EntitySearchMixinTestCase*

method), 42
test_search_payload_v3() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 42
test_search_raw() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 42
test_search_v1() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 42
test_search_v2() (*tests.test_entity_mixins.EntitySearchMixinTestCase method*), 42
test_str_is_returned() (*tests.test_entity_fields.StringFieldTestCase method*), 39
test_str_type_arg() (*tests.test_entity_fields.StringFieldTestCase method*), 39
test_true() (*tests.test_client.ContentTypeIsJsonTestCase method*), 36
test_update() (*tests.test_entity_mixins.EntityUpdateMixinTestCase method*), 43
test_update_json() (*tests.test_entity_mixins.EntityUpdateMixinTestCase method*), 43
test_update_payload_v1() (*tests.test_entity_mixins.EntityUpdateMixinTestCase method*), 43
test_update_payload_v2() (*tests.test_entity_mixins.EntityUpdateMixinTestCase method*), 43
test_update_raw() (*tests.test_entity_mixins.EntityUpdateMixinTestCase method*), 43
test_url_field() (*tests.test_entity_fields.GenValueTestCase method*), 39
TestClass (*class in tests.test_entity_fields*), 39
tests (*module*), 35
tests.test_client (*module*), 35
tests.test_config (*module*), 36
tests.test_entity_fields (*module*), 38
tests.test_entity_mixins (*module*), 40
to_json() (*nailgun.entity_mixins.Entity method*), 18
to_json_dict() (*nailgun.entity_mixins.Entity method*), 18
to_json_serializable() (*in module nailgun.entity_mixins*), 29

U

update() (*nailgun.entity_mixins.EntityUpdateMixin method*), 26
update_json() (*nailgun.entity_mixins.EntityUpdateMixin method*), 26
update_payload() (*nailgun.entity_mixins.EntityUpdateMixin method*), 26
update_raw() (*nailgun.entity_mixins.EntityUpdateMixin method*), 26
URLField (*class in nailgun.entity_fields*), 32