
Flask-Potion Documentation

Release 0.0.0

Lars Schöning

Sep 02, 2017

Contents

1	User's guide	3
1.1	Installation	3
1.2	Quick Start Guide	3
1.3	Field types	17
1.4	Filters	21
1.5	Api class	24
1.6	Routes & Route Sets	25
1.7	Resources	27
1.8	Managers	30
1.9	Signals	32
1.10	Permissions with <i>Flask-Principal</i>	34
1.11	Advanced Recipes	40
	Python Module Index	43



Flask-Potion is a powerful Flask extension for building RESTful JSON APIs. Potion features include validation, model resources and routes, relations, object permissions, filtering, sorting, pagination, signals, and automatic API schema generation.

Potion ships with backends for SQLAlchemy, peewee and MongoEngine models. It is possible to add backends for other data stores, or even to use a subset of Potion without any data store at all.

Installation

Install Flask-Potion using `pip`:

```
pip install flask-potion
```

Flask-Potion requires Python version 2.7 or 3.3+. It works best with Python 3.x.

If you are using SQLAlchemy for your backend, you should also install Flask-SQLAlchemy. For any backend you use, you'll need to also install their respective packages.

Quick Start Guide

This introductory guide describes how to set up an API using SQLAlchemy with Flask-Potion, query it, and attach routes to resources.

A minimal Flask-Potion API looks like this:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_potion import Api, ModelResource

app = Flask(__name__)
db = SQLAlchemy(app)

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(), nullable=False)
    year_published = db.Column(db.Integer)

db.create_all()
```

```
class BookResource(ModelResource):
    class Meta:
        model = Book

api = Api(app)
api.add_resource(BookResource)

if __name__ == '__main__':
    app.run()
```

Save this as `server.py` and run it using your Python interpreter. The application will create an in-memory SQLite database, so the state of the application will reset every time the server is restarted.

```
$ python server.py
* Running on http://127.0.0.1:5000/
```

We're going to use the excellent [HTTPie](#) command line client (seen here as `http`) to query the API. Let's first see if there are any *book* items on our server:

```
http :5000/book
```

```
HTTP/1.0 200 OK
Content-Length: 2
Content-Type: application/json
Date: Sat, 07 Feb 2015 10:25:26 GMT
Link: </book?page=1&per_page=20>; rel="self", </book?page=1&per_page=20>; rel="last"
Server: Werkzeug/0.9.6 Python/3.3.2
X-Total-Count: 0

[]
```

We can see that there are no *book* items. As the `Link` and `X-Total-Count` headers show us, the resource is paginated to 20 items per page (more on that under [Pagination](#)). We're now going to create a book:

```
$ http -v :5000/book title="On the Origin of Species" year_published:=1859
```

```
POST /book HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Content-Length: 61
Content-Type: application/json; charset=utf-8
Host: 127.0.0.1:5000
User-Agent: HTTPie/0.7.2

{
  "title": "On the Origin of Species",
  "year_published": 1859
}
```

```
HTTP/1.0 200 OK
Content-Length: 80
Content-Type: application/json
Date: Sat, 07 Feb 2015 11:12:33 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
```



```

"$uri": "/book/1",
"title": "On the Origin of Species",
"year_published": 1859
}

```

What did we do here? We used a `ModelResource` and defined a `model` in its `Meta` property. `Meta` and `Schema` are the two of the primary ways to describe resources (a third being `Route`, which we'll go into later).

Meta class attributes

The `Meta` class is how the basic functions of a resource are defined. Besides `model`, there are a few other properties that control how the `ModelResource` maps to the SQLAlchemy model:

Attribute name	Default	Description
<code>model</code>	—	The <i>Flask-SQLAlchemy</i> model
<code>name</code>	—	Name of the resource; defaults to the lower-case of the <i>model's</i> table name
<code>id_attribute</code>	'id'	With SQLAlchemy models, defaults to the name of the primary key of <i>model</i> .
<code>id_converter</code>	—	Flask URL converter for resource routes. Typically this is inferred from <i>id_field_class</i> .
<code>id_field_class</code>	<code>fields.Integer</code>	Field class to use for " <code>\$id</code> ", also used to determine the URL route converter for resource routes.
<code>include_id</code>	False	Whether to include the id of the item as an " <code>\$id</code> " attribute. The default is a " <code>\$uri</code> " attribute with the URI of the item.
<code>include_type</code>	False	Whether to include a " <code>\$type</code> " attribute with the type of the resource
<code>include_fields</code>	—	A list of fields that should be imported from the <i>model</i> . By default, all columns other than foreign key and primary key columns are imported. <code>sqlalchemy.orm.relationship()</code> model attributes and hybrid properties cannot be defined in this way and have to be specified explicitly in <code>Schema</code> .
<code>exclude_fields</code>	—	A list of fields that should not be imported from the <i>model</i> .
<code>required_fields</code>	—	Fields that are automatically imported from the model are automatically required if their columns are not <i>nullable</i> and do not have a <i>default</i> .
<code>read_only_fields</code>	—	A list of fields that are returned by the resource but are ignored in <i>POST</i> and <i>PATCH</i> requests. Useful for e.g. timestamps.
<code>filters</code>	True	Used to configure what properties of an item can be filtered and what filters can be used.
<code>write_only_fields</code>	—	A list of fields that can be written to but are not returned. For secret stuff.
<code>title</code>	—	JSON-schema title declaration
<code>description</code>	—	JSON-schema description declaration
<code>manager</code>	<code>SQLAlchemyManager</code>	A manager class that takes care of reading from and writing to the data store
<code>key_converters</code>	<code>RefKey()</code> , <code>IDKey()</code>	A list of <code>natural_keys.Key</code> instances. The first is used for formatting fields. <code>ToOne</code> references.
<code>natural_key</code>	None	A string, or tuple of strings, corresponding to schema field names, for a natural key.
<code>exclude_routes</code>	—	A list of rel-strings for any previously defined routes that should not be published for this resource.

Schema class attributes

Schema is used to define a default schema for a resource. The `Schema` class contains a set of fields that inherit from `fields.Raw`

Using `ModelResource` with a SQLAlchemy model, the schema is for the most part auto-generated for us. Yet it still on occasion makes sense to manually describe a field. The reference field types, `fields.ToOne` and `fields.ToMany`, also need to be set by hand.

For instance, our `book` resource only stores books produced by the printing press. Let's acknowledge this by setting a sensible minimum for `year_published`:

```
from flask_potion import fields

class BookResource(ModelResource):
    class Meta:
        model = Book

    class Schema:
        year_published = fields.Integer(minimum=1400)
```

This also serves as our introduction to error messages:

```
$ http :5000/book title="Jikji" year_published:=1377
```

```
HTTP/1.0 400 BAD REQUEST
Content-Length: 187
Content-Type: application/json
Date: Sat, 07 Feb 2015 11:52:05 GMT
Server: Werkzeug/0.9.6 Python/3.3.2
```

```
{
  "errors": [
    {
      "path": [
        "year_published"
      ],
      "validationOf": {
        "minimum": 1400
      }
    }
  ],
  "message": "Bad Request",
  "status": 400
}
```

Oops.

Relationships

RESTful relationships create a variety of API client design and caching problems that Potion has been written to address. To preface what you will see now, it needs to be said that Potion should be used with SPDY or the upcoming HTTP/2 as it generates more requests than some alternative approaches.

We now have both an *author* and a *book* resource:

```

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import backref
from flask_potion.routes import Relation
from flask_potion import ModelResource, fields, Api

app = Flask(__name__)
db = SQLAlchemy(app)

class Author(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(), nullable=False)
    last_name = db.Column(db.String(), nullable=False)

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey(Author.id), nullable=False)

    title = db.Column(db.String(), nullable=False)
    year_published = db.Column(db.Integer)

    author = db.relationship(Author, backref=backref('books', lazy='dynamic'))

db.create_all()

class BookResource(ModelResource):
    class Meta:
        model = Book

    class Schema:
        author = fields.ToOne('author')

class AuthorResource(ModelResource):
    books = Relation('book')

    class Meta:
        model = Author

api = Api(app)
api.add_resource(BookResource)
api.add_resource(AuthorResource)

if __name__ == '__main__':
    app.run()

```

We're going to add two authors and books:

```
http :5000/author first_name=Charles last_name=Darwin
```

```

HTTP/1.0 200 OK
Content-Length: 69
Content-Type: application/json
Date: Sat, 07 Feb 2015 12:11:33 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
  "$uri": "/author/1",

```

```
"first_name": "Charles",
"last_name": "Darwin"
}
```

Note: At the moment, references always need to be declared as json-ref objects. This is tedious during command-line use, and an enhancement to Potion to support using ids and natural keys in requests is already in the works.

```
http :5000/book title="On the Origin of Species" author:=1 year_published:=1859
```

```
HTTP/1.0 200 OK
Content-Length: 113
Content-Type: application/json
Date: Sat, 07 Feb 2015 12:16:11 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
  "$uri": "/book/1",
  "author": {
    "$ref": "/author/1"
  },
  "title": "On the Origin of Species",
  "year_published": 1859
}
```

```
http :5000/author first_name=James last_name=Watson > /dev/null
http :5000/book title="The Double Helix" author:=2 year_published:=1968 > /dev/null
```

As you can see, references in Potion are [JSON Reference](#) draft reference objects. These objects always have the same format — `{"$ref": "target-uri"}` — and can easily be recognized by an API client when deserializing JSON. An API client can first check its cache for the target item and, if necessary, query it from the server.

Requests allow both plain ids and *json-ref* objects — it's all the same to the server.

There are now two ways available to us for querying the relationship between the resources. The first is the author's `Relation('book')`, which created a new route on the *author* resource with references to the book resource. Let's query Charles' books:

```
http :5000/author/1/books
```

```
HTTP/1.0 200 OK
Content-Length: 21
Content-Type: application/json
Date: Sat, 07 Feb 2015 12:18:45 GMT
Link: </author/1/books?page=1&per_page=20>; rel="self", </author/1/books?page=1&per_
    ↪page=20>; rel="last"
Server: Werkzeug/0.9.6 Python/3.3.2
X-Total-Count: 1

[
  {
    "$ref": "/book/1"
  }
]
```

This is not a particularly good example for using `Relation`, and in fact there are few at all. There is a more RESTful way for querying a *one-to-many* relation:

```
http GET :5000/book where=='{"author": {"$ref": "/author/1"}}'
```

```
HTTP/1.0 200 OK
Content-Length: 115
Content-Type: application/json
Date: Sat, 07 Feb 2015 12:34:18 GMT
Link: </book?page=1&per_page=20>; rel="self",</book?page=1&per_page=20>; rel="last"
Server: Werkzeug/0.9.6 Python/3.3.2
X-Total-Count: 1

[
  {
    "$uri": "/book/1",
    "author": {
      "$ref": "/author/1"
    },
    "title": "On the Origin of Species",
    "year_published": 1859
  }
]
```

So far, in our queries, we have used item ids and *json-ref* objects to refer to items. These *surrogate keys* can be difficult to remember and tedious to work with on the command line — but Potion has a solution:

Natural Keys

A *natural key* is a unique identifier that exists in the real world and is often more memorable than a surrogate key. Potion ships with support for declaring natural keys.

The *author* model has both a first name and a last name. Together, these two names form a natural key for the *author* resource. We'll update both our database model and our resource to reflect this:

```
class Author(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(), nullable=False)
    last_name = db.Column(db.String(), nullable=False)

    __table_args__ = (
        UniqueConstraint('first_name', 'last_name'), # unique constraint added here
    )
```

```
class AuthorResource(ModelResource):
    class Meta:
        model = Author
        natural_key = ('first_name', 'last_name') # natural key declaration added_
↪here
```

Now our earlier query can be written using the full name of the author:

```
http GET :5000/book where=='{"author": ["Charles", "Darwin"]}'
```

Natural keys can be declared as either a single unique field or a tuple of fields that are unique together.

Filtering & Sorting

Instances of a *ModelResource* can be filtered using the *where* query and sorted using *sort*.

We were interested in relations, so we filtered a *fields.ToOne* field for equality. Most other field types can also be filtered and support custom comparators. Here are some examples of *where* queries:

```
http :5000/book where=='{"year_published": {"$gt": 1900}}' # Book.year_
↪published > 1900
http :5000/author where=='{"first_name": {"$startswith": "C"}}' # Author.
↪first_name starts with 'C'
http :5000/author where=='{"first_name": {"$in": ["Charles", "James"]}}' # Author.
↪first_name in ['Charles', 'James']
http :5000/book where=='{"title": "The Double Helix", "year_published": {"$lt": 2000}}'
↪'
```

Here are some examples of *sort* queries:

```
http :5000/book sort=='{"year_published": false}' # Book.year_
↪published ascending
http :5000/book sort=='{"year_published": false, "title": true}' # Book.year_
↪published ascending, Book.title descending
```

Both *where* and *sort* need to be valid JSON, so use double quotes.

See *Filters* for a full list of possible filters.

Pagination

Potion pagination is borrowed from the *GitHub API*. Pages are requested using the *page* and *per_page* query string arguments. The *Link* header lists links to the current, first, previous, next, and last page. In addition, the *X-Total-Count* header contains a count of the total number of items.

```
HTTP/1.0 200 OK
Content-Type: application/json
Link: </book?page=1&per_page=20>; rel="self",
      </book?page=3&per_page=20>; rel="last",
      </book?page=2&per_page=20>; rel="next"
X-Total-Count: 55
```

ModelResource items are paginated automatically.

The default and maximum number of items per page can be configured using the '*POTION_DEFAULT_PER_PAGE*' and '*POTION_MAX_PER_PAGE*' configuration variables.

Routes

Routes are added using decorators named after the HTTP methods, declared either with or without arguments. The format for the route decorators is:

```
Route.METHOD(rule = None,
               rel=None,
               attribute=None,
               schema=None,
               response_schema=None)
```

A Route instance itself also has decorators for each method, so that they can define different functions for different HTTP methods on the same endpoint.

Each method has its own `schema` and `response_schema` used to decode, verify, and encode requests and responses. If `schema` is a `schema.FieldSet`, its properties are spread over the route function as keyword arguments.

`ItemRoute` is a special route, used with `ModelResource`, whose rule is prefixed `'/<id_converter:id>'` and that passes the item as the first function argument.

Here is a slightly different Book model (a rating has been added) and a `book` resource with some of the different kinds of routes:

```
class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(), nullable=False)
    year_published = db.Column(db.Integer)
    rating = db.Column(db.Integer, default=5)

class BookResource(ModelResource):
    class Meta:
        model = Book
        excluded_fields = ['rating']

    @ItemRoute.GET('/rating')
    def rating(self, book) -> fields.Integer():
        return book.rating

    @rating.POST
    def rate(self, book, value: fields.Integer(minimum=1, maximum=10)) -> fields.
    ↪Integer():
        self.manager.update(book, {"rating": value})
        return value

    @ItemRoute.GET
    def is_recent(self, book) -> fields.Boolean():
        return datetime.date.today().year <= book.year_published + 10

    @Route.GET
    def genres(self) -> fields.List(fields.String, description="A list of genres"):
        return ['biography', 'history', 'essay', 'law', 'philosophy']
```

Note: This example makes use of [function annotations](#), which appeared in Python 3.0. If you are developing for Python 2.x, you will have to set these properties manually using the `schema` and `response_schema` decorator arguments:

```
@Route.POST('/rating',
            schema=FieldSet({"value": fields.Integer(minimum=1, maximum=5)}),
            response_schema=fields.Integer())
def rate(self, book, value):
    self.manager.update(book, {"rating": value})
    return value
```

After adding a book, we can give these routes a spin:

```
http GET :5000/book/1/rating
```

```
HTTP/1.0 200 OK
Content-Length: 3
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:16:37 GMT
Server: Werkzeug/0.9.6 Python/3.3.2
```

```
5
```

```
http POST :5000/book/1/rating value=7
```

```
HTTP/1.0 200 OK
Content-Length: 1
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:17:59 GMT
Server: Werkzeug/0.9.6 Python/3.3.2
```

```
7
```

```
http GET :5000/book/1/is-recent
```

```
HTTP/1.0 200 OK
Content-Length: 5
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:20:19 GMT
Server: Werkzeug/0.9.6 Python/3.3.2
```

```
false
```

```
http GET :5000/book/genres
```

```
HTTP/1.0 200 OK
Content-Length: 54
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:20:44 GMT
Server: Werkzeug/0.9.6 Python/3.3.2
```

```
[
  "biography",
  "history",
  "essay",
  "law",
  "philosophy"
]
```

It is worth noting that *ModelResource* is not much more than the empty *Resource* type with a few custom routes. *Route* and *Resource* are the backbone of Potion.

Route Sets & Mixins

In the example above, we have one property — rating — which can be read and updated by accessing a specific route. Potion provides a shortcut for this common pattern. Let's use *ItemAttributeRoute* to rewrite the rating getter and setter:


```
class BookResource(ModelResource):
    rating = ItemAttributeRoute(fields.Number)

    # ...
```

Done. Now, this isn't strictly a *set* of routes — but it implements `RouteSet`, which can be used to write reusable groups of routes. (`Relation` is also a route set).

Two additional built-in route-sets are planned: `ItemMapAttribute` and `ItemSetAttribute` for dictionary and collection item properties.

A second pattern for reusability is the *mix-in*. They can augment the `Schema` and `Meta` attributes and add new routes and route sets to the resources. Here is an example mix-in, adding two new fields to the schema:

```
class MetaMixin(object):
    class Schema:
        created_at = fields.DateTime(io='r')
        updated_at = fields.DateTime(io='r', nullable=True)
```

```
class BookResource(MetaMixin, ModelResource):
    # ...
```

Mixin and Resource base classes are evaluated left-to-right.

Self-documenting API

It can be a huge hassle to write and maintain the documentation of an API—not with Potion! In fact, every API you saw in this quick start guide was fully documented.

Potion documents itself using [JSON Hyper-Schema](#). A `/schema` route at the route of the API enumerates all API resources and the location of their schemas. The schema of a `ModelResource` can get quite overwhelming, so to begin with we'll look at a very simple resource:

```
from flask import Flask
from flask_potion import Api, Resource, fields

app = Flask(__name__)

api = Api(app)
api.add_resource(Resource)

if __name__ == '__main__':
    app.run()
```

```
http :5000/schema
```

```
HTTP/1.0 200 OK
Content-Length: 138
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:32:21 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  "definitions": {},
  "properties": {
```

```

    "resource": {
      "$ref": "/resource/schema#"
    }
  }
}

```

```
http :5000/resource/schema
```

```

HTTP/1.0 200 OK
Content-Length: 140
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:32:37 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  "links": [
    {
      "href": "/resource/schema",
      "method": "GET",
      "rel": "describedBy"
    }
  ]
}

```

As we can see from the schema above, *Resource* has a very simple schema with a single link – its schema! This example is perhaps *too* simple, so we’re going to complete the guide with a slightly more complicated resource and schema:

```

class SimpleResource(Resource):
    class Meta:
        name = 'simple'

    class Schema:
        name = fields.String()
        value = fields.Number()

    @Route.POST
    def create(self, value: fields.Number()) -> fields.Inline('self'):
        return {"name": "foo", "value": value}

api.add_resource(SimpleResource)

```

```
http :5000/simple/schema
```

```

HTTP/1.0 200 OK
Content-Length: 429
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:38:01 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  "links": [
    {
      "href": "/simple/create",

```

```

    "method": "POST",
    "rel": "create",
    "schema": {
        "additionalProperties": false,
        "properties": {
            "value": {
                "type": "number"
            }
        },
        "type": "object"
    },
    "targetSchema": {
        "$ref": "#"
    }
},
{
    "href": "/simple/schema",
    "method": "GET",
    "rel": "describedBy"
}
],
"properties": {
    "name": {
        "type": "string"
    },
    "value": {
        "type": "number"
    }
},
"type": "object"
}

```

```
http :5000/simple/create value:=1.23
```

```

HTTP/1.0 200 OK
Content-Length: 30
Content-Type: application/json
Date: Sat, 07 Feb 2015 16:40:59 GMT
Server: Werkzeug/0.9.6 Python/3.3.2

{
    "name": "foo",
    "value": 1.23
}

```

In production APIs developed using Potion, resource routes are often decorated using an authentication decorator provided when initializing [Api](#).

To make the documentation of an API protected by decorators available to unauthenticated users, you may skip the decoration of schema routes by setting the 'POTION_DECORATE_SCHEMA_ENDPOINTS' configuration variable to False.

Peewee backend

Potion also includes a Peewee backend if you want a more lightweight ORM. The Peewee backend is very similar to the SQLAlchemy one and only requires a few minor changes to the example above.

First, you'll need to install peewee:

```
$ pip install peewee
```

Second, when instantiating your Potion API you'll want to set the default manager to the PeeweeManager:

```
from flask_potion.contrib.peewee import PeeweeManager

# ...

api = Api(app, default_manager=PeeweeManager)
```

After that you can pass your Peewee models to your ModelResources just like you would with SQLAlchemy:

```
from peewee import Model, CharField, IntegerField

class Book(Model):
    title = CharField(null=False)
    year_published = IntegerField()

    class Meta:
        database = db

class BookResource(ModelResource):
    class Meta:
        model = Book
```

See the examples directory for a fully functioning example using Peewee.

MongoEngine backend

If you are more of a NoSQL person, you can use [MongoEngine](#) — an ORM for MongoDB.

First ensure you have installed the `flask_mongoengine` package:

```
$ pip install flask_mongoengine
```

The MongoEngine manager works just like the other managers. See the examples directory for an example using MongoEngine.

Next steps...

This guide has only skimmed the surface of what Potion can do for you.

In particular you may be interested in [Permissions with Flask-Principal](#), a guide to a fully-fledged permissions system for SQLAlchemy using Flask-Principal.

Potion API clients

Do you need a client for Potion? Look no further:

- [potion-client](#) is a client written in Python that auto-generates Resources using APIs' JSON schema endpoints. IPython/Jupyter Notebook support helps explore APIs.
- [potion-node](#) is a client written in TypeScript for Node with integrations for AngularJS 1.x and AngularJS 2+.

Field types

Raw field class

class `fields.Raw`(*schema*, *io*='rw', *default*=None, *attribute*=None, *nullable*=False, *title*=None, *description*=None)

This is the base class for all field types, can be given any JSON-schema.

```
>>> f = fields.Raw({"type": "string"}, io="r")
>>> f.response
{'readOnly': True, 'type': 'string'}
```

Parameters

- **io** – one or more of “r” (read), “c” (create), “u” (update) and “w” (write), default: “rw”; used to control presence in fieldsets/parent schemas
- **schema** – JSON-schema for field, or callable resolving to a JSON-schema when called
- **default** – optional default value, must be JSON-convertible; may be a callable with no arguments
- **attribute** – key on parent object, optional.
- **nullable** – whether the field is nullable.
- **title** – optional title for JSON schema
- **description** – optional description for JSON schema

schema ()

JSON schema representation

format (*value*)

Format a Python value representation for output in JSON. Noop by default.

convert (*instance*, *update*=False, *validate*=True)

Convert a JSON value representation to a Python object. Noop by default.

Reference field types

class `fields.ToOne`(*resource*, ***kwargs*)

Represents references between resources as *json-ref* objects.

Resource references can be one of the following:

- Resource class
- a string with a resource name
- a string with a module name and class name of a resource
- "self" — which resolves to the resource this field is bound to

Parameters **resource** – a resource reference

class `fields.ToMany`(*resource*, ***kwargs*)

Like *ToOne*, but for arrays of references.

Basic field types

class `fields.Any` (***kwargs*)
 A field type that allows any value.

class `fields.String` (*min_length=None, max_length=None, pattern=None, enum=None, format=None, **kwargs*)

Parameters

- **min_length** (*int*) – minimum length of string
- **max_length** (*int*) – maximum length of string
- **pattern** (*str*) – regex pattern that the string must match
- **format** (*str*) – a JSON Schema format string to validate against

Warning: The validation of format-strings is handled by `jsonschema` and may require additional package dependencies.

- **enum** (*list*) – list of strings with enumeration

class `fields.Integer` (*minimum=None, maximum=None, default=None, **kwargs*)

class `fields.PositiveInteger` (*maximum=None, **kwargs*)
 A *Integer* field that only accepts integers ≥ 1 .

class `fields.Number` (*minimum=None, maximum=None, exclusive_minimum=False, exclusive_maximum=False, **kwargs*)

class `fields.Boolean` (***kwargs*)

class `fields.Date` (***kwargs*)
 A field for EJSON-style dates in the format:

```
{"$date": MILLISECONDS_SINCE_EPOCH}
```

Converts to `datetime.date` with UTC timezone.

class `fields.DateTime` (***kwargs*)
 A field for EJSON-style date-times in the format:

```
{"$date": MILLISECONDS_SINCE_EPOCH}
```

Converts to `datetime.datetime` with UTC timezone.

class `fields.DateString` (***kwargs*)
 A field for ISO8601-formatted date strings.

class `fields.DateTimeString` (***kwargs*)
 A field for ISO8601-formatted date-time strings.

class `fields.Uri` (***kwargs*)

class `fields.UUID` (***kwargs*)
 A field for UUID strings in canonical form.

class `fields.Custom` (*schema, converter=None, formatter=None, **kwargs*)
 A field type that can be passed any schema and optional formatter/converter transformers. It is a very thin wrapper over *Raw*.

Parameters

- **schema** (*dict*) – JSON-schema
- **converter** (*callable*) – convert function
- **formatter** (*callable*) – format function

Composite field types

class `fields.Array` (*cls_or_instance*, *min_items=None*, *max_items=None*, *unique=None*, ***kwargs*)
A field for an array of a given field type.

Parameters

- **cls_or_instance** (*Raw*) – field class or instance
- **min_items** (*int*) – minimum number of items
- **max_items** (*int*) – maximum number of items
- **unique** (*bool*) – if True, all values in the list must be unique

class `fields.Object` (*properties=None*, *pattern=None*, *pattern_properties=None*, *additional_properties=None*, ***kwargs*)
A versatile field for an object, containing either properties all of a single type, properties matching a pattern, or named properties matching some fields.

Raw.attribute is not used in pattern properties and additional properties.

Parameters

- **properties** – field class, instance, or dictionary of {property: field} pairs
- **pattern** (*str*) – an optional regular expression that all property keys must match
- **pattern_properties** (*dict*) – dictionary of {property: field} pairs
- **additional_properties** (*Raw*) – field class or instance

class `fields.AttributeMapped` (*cls_or_instance*, *mapping_attribute=None*, ***kwargs*)
Maps property keys from a JSON object to a list of items using *mapping_attribute*. The mapping attribute is the name of the attribute where the value of the property key is set on the property values.
`contrib.alchemy.fields.InlineModel` is typically used with this field in a common SQLAlchemy pattern.

Parameters

- **cls_or_instance** (*Raw*) – field class or instance
- **pattern** (*str*) – an optional regular expression that all property keys must match
- **mapping_attribute** (*str*) – mapping attribute

SQLAlchemy-specific field types

class `contrib.alchemy.fields.InlineModel`
Changed in version 0.11: Renamed from `fields.sa.InlineModel` to `contrib.alchemy.fields.InlineModel`.

For creating SQLAlchemy models without having to give them their own resource.

Usage example:

```
class FooResource(Resource):
    class Meta:
        model = Foo

    class Schema:
        # Here, Foo.bars is a collection of Bar items
        bars = fields.List(fields.InlineModel({
            "name": fields.String(description="Bar name"),
            "height": fields.Integer(description="Height of bar")
        }, model=Bar))
```

Parameters

- **properties** (*dict*) – A dictionary of *Raw* objects
- **model** – An SQLAlchemy model

Internal types

Field types

class `fields.Inline` (*resource*, *patchable=False*, ***kwargs*)

Formats and converts items in a `ModelResource` using the resource's schema.

Parameters

- **resource** – a resource reference as in *ToOne*
- **patchable** (*bool*) – whether to allow partial objects

class `fields.ItemType` (*resource*)

A string field that formats the name of a resource; read-only.

class `fields.ItemUri` (*resource*, *attribute=None*)

A string field that formats the url of a resource item; read-only.

Schema types

class `schema.Schema`

The base class for all types with a schema in Potion. Has *response* and a *request* attributes for the schema to be used, respectively, for serializing and de-serializing.

Any class inheriting from schema needs to implement *schema()*.

response

JSON-schema used to represent data returned by the server.

request

JSON-schema used for validation of data sent to the server.

schema()

Abstract method returning the JSON schema used by both *response* and *request*.

Returns a JSON-schema or a tuple of JSON-schemas in the formats (*response_schema*, *request_schema*) or (*read_schema*, *create_schema*, *update_schema*)

format (value)

Formats a python object for JSON serialization. Noop by default.

Parameters *value* (*object*) –

Returns

convert (*instance*, *update=False*)

Validates a deserialized JSON object against *request* and converts it into a python object.

Parameters *instance* – JSON import

Raises **PotionValidationError** – if validation failed

parse_request (*request*)

Parses a Flask request object, validates it the against *request* and returns the converted request data.

Parameters *request* – Flask request object

Returns

format_response (*response*)

Takes a response value, which can be a data object or a tuple (data, code) or (data, code, headers) and formats it using *format()*.

Parameters *response* – A response tuple.

Returns A tuple in the form (data, code, headers)

class `schema.FieldSet` (*fields*, *required_fields=None*)

A schema representation of a dictionary of *fields.Raw* objects.

Uses the fields' *io* attributes to determine whether they are read-only, write-only, or read-write.

Parameters

- **fields** (*dict*) – a dictionary of *fields.Raw* objects
- **required_fields** – a list or tuple of field names that are required during parsing

convert (*instance*, *update=False*, *pre_resolved_properties=None*, *patchable=False*, *strict=False*)

Parameters

- **instance** – JSON-object
- **pre_resolved_properties** – optional dictionary of properties that are already known
- **patchable** (*bool*) – when True does not check for required fields
- **strict** (*bool*) –

Returns

Filters

Filter expressions

Changed in version 0.11: `Meta.allowed_filters` has been renamed to `Meta.filters` and the format for filter expressions has changed.

`Meta.filters` may contain an expression used to specify which properties of items belonging to a resource can be filtered, and how.

The *filters* expression can be a `bool` or a `dict` keyed by field names. The values of the `dict` can be either a `bool` or a list of filter names. The '*' attribute is a wildcard for any remaining field names.

For example, the following allows all filters:

```
filters = True
```

The following allows filtering on the "name" field:

```
filters = {  
    "name": True  
}
```

The following allows filtering by equals and not equals on the "name" field:

```
filters = {  
    "name": ['eq', 'ne']  
}
```

In addition it is also possible to specify custom filters this way:

```
filters = {  
    "name": {  
        "text": MyTextFilter  
    },  
    "*": True  
}
```

Built-in default filters

Filters are implemented for each contributed backend individually. The following filter classes are implemented for most or all backends:

Name	Filter class	Description	Used with
—	<code>filters.EqualFilter</code>	Equal	<code>fields.Boolean</code> , <code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.ToOne</code> , <code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>
ne	<code>filters.NotEqualFilter</code>	Not equal	<code>fields.Boolean</code> , <code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.ToOne</code>
in	<code>filters.InFilter</code>	In (expects an Array)	<code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>
contains	<code>filters.ContainsFilter</code>	Contains	<code>fields.Array</code> , <code>fields.ToMany</code>
lt	<code>filters.LessThanFilter</code>	Less than	<code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>
gt	<code>filters.GreaterThanFilter</code>	Greater than	<code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>
lte	<code>filters.LessThanEqualFilter</code>	Less than or equal	<code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>
gte	<code>filters.GreaterThanEqualFilter</code>	Greater than or equal	<code>fields.String</code> , <code>fields.Integer</code> , <code>fields.Number</code> , <code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>
contains	<code>filters.StringContainsFilter</code>	Contains (String)	<code>fields.String</code>
icontains	<code>filters.StringIContainsFilter</code>	Contains (String, case-insensitive)	<code>fields.String</code>
startswith	<code>filters.StartsWithFilter</code>	Starts with	<code>fields.String</code>
endswith	<code>filters.EndsWithFilter</code>	Ends with	<code>fields.String</code>
istartswith	<code>filters.StartsWithFilter</code>	Starts with (case-insensitive)	<code>fields.String</code>
iendswith	<code>filters.EndsWithFilter</code>	Ends with (case-insensitive)	<code>fields.String</code>
between	<code>filters.DateBetweenFilter</code>	Date between	<code>fields.Date</code> , <code>fields.DateTime</code> , <code>fields.DateString</code> , <code>fields.DateTimeString</code>

Note: `filters.EqualFilter` uses both the keys 'eq' and None. This is so that you can write an equality comparison both ways:

```
GET /user?where={"name": "foo"}
GET /user?where={"name": {"$eq": "foo"}}
```

filters.BaseFilter

New in version 0.11.

class flask_potion.filters.**BaseFilter** (*name*, *field=None*, *attribute=None*)

Base-class for all filter types. Filters are specified on a field-level. Each backend implements its own filters and defaults. Custom filters can be specified using the `ModelResource.Meta.filter` configuration.

Named and unnamed filters:

`EqualFilter` is a special filter type. This is because an equality condition is can be written in the format `{"property": condition}`, whereas every other filter needs to be written as `{"property": {"$filter": condition}}`. To implement this, a filter can be either named or unnamed.

Due to the way the equality comparison is done, users need to be watchful when comparing objects. Some object comparisons can be ambiguous, e.g. `{"foo": {"$foo": "bar"}}`. If a condition contains an object with exactly one property, the name of the property will be matched against all valid filters for that field. If necessary, the equality filter can be declared explicitly to avoid comparing against the wrong filter, e.g. `{"foo": {"$eq": {"$foo": "bar"}}}`.

Multiple filters can have the same filter name so long as they are not valid for the same field types. For example, `StringContainsFilter` for strings and `ContainsFilter` for arrays.

attribute

Attribute to filter on. Defaults to `field.attribute`.

field

Field to filter on.

name

Name of the filter as specified in the `where` object in the GET request. A filter `foo` on field `field` is specified as: `?where={"field": {"$foo": filter-expression}}`

op (*a*, *b*)

Matches an attribute of an item *a* against a value *b* provided by the user.

Parameters

- **a** – item’s attribute’s value
- **b** – value filtered by

Returns True on match, False otherwise

schema ()

Returns the schema for this filter.

This depends on the name of the filter. If the filter is named, it needs to be formatted as `{"$name": schema}`. Usually the equality filter is unnamed and all other filters are named.

Api class

There is not much to say about `Api` except that it has an optional `prefix` and `decorators`. Use:

```
api.add_resource(YourResource)
```

To add a resource to the API. You can only add a single resource with a given name.

class flask_potion.**Api** (*app=None*, *decorators=None*, *prefix=None*, *title=None*, *description=None*, *default_manager=None*)

This is the Potion extension.

You need to register `Api` with a Flask application either upon initializing `Api` or later using `init_app()`.

Parameters

- **app** – a `Flask` instance
- **decorators** (*list*) – an optional list of decorator functions
- **prefix** – an optional API prefix. Must start with “/”
- **title** (*str*) – an optional title for the schema
- **description** (*str*) – an optional description for the schema
- **default_manager** (*Manager*) – an optional manager to use as default. If `SQLAlchemy` is installed, will use `contrib.alchemy.SQLAlchemyManager`

add_resource (*resource*)

Add a `Resource` class to the API and generate endpoints for all its routes.

Parameters **resource** (*Resource*) – resource

Returns

Routes & Route Sets

class `routes.Route` (*method=None, view_func=None, rule=None, attribute=None, rel=None, title=None, description=None, schema=None, response_schema=None, format_response=True*)

Routes are not bound to a specific resource and their schema, generated using `schema_factory()` can vary depending on the resource.

If `view_func` has an `__annotations__` attribute (a Python 3.x function annotation), the annotations will be used to generate the `request_schema` and `response_schema`. The `return` annotation in this case is expected to be a `schema.Schema` used for responses, and all other annotations are expected to be of type `fields.Raw` and are combined into a `schema.Fieldset`.

relation

A relation for the string, equal to `rel` if one was given.

request_schema

request schema (not resource-bound)

response_schema

response schema (not resource-bound)

@METHOD (*rule=None, attribute=None, rel=None, title=None, description=None, schema=None, response_schema=None, format_response=True*)

A decorator for registering the `METHOD` method handler of a route. Can be used with or without arguments and on both class and route instances. The `rule` and `attribute` arguments are only available on the class.

When used with an instance will add or replace the view function for the `METHOD` method of this `Route` with the decorated function; otherwise instantiates a new `Route` with the view function.

This decorator is defined for the `GET`, `PUT`, `POST`, `PATCH` and `DELETE` methods.

Parameters

- **rule** (*str*) – (class-only) route URI relative to the resource, defaults to `{attribute}`, replacing any `'_'` (underscore) in `attribute` with `'-'` (dash).
- **attribute** (*str*) – (class-only) attribute on the parent resource, used to identify the route internally; defaults to the attribute name of the decorated view function within the parent resource.
- **rel** (*str*) – relation of the method link to the resource

- **title** (*str*) – title of link schema
- **description** (*str*) – description of link schema
- **schema** (*schema.Schema*) – request schema
- **response_schema** (*schema.Schema*) – response schema
- **format_response** (*bool*) – whether the response should be converted using the response schema

schema

Used to get and set the request schema for the most recently decorated request method view function

response_schema

Used to get and set the response schema for the most recently decorated request method view function

method_links

A dictionary mapping of method names (in upper case) to `routes.Link` objects containing the method view functions.

Parameters

- **method** (*str*) – a HTTP request method name (upper case)
- **view_func** (*callable*) – view function
- **rule** – url rule string or callable returning a string
- **rel** (*str*) – relation
- **title** (*str*) – title of schema
- **description** (*str*) – description of schema
- **route** (*routes.Route*) – route this link belongs to
- **schema** (*schema.Schema*) – request schema
- **response_schema** (*schema.Schema*) – response schema
- **format_response** (*bool*) – whether the response should be converted using the response schema

schema_factory (*resource*)

Returns a link schema for a specific resource.

rule_factory (*resource*, *relative=False*)

Returns a URL rule string for this route and resource.

Parameters

- **resource** (*flask_potion.Resource*) –
- **relative** (*bool*) – whether the rule should be relative to resource.
route_prefix

view_factory (*name*, *resource*)

Returns a view function for all links within this route and resource.

Parameters

- **name** – Flask view name
- **resource** (*flask_potion.Resource*) –

```
class routes.ItemRoute (method=None, view_func=None, rule=None, attribute=None, rel=None, title=None, description=None, schema=None, response_schema=None, format_response=True)
```

This route can be used with `flask_potion.ModelResource`. It is a simple extension over `Route` with the following adjustments:

- `rule_factory()` is changed to `prefix <{id_converter}:id>` with any rule.
- It changes the implementation of `view_factory()` so that it passes the resolved resource item matching `id` as the first positional argument to the view function.

```
class routes.RouteSet
```

An abstract class for combining related routes into one, which can also be used as a route factory.

```
routes ()
```

Returns an iterator over `Route` objects

```
class routes.Relation (resource, backref=None, io='rw', attribute=None, **kwargs)
```

Used to define a relation to another `ModelResource`.

```
class routes.ItemAttributeRoute (cls_or_instance, io=None, attribute=None)
```

Parameters

- **cls_or_instance** (`fields.Raw`) – a field class or instance
- **attribute** (`str`) – defaults to the field's `attribute` attribute
- **io** (`str`) – r, u, or ru - defaults to the field's `io` attribute

Resources

Resource

`Resource` is the base class for all other resource types and by contains only one route, which returns the schema for the resource.

```
class flask_potion.Resource
```

A plain resource with nothing but a schema.

A resource is configured using the `Schema` and `Meta` attributes as well as any properties that are of type `routes.Route` or `routes.RouteSet`.

Meta class attributes:

Attribute name	De-fault	Description
<code>name</code>	—	Name of the resource; defaults to the lower-case of the <i>model's</i> class name
<code>title</code>	None	JSON-schema title declaration
<code>description</code>	None	JSON-schema description declaration
<code>exclude_routes</code>	<code>()</code>	A list of strings; any routes — including inherited routes — whose <code>Route.relation</code> match one of these string is omitted from the resource.
<code>route_decorators</code>	<code>()</code>	A dictionary of decorators to apply to routes in the resource. The keys must match the <code>Route.relation</code> attribute.
<code>exclude_fields</code>	<code>()</code>	A list of fields that should not be imported from the <i>model</i> .
<code>required_fields</code>	<code>()</code>	Fields that are automatically imported from the model are automatically required if their columns are not <i>nullable</i> and do not have a <i>default</i> .
<code>read_only_fields</code>	<code>()</code>	A list of fields that are returned by the resource but are ignored in <i>POST</i> and <i>PATCH</i> requests. Useful for e.g. timestamps.
<code>write_only_fields</code>	<code>()</code>	A list of fields that can be written to but are not returned. For secret stuff.

Usage example:

```
class LogResource(Resource):
    class Schema:
        level = fields.String(enum=['info', 'warning', 'error'])
        message = fields.String()

    class Meta:
        name = 'log'

    @Route.POST('',
                rel="create",
                schema=fields.Inline('self'),
                response_schema=fields.Inline('self'))
    def create(self, properties):
        print('{level}: {message}'.format(**properties))
        return properties
```

api

Back reference to the [Api](#) this resource is registered on.

meta

A `AttributeDict` of configuration attributes collected from the `Meta` attributes of the base classes.

routes

A dictionary of routes registered with this resource. Keyed by `Route.relation`.

schema

A `FieldSet` containing fields collected from the `Schema` attributes of the base classes.

route_prefix

The prefix URI to any route in this resource; includes the API prefix.

described_by()

A `Route` at `/schema` that contains the JSON Hyper-Schema for this resource.

ModelResource

ModelResource is written for create, read, update, delete actions on collections of items matching the resource

schema.

A data store connection is maintained by a *manager.Manager* instance. The manager class can be specified in `Meta.manager`; if no manager is specified, `Api.default_manager` is used. Managers are configured through attributes in `Meta`. Most managers expect a *model* to be defined under `Meta.model`.

class flask_potion.ModelResource

create()

A link — part of a `Route` at the root of the resource — for creating new items.

Parameters `properties` –

Returns created item

instances()

A link — part of a `Route` at the root of the resource — for reading item instances.

Parameters

- `where` –
- `sort` –
- `page(int)` –
- `per_page(int)` –

Returns list of items

read()

A link — part of a `Route` at `/<{Resource.meta.id_converter}:id>` — for reading a specific item.

Parameters `id` – item id

Returns item

update()

A link — part of a `Route` at `/<{Resource.meta.id_converter}:id>` — for updating a specific item.

Parameters

- `id` – item id
- `properties` – changes

Returns item

destroy()

A link — part of a `Route` at `/<{Resource.meta.id_converter}:id>` — for deleting a specific item.

Parameters `id` – item id

Returns `(None, 204)`

Managers

Manager base class

`manager.Manager` is used by `ModelResource` to implement a backend integration.

```
class flask_potion.manager.Manager(resource, model)
```

Parameters

- **resource** (`flask_potion.resource.Resource`) – resource class
- **model** – model read from `Meta.model` or `None`

```
relation_instances(item, attribute, target_resource, page=None, per_page=None)
```

Parameters

- **item** –
- **attribute** –
- **target_resource** –
- **page** –
- **per_page** –

Returns

```
relation_add(item, attribute, target_resource, target_item)
```

Parameters

- **item** –
- **attribute** –
- **target_resource** –
- **target_item** –

Returns

```
relation_remove(item, attribute, target_resource, target_item)
```

Parameters

- **item** –
- **attribute** –
- **target_resource** –
- **target_item** –

Returns

```
paginated_instances(page, per_page, where=None, sort=None)
```

Parameters

- **page** –
- **per_page** –
- **where** –
- **sort** –

Returns a `Pagination` object or similar

instances (*where=None, sort=None*)

Parameters

- **where** –
- **sort** –

Returns

first (*where=None, sort=None*)

Parameters

- **where** –
- **sort** –

Returns

Raises `exceptions.ItemNotFound` –

create (*properties, commit=True*)

Parameters

- **properties** –
- **commit** –

Returns

read (*id*)

Parameters *id* –

Returns

update (*item, changes, commit=True*)

Parameters

- **item** –
- **changes** –
- **commit** –

Returns

delete (*item*)

Parameters *item* –

Returns

delete_by_id (*id*)

Parameters *id* –

Returns

class `flask_potion.manager.RelationalManager` (*resource, model*)

`RelationalManager` is a base class for managers that do relational lookups on the basis of a query builder.

Manager implementations

The following backend managers ship with *Flask-Potion*:

class `contrib.memory.MemoryManager(resource, model)`
An in-memory, pure-python Manager implementation.

Warning: This manager is intended for debugging & testing only and should not be used in production.

class `contrib.alchemy.SQLAlchemyManager(resource, model)`
A manager for SQLAlchemy models.
Expects that `Meta.model` contains a SQLAlchemy declarative model.

class `contrib.peewee.PeeWeeManager(resource, model)`
A manager for Peewee models.

Additionally, `contrib.alchemy.SQLAlchemyManager` can be extended with `contrib.principals.PrincipalsMixin` to form a new manager that implements a permissions system based on *Flask-Principals*.

Signals

Potion comes with several [Blinker](#) signals. The signals can be used to pre-process and post-process most parts of the read, create, update cycle.

Resources using the `SQLAlchemyManager` or `PeeWeeManager` hook into these signals. Other Manager implementations should be written to hook into them as well.

Signal listeners can edit the item:

```
>>> @before_create.connect_via(ArticleResource)
... def on_before_create_article(sender, item):
...     item.author_id = current_user.id
```

Listeners may also raise exceptions:

```
>>> @before_create.connect_via(ArticleResource)
... def on_before_create_article(sender, item):
...     if not current_user.is_editor:
...         raise BadRequest()
```

The complete list of signals:

class `signals.before_create`

Parameters

- **sender** – item resource
- **item** – instance of item

class `signals.after_create`

Parameters

- **sender** – item resource
- **item** – instance of item

`class signals.before_update`

Parameters

- **sender** – item resource
- **item** – instance of item
- **changes** (*dict*) – dictionary of changes, already parsed

`class signals.after_update`

Parameters

- **sender** – item resource
- **item** – instance of item
- **changes** (*dict*) – dictionary of changes, already parsed

`class signals.before_delete`

Parameters

- **sender** – item resource
- **item** – instance of item

`class signals.after_delete`

Parameters

- **sender** – item resource
- **item** – instance of item

`class signals.before_add_to_relation`

Parameters

- **sender** – parent resource
- **item** – instance of parent item
- **attribute** – name of relationship to child
- **child** – instance of child item

`class signals.after_add_to_relation`

Parameters

- **sender** – parent resource
- **item** – instance of parent item
- **attribute** – name of relationship to child
- **child** – instance of child item

`class signals.before_remove_from_relation`

Parameters

- **sender** – parent resource
- **item** – instance of parent item
- **attribute** – name of relationship to child
- **child** – instance of child item

`class signals.after_remove_from_relation`

Parameters

- **sender** – parent resource
- **item** – instance of parent item
- **attribute** – name of relationship to child
- **child** – instance of child item

Note: Relation-related signals are only used by `Relation`, They do not apply to relations created or removed by updating an item with `fields.ToOne` or `fields.ToMany` fields.

Permissions with *Flask-Principal*

Flask-Potion includes a permission system. The permissions system is built on `Flask-Principal`. and enabled by decorating a `manager.RelationalManager` with `contrib.principals.principals`, which returns a class extending both the manager and `contrib.principals.PrincipalMixin`.

Permissions are specified as a dict in `Meta.permissions`.

Defining Permissions

There are four basic *actions* — read, create, update, delete — for which permissions must be defined. Additional virtual actions can be declared for various purposes.

For example, the default permission declaration looks somewhat like this:

```
class Meta:
    permissions = {
        'read': 'yes',
        'create': 'no',
        'update': 'create',
        'delete': 'update'
    }
```

Patterns and *Needs* they produce:

Pattern	Matches	Description
{action}	a key in the permissions dict If	equal to the action it is declared for — e.g. {'create': 'create'} — evaluate to: <code>HybridItemNeed({action}, resource_name)</code> Otherwise re-use needs from other action.
{role}	not a key in the permissions dict	<code>RoleNeed({role})</code>
{action}:{field}	*.*	Copy {action} permissions from ToOne linked resource at {field}.
user:{field}	user:*	<code>UserNeed(item.{field}.id)</code> for ToOne fields.
no, nobody	no	Do not permit.
yes, everybody	yes	Always permit.

Note: When protecting an `ItemRoute`, read access permissions, and updates using the resource manager are checked automatically; for other actions, permissions have to be checked manually from within the function. The manager has helper functions such as `PrincipalMixin.can_update_item()` to facilitate this.

Example API with permissions

Changed in version 0.11: The `PrincipalManager` extending `SQLAlchemyManager` has been replaced by a `principals()` class-decorator.

We’re going to go ahead and create an example API using `PrincipalMixin` with [Flask-Login](#) for authentication. Since there are quite a few moving parts, this example is split up into several sections.

Our example is a simple blog with *articles* and *comments*. First, let’s create the database models:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import UserMixin
from sqlalchemy.orm import relationship

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret' # XXX replace with actual secret and don't keep_
    ↳ it in source code

db = SQLAlchemy(app)

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(), nullable=False)
    is_admin = db.Column(db.Boolean(), default=False)
    is_editor = db.Column(db.Boolean(), default=False)

class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey(User.id), nullable=False)
    author = relationship(User)
    content = db.Column(db.Text)
```

```
class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    article_id = db.Column(db.Integer, db.ForeignKey(Article.id), nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey(User.id), nullable=False)
    article = relationship(Article)
    author = relationship(User)
    message = db.Column(db.Text)

db.create_all()
```

We're going to use *Flask-Login* to authenticate requests using *Basic Authentication*:

```
from flask_login import LoginManager, current_user

login_manager = LoginManager(app)

@login_manager.request_loader
def load_user_from_request(request):
    if request.authorization:
        username, password = request.authorization.username, request.authorization.
        ↪password

        # XXX replace this with an actual password check.
        if username == password:
            return User.query.filter_by(username=username).first()
    return None
```

This is where *Flask-Principal* comes in. With every request it adds the *needs* the identity should provide. Authenticated users are given a *user need* and maybe some *role needs*. If this example had some top-level object based permissions (think groups, projects, teams, etc.) they would also be added here.

```
from flask_principal import Principal, Identity, UserNeed, AnonymousIdentity,
    ↪identity_loaded, RoleNeed

principals = Principal(app)

@principals.identity_loader
def read_identity_from_flask_login():
    if current_user.is_authenticated():
        return Identity(current_user.id)
    return AnonymousIdentity()

@identity_loaded.connect_via(app)
def on_identity_loaded(sender, identity):
    if not isinstance(identity, AnonymousIdentity):
        identity.provides.add(UserNeed(identity.id))

        if current_user.is_editor:
            identity.provides.add(RoleNeed('editor'))

        if current_user.is_admin:
            identity.provides.add(RoleNeed('admin'))
```


Finally, we create our API with the `login_required` decorator from *Flask-Login*.

```
from flask_login import login_required
from flask_potion import fields, signals, Api, ModelResource
from flask_potion.contrib.alchemy import SQLAlchemyManager
from flask_potion.contrib.principals import principals

api = Api(app, decorators=[login_required])

class PrincipalResource(ModelResource):
    class Meta:
        manager = principals(SQLAlchemyManager)

class UserResource(PrincipalResource):
    class Meta:
        model = User

class ArticleResource(PrincipalResource):
    class Schema:
        author = fields.ToOne('user')

    class Meta:
        model = Article
        read_only_fields = ['author']
        permissions = {
            'create': 'editor',
            'update': ['user:author', 'admin']
        }

class CommentResource(PrincipalResource):
    class Schema:
        article = fields.ToOne('article')
        author = fields.ToOne('user')

    class Meta:
        model = Comment
        read_only_fields = ['author']
        permissions = {
            'create': 'anybody',
            'update': 'user:author',
            'delete': ['update:article', 'admin']
        }

api.add_resource(UserResource)
api.add_resource(ArticleResource)
api.add_resource(CommentResource)

# add the author to articles & comments when they are created:
@signals.before_create.connect_via(ANY)
def before_create_article_comment(sender, item):
    if issubclass(sender, (ArticleResource, CommentResource)):
        item.author_id = current_user.id
```

We've implemented the following permissions:

- only editors can create articles

- articles can be updated or deleted by either their authors or by admins
- comments can be created by anyone who is authenticated
- comments can updated only by the person who wrote the comment, but deleted both by admins and the author of the article

Now we just need to start the app:

```
if __name__ == '__main__':
    # add some example users & run the application
    db.session.add(User(username='editorA', is_editor=True))
    db.session.add(User(username='editorB', is_editor=True))
    db.session.add(User(username='admin', is_admin=True))
    db.session.add(User(username='user'))
    db.session.commit()

    app.run()
```

You can find the complete example code on GitHub under:

```
examples/permissions_example.py
```

```
http --auth editorA:editorA :5000/article content=foo
```

```
HTTP/1.0 200 OK
Content-Length: 71
Content-Type: application/json
Date: Sun, 08 Feb 2015 10:48:03 GMT
Server: Werkzeug/0.9.6 Python/3.3.2
Set-Cookie: session=
↪eJyrVorPTFGyUjK3SEw0TD0zSDRPtDRJtUxNMzZKM0pNNE4zNks1TU6zVNJRyKxJzSvJLKnUSywtYgvgSxIVbLKK83JQZIBGW
↪B7jQYw.Nhh6qE-h5WrGPfsYibXnDzCaJQM; HttpOnly; Path=/

{
  "$uri": "/article/2",
  "author": {
    "$ref": "/user/1"
  },
  "content": "foo"
}
```

Object-based permissions

The example above did already *sort of* touch on object-based permissions, with the 'user:author' pattern that restricts access to the user who has authored a *comment* or *article*. We've also used permissions options, with more than one *need* potentially providing access. Finally, you have seen a hint of cascading object-based permissions with the 'update:article' pattern that conditions access to the permissions on a relation.

There is another permission layer, building on `flask_principal.ItemNeed`, for object-specific permissions. You would want to use them on something important, such as this *project* resource:

```
class ProjectResource(ModelResource):
    class Meta:
        manager = principals(SQLAlchemyManager)
        model = Project
        permissions = {
```

```

        'create': 'anybody',
        'update': 'manage',
        'manage': 'manage'
    }

```

To update a project, your identity needs this *need*:

```
ItemNeed('manage', PROJECT_ID, 'project')
```

The pair {'manage': 'manage'} makes manage a new virtual action, which is why the flask_principals.ItemNeed wants a 'manage' permission. We could also have written {'update': 'update'} — then the required *need* would have been:

```
ItemNeed('update', PROJECT_ID, 'project')
```

With cascading permissions, role-based, user-based, and object-based permissions you should now have all the tools to implement all sorts of complex permissions setups.

PrincipalMixin class

```
class flask_potion.contrib.principals.PrincipalMixin(*args, **kwargs)
```

get_permissions_for_item(item)

Returns a dictionary of evaluated permissions for an item. :param item: :return: Dictionary in the form {operation: bool, ..}

can_create_item(item)

Looks up permissions on whether an item may be created. :param item:

can_update_item(item, changes=None)

Looks up permissions on whether an item may be updated. :param item: :param changes: dictionary of changes

can_delete_item(item)

Looks up permissions on whether an item may be deleted. :param item:

Efficiency

Those who have worked with Flask-Principal know that it is on its own not well-suited for object-based permissions where large numbers of objects are involved, because each permission has to be loaded into memory as ItemNeed at the start of the session.

The permission system built into Potion introduces the HybridNeed and HybridPermission classes to solve this issue. They can either be evaluated directly or be applied to SQLAlchemy queries, and are therefore efficient with any number of object-based permissions.

```
class flask_potion.contrib.principals.needs.HybridNeed
```

HybridNeed base class. Hybrid needs can both be evaluated directly or produce an expression for use with SQLAlchemy.

```
class flask_potion.contrib.principals.permission.HybridPermission(*needs)
```

Hybrid flask_principal.Permission that evaluates both regular and hybrid needs.

allows(identity)

Determines whether a given identity meets this permission.

Parameters `identity` (`flask_principal.Identity`) – An identity with a set of provided *needs*

can (`item=None`)

Depending on whether or not `item` is given, this function either:

- evaluates all regular needs *needs*
- also evaluates the hybrid needs against the item

If any of the needs are met, the function returns `True`.

Parameters `item` – SQLAlchemy model instance

Advanced Recipes

HistoryMixin

This mixin keeps a simple history of changes that have been made to a resource, storing them in a database table with a JSON field. HistoryMixin is a drop-in addition to any ModelResource.

```
ChangeSet = fields.Object({
    "updated_at": fields.DateTime(),
    "changes": fields.List(fields.Object({
        "attribute": fields.String(),
        "old": fields.Any(nullable=True),
        "new": fields.Any(nullable=True)
    }))
})

class HistoryRecord(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    object_type = db.Column(db.String(20), index=True, nullable=False)
    object_id = db.Column(db.Integer, index=True, nullable=False)
    updated_at = db.Column(db.DateTime, default=func.now(), nullable=False)
    changes = db.Column(postgresql.JSONB)

    __mapper_args__ = {
        "order_by": "updated_at"
    }

class HistoryMixin(object):
    @ItemRoute.GET('/history', rel="history")
    def history(self, item) -> fields.List(ChangeSet):
        history = HistoryRecord.query \
            .filter_by(object_type=self.meta.model.__tablename__,
                      object_id=getattr(item, self.meta.get('id_attribute', 'id')) \
            .all()

        return history

@before_update.connect_via(ANY, weak=False)
def history_on_update(resource, item, changes):
    if issubclass(resource, HistoryMixin):
```

```

        history = HistoryRecord(object_type=item.__tablename__,
                                object_id=getattr(item, resource.meta.get('id_
↪attribute', 'id')),
                                changes=[])

        fields_by_attribute = {
            field.attribute or key: field for key, field in resource.schema.fields.
↪items()
        }

        for attribute, change in changes.items():
            field = fields_by_attribute[attribute]
            history.changes.append({
                "attribute": attribute,
                "old": field.output(attribute, item),
                "new": field.output(attribute, changes)
            })

        db.session.add(history)

```

ArchivingResource

Sometimes soft-deletion is preferable over full deletion. This custom `ModelResource` and `Manager` does not delete items, instead it *archives* them, removing them from the main instances route. Archived items can be viewed in the archive route from where they can be restored but not updated.

Replace `RelationalManager` with an appropriate base class, such as `SQLAlchemyManager`. `PrincipalManager` can also be used as the base class for the manager with some minor changes.

```

class Location(Enum):
    ARCHIVE_ONLY = 1
    INSTANCES_ONLY = 2
    BOTH = 3

class ArchiveManager(RelationalManager):
    def _query(self, source=Location.INSTANCES_ONLY):
        query = super()._query(self)

        if source == Location.BOTH:
            return query
        elif source == Location.ARCHIVE_ONLY:
            return query.filter(getattr(self.model, 'is_archived') == True)
        else:
            return query.filter(getattr(self.model, 'is_archived') == False)

    def instances(self, where=None, sort=None, source=Location.INSTANCES_ONLY):
        query = self._query(source)
        if where:
            expressions = [self._expression_for_condition(condition) for condition in_
↪where]
            query = self._query_filter(query, self._and_expression(expressions))
        if sort:
            query = self._query_order_by(query, sort)
        return query

```

```

def archive_instances(self, page, per_page, where=None, sort=None):
    return self\
        .instances(where=where, sort=sort, source=Location.ARCHIVE_ONLY)\
        .paginate(page=page, per_page=per_page)

def read(self, id, source=Location.INSTANCE_ONLY):
    query = self._query(source)
    if query is None:
        raise ItemNotFound(self.resource, id=id)
    return self._query_filter_by_id(query, id)

class ArchivingResource(ModelResource):
    class Meta:
        manager = ArchiveManager
        exclude_routes = ['destroy'] # we're using rel="archive" instead.

    class Schema:
        is_archived = fields.Boolean(if='r')

    @Route.GET('/<int:id>', rel="self", attribute="instance")
    def read(self, id) -> fields.Inline('self'):
        return self.manager.read(id, source=Location.BOTH)

    @read.PATCH(rel="update")
    def update(self, properties, id):
        item = self.manager.read(id, source=Location.INSTANCE_ONLY)
        updated_item = self.manager.update(item, properties)
        return updated_item

    update.response_schema = update.request_schema = fields.Inline('self', patch_
    ↪instance=True)

    @update.DELETE(rel="archive")
    def destroy(self, id):
        item = self.manager.read(id, source=Location.INSTANCE_ONLY)
        self.manager.update(item, {"is_archived": True})
        return None, 204

    @Route.GET("/archive")
    def archive_instances(self, **kwargs):
        return self.manager.archive_instances(**kwargs)

    archive_instances.request_schema = archive_instances.response_schema = Instances()

    @Route.GET('/archive/<int:id>', rel="readArchived")
    def read_archive(self, id) -> fields.Inline('self'):
        item = self.manager.read(id, source=Location.ARCHIVE_ONLY)

    @Route.POST('/archive/<int:id>/restore', rel="restoreFromArchive")
    def restore_from_archive(self, id) -> fields.Inline('self'):
        item = self.manager.read(id, source=Location.ARCHIVE_ONLY)
        return self.manager.update(item, {"is_archived": False})

```

f

`fields`, [17](#)
`flask_potion`, [27](#)
`flask_potion.contrib.principals`, [39](#)
`flask_potion.contrib.principals.needs`,
 [39](#)
`flask_potion.contrib.principals.permission`,
 [39](#)
`flask_potion.filters`, [23](#)

r

`routes`, [25](#)

s

`schema`, [20](#)
`signals`, [32](#)

A

[add_resource\(\)](#) (flask_potion.Api method), 25
[after_add_to_relation](#) (class in signals), 33
[after_create](#) (class in signals), 32
[after_delete](#) (class in signals), 33
[after_remove_from_relation](#) (class in signals), 33
[after_update](#) (class in signals), 33
[allows\(\)](#) (flask_potion.contrib.principals.permission.HybridPermission method), 39
[Any](#) (class in fields), 18
[Api](#) (class in flask_potion), 24
[api](#) (flask_potion.Resource attribute), 28
[Array](#) (class in fields), 19
[attribute](#) (flask_potion.filters.BaseFilter attribute), 24
[AttributeMapped](#) (class in fields), 19

B

[BaseFilter](#) (class in flask_potion.filters), 23
[before_add_to_relation](#) (class in signals), 33
[before_create](#) (class in signals), 32
[before_delete](#) (class in signals), 33
[before_remove_from_relation](#) (class in signals), 33
[before_update](#) (class in signals), 32
[Boolean](#) (class in fields), 18

C

[can\(\)](#) (flask_potion.contrib.principals.permission.HybridPermission method), 40
[can_create_item\(\)](#) (flask_potion.contrib.principals.PrincipalMixin method), 39
[can_delete_item\(\)](#) (flask_potion.contrib.principals.PrincipalMixin method), 39
[can_update_item\(\)](#) (flask_potion.contrib.principals.PrincipalMixin method), 39
[contrib.alchemy.fields.InlineModel](#) (class in fields), 19
[convert\(\)](#) (fields.Raw method), 17
[convert\(\)](#) (schema.FieldSet method), 21
[convert\(\)](#) (schema.Schema method), 21
[create\(\)](#) (flask_potion.manager.Manager method), 31

[create\(\)](#) (flask_potion.ModelResource method), 29
[Custom](#) (class in fields), 18

D

[Date](#) (class in fields), 18
[DateString](#) (class in fields), 18
[DateTime](#) (class in fields), 18
[DateTimeString](#) (class in fields), 18
[delete\(\)](#) (flask_potion.manager.Manager method), 31
[delete_by_id\(\)](#) (flask_potion.manager.Manager method), 31
[described_by\(\)](#) (flask_potion.Resource method), 28
[destroy\(\)](#) (flask_potion.ModelResource method), 29

F

[field](#) (flask_potion.filters.BaseFilter attribute), 24
[fields](#) (module), 17
[FieldSet](#) (class in schema), 21
[first\(\)](#) (flask_potion.manager.Manager method), 31
[flask_potion](#) (module), 3, 21, 24, 27, 30, 34
[flask_potion.contrib.principals](#) (module), 39
[flask_potion.contrib.principals.needs](#) (module), 39
[flask_potion.contrib.principals.permission](#) (module), 39
[flask_potion.filters](#) (module), 23
[format\(\)](#) (fields.Raw method), 17
[format\(\)](#) (schema.Schema method), 20
[format_response\(\)](#) (schema.Schema method), 21

G

[get_permissions_for_item\(\)](#) (flask_potion.contrib.principals.PrincipalMixin method), 39

H

[HybridNeed](#) (class in flask_potion.contrib.principals.needs), 39
[HybridPermission](#) (class in flask_potion.contrib.principals.permission), 39

I

Inline (class in fields), 20
 instances() (flask_potion.manager.Manager method), 31
 instances() (flask_potion.ModelResource method), 29
 Integer (class in fields), 18
 ItemAttributeRoute (class in routes), 27
 ItemRoute (class in routes), 26
 ItemType (class in fields), 20
 ItemUri (class in fields), 20

M

Manager (class in flask_potion.manager), 30
 MemoryManager (class in contrib.memory), 32
 meta (flask_potion.Resource attribute), 28
 METHOD() (routes.Route method), 25
 method_links (routes.Route attribute), 26
 ModelResource (class in flask_potion), 29

N

name (flask_potion.filters.BaseFilter attribute), 24
 Number (class in fields), 18

O

Object (class in fields), 19
 op() (flask_potion.filters.BaseFilter method), 24

P

paginated_instances() (flask_potion.manager.Manager method), 30
 parse_request() (schema.Schema method), 21
 PeeweeManager (class in contrib.peewee), 32
 PositiveInteger (class in fields), 18
 PrincipalMixin (class in flask_potion.contrib.principals), 39

R

Raw (class in fields), 17
 read() (flask_potion.manager.Manager method), 31
 read() (flask_potion.ModelResource method), 29
 Relation (class in routes), 27
 relation (routes.Route attribute), 25
 relation_add() (flask_potion.manager.Manager method), 30
 relation_instances() (flask_potion.manager.Manager method), 30
 relation_remove() (flask_potion.manager.Manager method), 30
 RelationalManager (class in flask_potion.manager), 31
 request (schema.Schema attribute), 20
 request_schema (routes.Route attribute), 25
 Resource (class in flask_potion), 27
 response (schema.Schema attribute), 20
 response_schema (routes.Route attribute), 25, 26

Route (class in routes), 25
 route_prefix (flask_potion.Resource attribute), 28
 routes (flask_potion.Resource attribute), 28
 routes (module), 25
 routes() (routes.RouteSet method), 27
 RouteSet (class in routes), 27
 rule_factory() (routes.Route method), 26

S

Schema (class in schema), 20
 schema (flask_potion.Resource attribute), 28
 schema (module), 20
 schema (routes.Route attribute), 26
 schema() (fields.Raw method), 17
 schema() (flask_potion.filters.BaseFilter method), 24
 schema() (schema.Schema method), 20
 schema_factory() (routes.Route method), 26
 signals (module), 32
 SQLAlchemyManager (class in contrib.alchemy), 32
 String (class in fields), 18

T

ToMany (class in fields), 17
 ToOne (class in fields), 17

U

update() (flask_potion.manager.Manager method), 31
 update() (flask_potion.ModelResource method), 29
 Uri (class in fields), 18
 UUID (class in fields), 18

V

view_factory() (routes.Route method), 26