# Repose Documentation

***Release 1.0.0***

**Adam Charnock**

September 20, 2015

Contents

CHAPTER 1

# Getting Started

Repose allows you to declaratively define a client for restful APIs. There are three steps to getting started:

1. *Define your resources*
2. *Configure your API*
3. *Try it out*

## 1.1 1. Define your resources

Each `Resource` you define will generally map to a resource in your Api. Using GitHub's API as an example:

```
class User(Resource):
    id = fields.Integer()
    login = fields.String()
    avatar_url = fields.String()
    location = fields.String()
    site_admin = fields.Boolean()

    class Meta:
        # Endpoint for getting a single specific user
        endpoint = '/users/{login}'
        # Endpoint for listing all users
        endpoint_list = '/users'


class Repository(Resource):
    id = fields.Integer()
    name = fields.String()
    full_name = fields.String()
    description = fields.String()
    owner = fields.Embedded(User)

    class Meta:
        # Endpoint for getting a single specific repository
        endpoint = '/repos/{full_name}'
        # Endpoint for listing all repositories
        endpoint_list = '/repositories'
```

This represents a very small subset of the available GitHub API, but it serves well as a demonstration.

**See also:**

See the *Resource* class for more in-depth details regarding resource definition. Also see the *repose.fields* module for a list of available fields.

## 1.2 2. Configure your API

To configure your API you need to *instantiate* an *Api* class. You can customise the Api's behaviour either through parameters to *__init__()* or by defining your own subclass of *Api*.

In addition to providing high-level configuration, the Api instance must also be made aware of all available resources.

For example:

```
# A simple example of directly instantiating the Api class
github_api = Api(base_url='https://api.github.com')
github_api.register_resource(User)
github_api.register_resource(Repository)
```

Or, using extension:

```
# Alternatively, extend the Api class for added customisation options
class GitHubApi(Api):
    base_url = 'https://api.github.com'
    resources = [User, Repository]

github_api = GitHubApi()
```

The former is simpler, whereas the latter provides more flexibility for overriding the base *Api* class functionality.

**See also:**

See the *Api* class for more in-depth details regarding Api definitions.

## 1.3 3. Try it out

Now let's try it out and get some resources:

```
>>> # Provide the login to get a user
>>> # (as this is what we specified in Meta.endpoint)
>>> User.objects.get(login='adamcharnock')
<User(login=u'adamcharnock', site_admin=None, id=138215, avatar_url=u'https://avatars.githubuserconte

>>> # Provide the full_name to get a repository
>>> # (again, as this is what we specified in Meta.endpoint)
>>> seed_repo = Repository.objects.get(full_name='adamcharnock/seed')
>>> print seed_repo.description
A utility for easily creating and releasing Python packages

>>> # The repo's owner attribute will give us a User resource
>>> # as this is an `Embedded` field
>>> seed_repo.owner
<User(login=u'adamcharnock', site_admin=None, id=138215, avatar_url=u'https://avatars.githubuserconte
```

Ok, now let's get a list of all repositories:

```
>>> Repository.objects.count()
100 # That cannot be right...
>>> repos = Repository.objects.all()
>>> len(repos)
100
```

So we get some results, but only a hundred repositories in GitHub? That definitly sounds wrong. What is going on here then?

---

**Todo**

Implement pagination support

---

---

**Todo**

Discuss limitations of `count()`. Link into 'Using Managers' document where we'll discuss customising managers to provide this functionality.

---

CHAPTER 2

# Using Managers

CHAPTER 3

# API Reference

## 3.1 Api

**class** repose.api.**Api**(***options*)

A top-level API representation

Initialising an `Api` instance is a necessary step as doing so will furnish all registered Resources (and their Managers) with access to the API backend.

For example:

```
my_api = Api(base_url='http://example.com/api/v1')
my_api.register_resource(User)
my_api.register_resource(Comment)
my_api.register_resource(Page)
```

The same can be achieved by implementing a child class. This also gives the additional flexibility of being able to add more complex logic by overriding existing methods. For example:

```
class MyApi(Api):
    # Alternative way to provide base_url and resources
    base_url = '/api/v1'
    resources = [User, Comment, Page]

    # Additionally, customise the base URL generation
    def get_base_url(self):
        return 'http://{host}/api/{account}'.format(
            host=self.host,
            account=self.account,
        )

my_api = MyApi(host='myhost.com', account='my-account')
```

**base_url**

*str*

The fully-qualified base URL to the the API. (Eg: `"http://example.com"`)

**backend_class**

*ApiBackend*

The class to instantiate for use as the Api Backend (default: *ApiBackend*).

**resources**

*list[Resource]*

*Resource* classes to register with the API. Can also be registered using *register_resource()*.

**client_class**
The client class to instantiate. Should be either `Client` or a subclass thereof.

**__init__**(***options*)
Initialise the Api

Pass options in to customise instance variables. For example:

```
my_api = Api(base_url='http://example.com/api/v1')
```

**Parameters**

- **base_url** (*str*) – The fully-qualified base URL to the the API. (Eg: `"http://example.com"`)
- **backend_class** (*ApiBackend*) – The class to instantiate for use as the Api Backend (default: *ApiBackend*).
- **resources** (*list[Resource]*) – *Resource* classes to register with the API. Can also be registered using *register_resource()*.
- ****options** – All options specified will will become available as instance variables.

**backend_class**
alias of `ApiBackend`

**register_resource**(*resource*)
Register a resource with the Api

This will cause the resource's backend attribute to be populated.

**Parameters** **resource** (*Resource*) – The resource class to register

## 3.2 Resources

**class** `repose.resources.`**Resource**(***kwargs*)
Representation of an API resource

**parent_resource**
*list*

A list of all parent resources to this one. Often useful in generating endpoints for child resources. Parent resources are stored as `weakref.ref()`

**api**
*Api*

The API instance

**class Meta**
Override this class in child resources to provide configuration details.

The endpoints listed here can include placeholders in the form `{fieldname}`. If this resource is a child of another resource, the parent resource's fields may be accessed in the form `{parentname_fieldname}`, where `parentname` is the lowercase class name.

For example, a `User` resource may contain several `Comment` resources. In which case the `endpoint` for the `Comment` could be:

```
/user/{user_id}/comments/{id}
```

You could also expand the latter placeholder as follows:

```
/user/{user_id}/comments/{comment_id}
```

**endpoint**
*str*

Endpoint URL for a single resource (will be appended to the API's *base_url*)

**endpoint_list**
*str*

Endpoint URL for listing resources (will be appended to the API's *base_url*)

`Resource.__init__`(***kwargs*)
Initialise the resource with field values specified in `*kwargs`

**Parameters** ****kwargs** – Fields and their (decoded) values

**classmethod** `Resource.contribute_api`(*api*)
Contribute the API backend to this resource and its managers.

**Note:** Mainly for internal use

`Resource.contribute_parents`(*parent=None*)
Furnish this class with it's parent resources

**Note:** Mainly for internal use

`Resource.prepare_save`(*encoded*)
Prepare the resource to be saved

Will only return values which have changed

Can be used as a hook with which to tweak data before sending back to the server. For example:

```
def prepare_save(encoded):
    prepared = super(MyResource, self).prepare_save(encoded)
    prepared['extra_value'] = 'Something'
    return prepared
```

**Parameters** **encoded** (*dict*) – The encoded resource data

`Resource.save`()
Persist pending changes

## 3.3 Fields

**Todo**

List more fields

**class** `repose.fields.Dictionary`(**args*, ***kwargs*)
`Field` subclass with *dict* validation.

**__init__**(**args*, ***kwargs*)

**class** repose.fields.**IsoDate**(**args*, ***kwargs*)
Field subclass for ISO8601 dates.

---

**Todo**

The *IsoDate* field needs implementing Should parse ISO8601 strings into datetime objects and back again.

---

**class** repose.fields.**ManagedIdListCollection**(*model*, **args*, ***kwargs*)
Use for providing a managed collection upon a field which contains a list of model IDs.

This does a little fancy footwork to ensure that the values are only loaded when accessed. This functionality is largely provided by LazyList

**__init__**(*model*, **args*, ***kwargs*)

## 3.4 Managers

Managers have the task of managing access to resources.

---

**Note:** Managers are modelled after Django's ORM Managers.

---

For example, to access a group of fictional User resources you would use:

```
# Simple user of a manager
users = User.objects.all()
```

Here you access the objects manager on the User resource. The objects manager is known as the 'default' manager. Additional managers may also by provided. For example:

```
class User(Resource):
    ... define fields...

    # Note you need to explicitly define the 'objects' default
    # manager when you add custom managers
    objects = Manager()

    # Now add some custom managers
    active_users = Manager(filter=lambda u: u.is_active)
    inactive_users = Manager(filter=lambda u: not u.is_active)
    super_users = Manager(filter=lambda u: u.is_super_user)
```

Now you can use statements such as:

```
awesome_users = User.super_users.all()
total_active_users = User.active_users.count()
```

You can also extend the *Manager* class to provide both additional functionality and greater intelligence. For example:

```
class UserManager(Manager):

    def count(self):
        # Pull the count from the server rather than pulling all
        # users then counting them.
        json = self.api.get('/users/total_count')
        return json['total']
```

Or perhaps you want be able to perform custom actions on groups of Resources:

```
class LightManager(manager):

    def turn_on(self):
        for light in self.all():
            light.on = True
            light.save()
```

**Todo**

Implement support for pagination of resources

**class** repose.managers.**Manager**(*decoders=None*, *results_endpoint=None*, *filter=None*)

The base Manager class

**api**

*Api*

The Api instance

**decoders**

*list[Decoder]*

The decoders used to decode list data

**model**

*Resource*

The Resource class to be managed

**results**

*list*

The results as loaded from the API

**results_endpoint**

*list*

The results to be used to fetch results

**__init__**(*decoders=None*, *results_endpoint=None*, *filter=None*)

Initialise the Manager

**Parameters**

- **decoders** (*list[Decoder]*) – The decoders used to decode list data
- **results_endpoint** (*str*) – The results to be used to fetch results. Defaults to `Meta.endpoint_list`
- **filter** (*callable*) – The filter function to be applied to the results. Will be passed a single result and must return True/False if the result should be included/excluded in the results respectively.

**all**()

Return all results

**count**()

Return the total number of results

**Returns** int

**Note:** This is a naive implementation of `count()` which simply retrieves all results and counts them. You should consider overriding this (as demoed above) if dealing with non-trivial numbers of results.

**get**(***endpoint_params*)
Get a single resource

**Parameters** **endpoint_params** (*dict*) – Parameters which should be used to format the `Meta.endpoint` string.

**Returns**

**Return type** *Resource*

**get_decoders**()
Return the decoders to be used for decoding list data

**Returns** *Manager.decoders* by default

**Return type** list[Decoder]

**get_results_endpoint**()
Get the results endpoint

**Returns** `results_endpoint` as passed to *__init__()* or `Meta.endpoint_list`.

**Return type** str

## 3.5 ApiBackend

**class** repose.apibackend.**ApiBackend**(*base_url*)
Default backend implementation providing HTTP access to the remote API

This can be extended and passed into your *Api* instance at instantiation time. This can be useful if you need to customise how requests are made, or how responses are parsed.

**__init__**(*base_url*)
Instantiate this class

**Parameters** **base_url** (*str*) – The fully-qualified base URL to the the API. (Eg: `"http://example.com"`).

**delete**(*endpoint*, *json*)
Perform a HTTP DELETE request for the specified endpoint

**Parameters** **json** (*dict*) – The JSON body to post with the request

**Returns** Typically a python list, dictionary, or None

**Return type** object

**get**(*endpoint*, *params=None*)
Perform a HTTP GET request for the specified endpoint

**Parameters** **params** (*dict*) – Dictionary of URL params

**Returns** Typically a python list or dictionary

**Return type** object

**make_url**(*endpoint*)
Construct the fully qualified URL for the given endpoint.

For example:

```
>>> my_backend = ApiBackend(base_url="http://example.com/api")
>>> my_backend.make_url("/user/1")
"http://example.com/api/user/1"
```

**Parameters** **endpoint** (*str*) – The API endpoint (Eg: `"/user/1"`).

**Returns** The fully qualified URL

**Return type** str

**parse_response**(*response*)
Parse a response into a Python structure

**Parameters** **response** (`requests.Response`) – A Response object, unless otherwise provided by the *get()*

**Returns** Typically a python list or dictionary

**Return type** object

**post**(*endpoint*, *json*)
Perform a HTTP POST request for the specified endpoint

**Parameters** **json** (*dict*) – The JSON body to post with the request

**Returns** Typically a python list, dictionary, or None

**Return type** object

**put**(*endpoint*, *json*)
Perform a HTTP PUT request for the specified endpoint

**Parameters** **json** (*dict*) – The JSON body to post with the request

**Returns** Typically a python list, dictionary, or None

**Return type** object

## 3.6 Decoders

Decoders are used be fields to decode incoming data from the API into a form usable in Python.

Those listed here are typically used by the *fields* module. Unless you are creating your own field, you can probably focus your attention there.

This is the inverse operation to that of *encoders*.

**class** repose.decoders.**IdToLazyModelListDecoder**(*resource*)
Decode a list of resource IDs into a lazily loaded list of *Resource* objects

**__init__**(*resource*)
Initialise the decoder

**Parameters** **resource** (Resource) – The Resource class (*not an instance*) to which the IDs listed relate.

**decode**(*value*)
Decode the value into a `LazyList`.

**Note:** This assumes the destination *Resource* has an ID field and that the endpoint is in the form `/myresource/{myresource_id}`

**Todo**

Consider refactoring out these assumptions

## 3.7 Encoders

Decoders are used be fields to encode Python values into a form consumable by the API.

Those listed here are typically used by the *fields* module. Unless you are creating your own field, you can probably focus your attention there.

This is the inverse operation to that of *decoders*.

**class** `repose.encoders.`**ModelToIdListEncoder**
Encode a list of *Resource* instances into a list of resource IDs.

**encode**(*value*)
Initialise the encoder

**Parameters** **value** (*list[Resource]*) – A list of *Resource* instances to be encoded

## 3.8 Utilities

General utilities used within Repose.

*For the most part these can be ignored, their usage is mainly for internal purposes.*

**class** `repose.utilities.`**LazyList**(*generator=None*, *size=0*)
Wraps a generator from which data is only loaded when needed.

**Todo**

The *LazyList* loading logic could be more intelligent

**Todo**

Make the size parameter optional

**__init__**(*generator=None*, *size=0*)
Initialise the LazyList

**Parameters**

- **generator** (*generator*) – The generator to be lazy loaded
- **size** (*int*) – The size of the list to be loaded

`repose.utilities.`**get_values_from_endpoint**(*resource*, *endpoint_params*)
Determine if any values in the endpoint parameters should be used to populate fields.

An example of this would be resources which don't provide their own ID in the return data, and it must therefore come from the endpoint used to access the resource. In which case, you may define the resource's ID field as:

```
id = fields.Integer(from_endpoint='id')
```

**Parameters**

- **resource** (*repose.resources.Resource*) – The class of the resource being populated
- **endpoint_params** (*dict*) – All parameters available for formatting to the endpoint strings.

repose.utilities.**make_endpoint**(*model*)

Make an endpoint for a given model

See the *repose.resources.Resource.Meta* for a description of endpoint URL formatting.

CHAPTER 4

# Todo List

**Todo**

Consider refactoring out these assumptions

(The original entry is located in docstring of repose.decoders.IdToLazyModelListDecoder.decode, line 7.)

**Todo**

List more fields

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/repose/checkouts/latest/docs/api/fields.rst, line 4.)

**Todo**

The `IsoDate` field needs implementing Should parse ISO8601 strings into datetime objects and back again.

(The original entry is located in docstring of repose.fields.IsoDate, line 3.)

**Todo**

Implement support for pagination of resources

(The original entry is located in docstring of repose.managers, line 55.)

**Todo**

The `LazyList` loading logic could be more intelligent

(The original entry is located in docstring of repose.utilities.LazyList, line 3.)

**Todo**

Make the size parameter optional

(The original entry is located in docstring of repose.utilities.LazyList, line 5.)

**Todo**

Implement pagination support

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/repose/checkouts/latest/docs/getting_started.rst, line 148.)

**Todo**

Discuss limitations of `count()`. Link into 'Using Managers' document where we'll discuss customising managers to provide this functionality.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/repose/checkouts/latest/docs/getting_started.rst, line 152.)

Tested on Python 2.7, 3.2, 3.3, 3.4, 3.5

CHAPTER 5

# Installation

Installation using pip:

```
pip install repose
```

CHAPTER 6

# Credits

Developed by Adam Charnock, contributions very welcome!

repose is packaged using seed.

## r

## Symbols



## A



## B



## C



## D



## E



## G



## I



## L



## M



## P



## R

## S