Arrested Documentation

Release 0.1.0

Mikey Waites

Jan 02, 2018

Contents


chat
on gitter

CHAPTER 1

Introducing Arrested

Take the pain out of REST API creation with Arrested - batteries included for quick wins, highly extensible for specialist requirements.

```
from arrested import ArrestedAPI, Resource, Endpoint, GetListMixin, CreateMixin,
from example.models import db, Character

api_v1 = ArrestedAPI(url_prefix='/v1')

characters_resource = Resource('characters', __name__, url_prefix='/characters')


class CharactersIndexEndpoint(Endpoint, GetListMixin, CreateMixin):

    name = 'list'
    many = True

    def get_objects(self):

        characters = db.session.query(Character).all()
        return characters

    def save_object(self, obj):

        character = Character(**obj)
        db.session.add(character)
        db.session.commit()
        return character


characters_resource.add_endpoint(CharactersIndexEndpoint)
api_v1.register_resource(characters_resource)
```

CHAPTER 2

Flask-Arrested Features

Arrested is a framework for rapidly building REST API's with Flask.

- Un-Opinionated: Let's you decide "the best way" to implement *your* REST API.
- Battle Tested: In use across many services in a production environment.
- Batteries Included! Arrested ships with built in support for popular libraries such as SQLAlchemy, Kim and Marshmallow. Using something different or have your own tooling you need to support? Arrested provides a rich API that can be easily customised!
- Supports any storage backends: Want to use "hot new database technology X?" No problem! Arrested can be easily extended to handle all your data needs.
- Powerful middleware system - Inject logic at any step of the request/response cycle

CHAPTER 3

Get started in under a minute..

Use the Flask-Arrested cookie cutter to create a basic API to get you started in 4 simple commands. https://github.com/mikeywaites/arrested-cookiecutter.

```
$ cookiecutter gh:mikeywaites/arrested-cookiecutter
$ cd arrested-users-api
$ docker-compose up -d api
$ curl -u admin:secret localhost:8080/v1/users | python -m json.tool
```

CHAPTER 4

The User Guide

Get started with Flask-Arrested using the quickstart user guide or take a look at the in-depth API documentation.

4.1 Introduction

Why Arrested?

4.2 Installation

This part of the documentation covers the installation of Arrested. The first step to using any software package is getting it properly installed.

4.2.1 Pip Install Flask-Arrested

To install Arrested, simply run this command in your terminal of choice:

```
$ pip install arrested
```

4.3 Guide

Eager to get going? This page gives an introduction to getting started with Flask-Arrested.

First, make sure that:

- Arrested is *installed*

This tutorial steps through creating a simple Star Wars themed REST API using Arrested. We assume you have a working knowledge of both Python and Flask.

4.3.1 Example Application

The examples used in this guide can be found here https://github.com/mikeywaites/flask-arrested/tree/master/example.

Note: To follow along with the example you will need to install Docker for your operating system. Find out how here https://docker.com.

Alternatively you can use the Arrested cookiecutter to create a working API in 4 simple commands. https://github.com/mikeywaites/arrested-cookiecutter.

4.3.2 APIs, Resources and Endpoints - Defining your first API

Flask-Arrested is split into 3 key concepts. *ArrestedAPI*'s, *Resource* and *Endpoint*. APIs contian multiple Resources. For example Our API contains a Characters resource, a Planets resource and so on. Resources are a collection of Endpoints. Endpoints define the urls inside of our Resources `/v1/charcaters` `/v1/planets/hfyf66775gjgjjf` etc.

```
from flask import Flask

from arrested import (
    ArrestedAPI, Resource, Endpoint, GetListMixin, CreateMixin,
    GetObjectMixin, PutObjectMixin, DeleteObjectMixin, ResponseHandler
)

from example.hanlders import DBRequestHandler, character_serializer
from example.models import db, Character

app = Flask(__name__)
api_v1 = ArrestedAPI(app, url_prefix='/v1')
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////opt/code/example/starwars.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db.init_app(app)

characters_resource = Resource('characters', __name__, url_prefix='/characters')


class CharactersIndexEndpoint(Endpoint, GetListMixin, CreateMixin):

    name = 'list'
    many = True
    response_handler = DBResponseHandler

    def get_response_handler_params(self, **params):

        params['serializer'] = character_serializer
        return params

    def get_objects(self):

        characters = db.session.query(Character).all()
        return characters

    def save_object(self, obj):

        character = Character(**obj)
```

```
        db.session.add(character)
        db.session.commit()
        return character


class CharacterObjectEndpoint(Endpoint, GetObjectMixin,
                              PutObjectMixin, DeleteObjectMixin):

    name = 'object'
    url = '/<string:obj_id>'
    response_handler = DBResponseHandler

    def get_response_handler_params(self, **params):

        params['serializer'] = character_serializer
        return params

    def get_object(self):

        obj_id = self.kwargs['obj_id']
        obj = db.session.query(Character).filter(Character.id == obj_id).one_or_none()
        if not obj:
            payload = {
                "message": "Character object not found.",
            }
            self.return_error(404, payload=payload)

        return obj

    def update_object(self, obj):

        data = self.request.data
        allowed_fields = ['name']

        for key, val in data.items():
            if key in allowed_fields:
                setattr(obj, key, val)

        db.session.add(obj)
        db.session.commit()

        return obj

    def delete_object(self, obj):

        db.session.delete(obj)
        db.session.commit()


characters_resource.add_endpoint(CharactersIndexEndpoint)
characters_resource.add_endpoint(CharacterObjectEndpoint)
api_v1.register_resource(characters_resource)
```

Start the Docker container in the example/ directory.

```
$ docker-compose run --rm --service-ports api
```

Fetch a list of Character objects..

```
curl -X GET localhost:5000/v1/characters | python -m json.tool

    {
            "payload": [
                    {
                            "created_at": "2017-06-04T11:47:02.017094",
                            "id": 1,
                            "name": "Obe Wan"
                    }
            ]
    }
```

Add a new Character..

```
curl -H "Content-Type: application/json" -d '{"name":"Darth Vader"}' -X POST␣
↪localhost:5000/v1/characters | python -m json.tool

    {
            "payload": [
                    {
                            "created_at": "2017-09-01T04:51:45.456072",
                            "id": 2,
                            "name": "Darth Vader"
                    }
            ]
    }
```

Fetch a Character by id..

```
curl -X GET localhost:5000/v1/characters/2 | python -m json.tool

    {
            "payload": {
        "created_at": "2017-09-01T04:51:45.456072",
        "id": 2,
        "name": "Darth Vader"
            }
    }
```

Update a Character by id..

```
curl -H "Content-Type: application/json" -d '{"id": 2, "name":"Anakin Skywalker",
↪"created_at": "2017-09-01T04:51:45.456072"}' -X PUT localhost:5000/v1/characters/2␣
↪| python -m json.tool

    {
            "payload": {
        "created_at": "2017-09-01T04:51:45.456072",
        "id": 2,
        "name": "Anakin Skywalker"
            }
    }
```

And finally, Delete a Character by id..

```
curl -X DELETE localhost:5000/v1/characters/2
```

URLS && url_for

URLSs are automatically defined by Resoruces and Endpoints using Flask's built in url_mapping functionality. We optionally provide Resource with a url_prefix which is applied to all of it's registered Endponts. We can also specify a URI segment for the Endpoint using the `url` parameter. Endpoints require that the name attribute is provied. This is the name used when reversing the url using Flask's `url_for` function. Ie *url_for('news.list')* where new is the name given to the Resource and list of the name of one of its registered endpoints.

Getting objects

We defined an Endpoint within our characters Resource that accepts incoming GET requests to /v1/characters. This Endpoint fetches all the Character objects from the database and our custom DBRequestHandler handles converting them into a format that can be serialized as JSON. The topic of Request and Response handling is covered in more detail below so for now let's take a closer look at the *GetListMixin* mixin.

GetListMixin provides automatic handling of GET requests. It requires that we define a single method *GetListMixin.get_objects*. This method should return data that our specified ResponseHandler can serialize.

We tell Arrested that this endpoint returns many objects using the *many* class attribute. This setting is used by certain Response handlers when serializing the objects returned by Endpoints.

```
import redis
from arrested import Endpoint, GetListMixin

class NewsEndpoint(Endpoint, GetListMixin):

    many = True
    name = 'list'

    def get_objects(self, obj):

        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news')
```

Saving objects

The CharactersIndexEndpoint also inherits the *CreateMixin*. This mixin provides functionality for handling POST requests. The *CreateMixin* requires that the `save_object` method be implemented. The save_object method will be called with the obj or objects processed by the Endpoint's defined request_handler.

Here's an example Endpoint that store the incoming JSON data in Redis.

```
import redis
from arrested import Endpoint, GetListMixin, CreateMixin

class CustomEndpoint(Endpoint, GetListMixin, CreateMixin):

    many = True
    name = 'list'

    def get_objects(self, obj):

        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news')
```

```
    def save_object(self, obj):

        # obj will be a dict here as we're using the default RequestHandler
        r.hmset('news', obj)
        return obj
```

Object Endpoints

Object endpoints allow you to define APIS that typically let your users GET, PUT, PATCH and DELETE single objects. The Mixins can be combined to provide support for all the typical HTTP methods used when working with a single object. Regardless of the HTTP methods you're supporting, your object endpoints must provide the *get_object* method.

Getting a single object

To support GET requests that retrieve a single object from an Endpoint you should use the *GetObjectMixin*. In addition to the get_object method, we have also specified a url class attribute. Arrested will populate a kwargs property on your Endpoint instance which contains the named url paramaters from your Endpoint's url.

Below we use the obj_id passed as part of the url to fetch a new item from Redis by ID.

```
import redis
from arrested import Endpoint, GetObjectMixin

class CustomEndpoint(Endpoint, GetObjectMixin):

    url = '/<str:obj_id>'
    name = 'object'

    def get_object(self, obj):

        news_id = self.kwargs['obj_id']
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news:%s' % news_id)
```

Updating an object

Support for updating objects is provided by the *PutObjectMixin*. PutObjectMixin requires two methods be implemented. *get_object* and *update_object*.

```
import redis
from arrested import Endpoint, PutObjectMixin

class CustomEndpoint(Endpoint, PutObjectMixin):

    url = '/<str:obj_id>'
    name = 'object'

    def get_object(self, obj):

        news_id = self.kwargs['obj_id']
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news:%s' % news_id)
```

```
    def update_object(self, obj):

        news_id = self.kwargs['obj_id']
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmset('news:%s' % news_id, obj)
```

When a PUT request is handled by our CustomEndpoint the *get_object* method is called first to retrieve the existing object. If an object is found the *PutObjectMixin.update_object* method is then called.

To support updating objects via PATCH requests all we need to do is use the *PatchObjectMixin*. It works in same way as *PutObjectMixin* except that we the *patch_object* method is called when an object is returned by get_object.

```
import redis
from arrested import Endpoint, PutObjectMixin, PatchObjectMixin

class CustomEndpoint(Endpoint, PutObjectMixin, PatchObjectMixin):

    url = '/<str:obj_id>'
    name = 'object'

    def get_object(self, obj):

        news_id = self.kwargs['obj_id']
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news:%s' % news_id)

    def do_update(self, obj):

        news_id = self.kwargs['obj_id']
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmset('news%s' % news_id, obj)

    def update_object(self, obj):

        self.do_update(obj)

    def patch_object(self, obj):

        self.do_update(obj)
```

Deleting objects

Support for deleting objects is provided by the *DeleteObjectMixin*. DeleteObjectMixin requires two methods be implemented. *get_object* and `delete_object`.

```
import redis
from arrested import Endpoint, DeleteObjectMixin

class CustomEndpoint(Endpoint, DeleteObjectMixin):

    url = '/<str:obj_id>'
    name = 'object'

    def get_object(self, obj):
```

```
        news_id = self.kwargs['obj_id']
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news:%s' % news_id)

    def delete_object(self, obj):

        news_id = self.kwargs['obj_id']
        return r.delete('news:%s' % news_id)
```

4.3.3 Middleware

Flask comes with a great system for defining request middleware. Arrested builds on top of this system to allow more fine grained control of where and when your middleware is run.

API Middleware

Middleware can be applied at each level of the Arrested stack. You will often want a piece middleware to be applied across every resource and every endpoint defined in an API. An example of this might be authentication. The *ArrestedAPI* object supports two middleware hooks, `before_all_hooks` and `after_all_hooks`. Let's create a basic example that demonstrates how authentication can be applied across APIs.

```
def authenticated(endpoint):
    token_valid = request.args.get('token') == 'test-token'
    if not token_valid:
        endpoint.return_error(401)

api_v1 = ArrestedAPI(app, url_prefix='/v1', before_all_hooks=[authenticated])
```

Hit the `http://localhost:5000/v1/characters` url in your browser. We now get a 401 status code when requesting the characters API. A second request, this time providng our API token should return our character objects. `http://localhost:5000/v1/characters?token=test-token`

Resource Middleware

Middleware can also be applied on a per Resource basis. *Resource*, Like the *ArrestedAPI* object also has two options for injecting middleware into the request/response cycle. `before_all_hooks` and `after_request_hook`. Let's add some logging code to our characters resource using an after request hook.

```
def log_request(endpoint, response):

    app.logger.debug('request to characters resource made')
    return response

characters_resource = Resource('characters', __name__, url_prefix='/characters',␣
↪after_all_hooks=[log_request])
```

Our middleware is slightly different from the authenication example. When we're dealing with an after request hook we are also passed the response object as well as the endpoint instance. The response object should be returned from every after request hook defined on our APIs and Resources.

Endpoint Middleware

Lastly we come to the *Endpoint* object. *Endpoint* supports defining middleware using the following hooks:

- before_all_hooks
- before_get_hooks
- after_get_hooks
- before_post_hooks
- after_post_hooks
- before_put_hooks
- after_put_hooks
- before_patch_hooks
- after_patch_hooks
- before_delete_hooks
- after_delete_hooks
- after_all_hooks

As you can see, not only can we dfine the before_all_hooks and after_all_hooks like we have on the *ArrestedAPI* and *Resource*, we can also inject middleware before and after each HTTP method. Let's update our CharacterObjectEndpoint to require an admin for PUT requests.

```
def is_admin(endpoint):

    endpoint.return_error(403)

class CharacterObjectEndpoint(Endpoint, GetObjectMixin,
                              PutObjectMixin, DeleteObjectMixin):

    name = 'object'
    url = '/<string:obj_id>'
    response_handler = DBResponseHandler
    before_put_hooks = [is_admin, ]

    def get_response_handler_params(self, **params):

        params['serializer'] = character_serializer
        return params

    def get_object(self):

        obj_id = self.kwargs['obj_id']
        obj = db.session.query(Character).filter(Character.id == obj_id).one_or_none()
        if not obj:
            payload = {
                "message": "Character object not found.",
            }
            self.return_error(404, payload=payload)

        return obj

    def update_object(self, obj):
```

```
        data = self.request.data
        allowed_fields = ['name']

        for key, val in data.items():
            if key in allowed_fields:
                setattr(obj, key, val)

        db.session.add(obj)
        db.session.commit()

        return obj

    def delete_object(self, obj):

        db.session.delete(obj)
        db.session.commit()
```

Making a PUT request to `http://localhost:5000/v1/characters/1` using curl now returns a 403

4.3.4 Handling Requests and Responses

Arrested provides a flexible API for handling the data flowing into, and out from your APIs. Each endpoint can have a custom *RequestHandler* and `ResponseHandler`. This system provides support for any concievable way of processing data. Arrested also provides some out of the box integrations with popular serialization libraries, such as Kim and Marshmallow.

Request Handling

HTTP requests that process data require that a *RequestHandler* is defined on the Endpoint using the request_handler property. The default *RequestHandler* simply pulls the json data from the Flask request object, deserialises it into a dict and returns it verbatim. Let's suppose we want to apply some **very** basic validation ensuring that certain keys are present within the request payload. To do this we will implement a custom *RequestHandler* that takes a list of field names and ensures all the keys are present in the request data.

```
from arrested.handlers import RequestHandler

class ValidatingRequestHandler(RequestHandler):

    def __init__(self, endpoint, fields=None, *args, **kwargs):

        super(ValidatingRequestHandler, self).__init__(endpoint, *args, **params)
        self.fields = fields

    def handle(self, data, **kwargs):

        if self.fields and not sorted(data.keys()) == sorted(self.fields):
            payload = {
                "message": "Missing required fields",
            }
            self.endpoint.return_error(422, payload=payload)

        return super(ValidatingRequestHandler, self).handle(data, **kwargs)


class CustomEndpoint(Endpoint, GetListMixin, CreateMixin):
```

```
    many = True
    name = 'list'
    request_handler = ValidatingRequestHandler

    def get_objects(self, obj):

        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news')

    def save_object(self, obj):

        # obj will be a dict here as we're using the default RequestHandler
        return r.hmset('news', obj)

    def get_request_handler_params(self, **params):

        params = super(KimEndpoint, self).get_request_handler_params(**params)
        params['fields'] = ['field_one', 'field_two']

        return params
```

This simple examples demonstrates the flexibility the handler system offers. We can define handlers to accomodate any use case imaginable in Python. We can use the *Endpoint.get_request_handler_params* to configure the handler on an endpoint by endpoint basis.

Accessing the Request object

We've seen how to define a custom handler and how we configure it to process incoming data. So what does arrested do with all this stuff? Whenever a POST, PUT, PATCH request is made to one of your *Endpoint* arrested will instantiate the `request` object and set it on the Endpoint. This allows users to access the handler instance used to process the incoming request. An example of this in practice is the `CreateMixin.handle_post_request` method.

```
def handle_post_request(self):
    """Handle incoming POST request to an Endpoint and marshal the request data
    via the specified RequestHandler.  :meth:`.CreateMixin.save_object`. is then
    called and must be implemented by mixins implementing this interfce.

    .. seealso::
        :meth:`CreateMixin.save_object`
        :meth:`Endpoint.post`
    """
    self.request = self.get_request_handler()
    self.obj = self.request.process().data

    self.save_object(self.obj)
    return self.create_response()
```

Response Handling

Endpoints that return data will typically require that a *ResponseHandler* be defined on the Endpoint using the response_handler property. The default *ResponseHandler* simply attempts to serialize the obj passed to it using `json.dumps`. This works fine in simple cases but when we're dealing with more complex types like SQLAlchemy Models we need something a bit smarter.

Let's look at implementing a simple *ResponseHandler* that removes some fields from response data.

```
from arrested.handlers import RequestHandler

class ValidatingResponseHandler(RequestHandler):

    def __init__(self, endpoint, fields=None, *args, **kwargs):

        super(ValidatingRequestHandler, self).__init__(endpoint, *args, **params)
        self.fields = fields

    def handle(self, data, **kwargs):

        new_data = {}
        for key, value in data.items():
            if key in self.fields:
                new_data[key] = value

        return super(ValidatingResponseHandler, self).handle(new_data, **kwargs)


class CustomEndpoint(Endpoint, GetListMixin, CreateMixin):

    many = True
    name = 'list'
    response_handler = ValidatingResponseHandler

    def get_objects(self, obj):

        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        return r.hmget('news')

    def save_object(self, obj):

        # obj will be a dict here as we're using the default RequestHandler
        return r.hmset('news', obj)

    def get_response_handler_params(self, **params):

        params = super(KimEndpoint, self).get_response_handler_params(**params)
        params['fields'] = ['field_one', ]

        return params
```

Accessing the Response object

As we saw with the Request object, Arrested will store the response handler instance against a property on the Endpoint called `response`. By default a ResponseHandler is created for any request handled by an Endpoint. This means that the data generated by a RequestHandler will later be processed and returned by the provided ResponseHandler. We can see this in action in the *PutObjectMixin* mixin shown below.

```
class PutObjectMixin(HTTPMixin, ObjectMixin):
    """Base PutObjectMixins class that defines the expected API for all PutObjectMixin
    """

    def object_response(self, status=200):
        """Generic response generation for Endpoints that return a single
```

```
        serialized object.

        :param status: The HTTP status code returned with the response
        :returns: Response object
        """

        self.response = self.get_response_handler()
        self.response.process(self.obj)
        return self._response(self.response.get_response_data(), status=status)

    def put_request_response(self, status=200):
        """Pull the processed data from the response_handler and return a response.

        :param status: The HTTP status code returned with the response

        .. seealso:
            :meth:`ObjectMixin.object_response`
            :meth:`Endpoint.handle_put_request`
        """

        return self.object_response(status=status)

    def handle_put_request(self):
        """
        """
        obj = self.obj
        self.request = self.get_request_handler()
        self.request.process()

        self.update_object(obj)
        return self.put_request_response()

    def update_object(self, obj):
        """Called by :meth:`PutObjectMixin.handle_put_request` ater the incoming data␣
↪has
        been marshalled by the RequestHandler.

        :param obj: The marhsaled object from RequestHandler.
        """
        return obj
```

Handling Errors

Returning specific HTTP status codes under certain conditions is an important part of building REST APIs. Arrested provides users with a simple, and consistent way to handle generating error responses from their Endpoints. An example of this might be returning a 404 when an object is not found.

Our CharacterObjectEndpoint has already demonstrated this above in the get_object method. When we fail to find the object we're looking for from the database, we call the *Endpoint.return_error* method to have Flask abort execution of the request and immediately return an error.

We simply provide the status code we want to return along with an optional request payload that will be serialized as JSON and retured as the response body.

```
def get_object(self):

    obj_id = self.kwargs['obj_id']
```

```
    obj = db.session.query(Character).filter(Character.id == obj_id).one_or_none()
    if not obj:
        payload = {
            "message": "Character object not found.",
        }
        self.return_error(404, payload=payload)

    return obj
```

CHAPTER 5

Integrations & Recipies

5.1 Flask-SQLAlchemy

Arrested comes with built in support for working with Flask-SQLAlchemy. The SQLAlchemy mixins support automatically committing the database session when saving objects, filtering Queries by a model's primary key when fetching single objects and the removal of objects from the database when deleting. Simply put, the SQLAlchemy mixin takes care of the boring stuff when integrating Arrested with SQLAlchemy.

Note: The referrence to Flask-SQLAlchemy does not strictly mean you can't use vanilla SQLAlchemy with Arrested. See the section on providing access to the SQLAlchemy session object below for more information on custom configurations.

5.1.1 Usage

Let's refactor the Characters resource from the quickstart example application to integrate with Flask-SQLAlchemy.

```
from arrested.contrib.sql_alchemy import DBListMixin, DBCreateMixin, DBObjectMixin

class CharacterMixin(object):

    response_handler = DBResponseHandler
    model = Character

    def get_query(self):

        stmt = db.session.query(Character)
        return stmt

class CharactersIndexEndpoint(Endpoint, DBListMixin, DBCreateMixin, CharacterMixin):

    name = 'list'
```

```
    many = True

    def get_response_handler_params(self, **params):

        params['serializer'] = character_serializer
        return params


class CharacterObjectEndpoint(Endpoint, DBObjectMixin, CharacterMixin):

    name = 'object'
    url = '/<string:obj_id>'

    def get_response_handler_params(self, **params):

        params['serializer'] = character_serializer
        return params

    def update_object(self, obj):

        data = self.request.data
        allowed_fields = ['name']

        for key, val in data.items():
            if key in allowed_fields:
                setattr(obj, key, val)

        return super(CharacterObjectEndpoint, self).update_object(obj)
```

We've managed to remove quite a bit of boilerplate code by using the DBMixins provided by the SQLAlchemy contrib module. We no longer need to add the objects to the session ourselves when creating or updating objects. We also removed the code from get_object that would return a 404 if the object was not found as the DBObjectMixin, by default, does this for us. Lastly, we removed the delete_object method as the DBObjectMixin takes care of this by default.

DBListMixin

The `DBListMixin` mixin implments the standard *`GetListMixin`* interface. We'd normally be required to implement the *`GetListMixin.get_objects`* method. DBListMixin handles this method and instead requires the `get_query` method. This method should return a Query object as opposed to a scalar Python type. DBListMixin will automatically call `.all()` on the Query object for you. If for some reason you need to handle this yourself you can overwrite the `get_result` method.

```
def get_query(self):

    return db.session.query(Character).all()

def get_result(self, query):

    if isinstance(query, list):
        return query
    else:
        return query.all()
```

DBObjectMixin

The `DBListMixin` mixin implements the *`ObjectMixin`* interface. It provides handling for GET, PATCH, PUT and DELETE requests for single objects in a single mixin. When implementing the DBObjectMixin interace we'd normally be required to implement the *`get_object`* method. Instead the DBObjectMixin requires the `get_query` method. This method should return a Query object opposed to a scalar Python type. DBObjectMixin will automatically apply a WHERE clause to filter the returned Query object by the primary key of the Endpoints model.

Filtering queries by id

The automatic filtering of the returned Query object can be configured using some class level attributes exposed by the DBObjectMixin class.

```
class CharacterObjectEndpoint(Endpoint, DBObjectMixin, CharacterMixin):

    name = 'object'
    url = '/<string:slug>'
    url_id_param = 'slug'
    model_id_param = 'slug'

    def get_response_handler_params(self, **params):

        params['serializer'] = character_serializer
        return params

    def update_object(self, obj):

        data = self.request.data
        allowed_fields = ['name']

        for key, val in data.items():
            if key in allowed_fields:
                setattr(obj, key, val)

        return super(CharacterObjectEndpoint, self).update_object(obj)
```

The `model_id_param` and `url_id_param` are used in conjunction to pull a custom kwarg from our url_mapping rule and then use it to filter a "slug" field on our model.

Cutom result handling

We can also control how DBObjectMixin converts the Query obejct returned by get_query into a scalar Python type using the `get_result`. By default, DBObjectMixin will call `one_or_none()` on the Query object returned.

```
def get_result(self, query):

    # We've already handled the query, just return it..
    return query
```

5.1.2 Custom Session configuration

Arrested assumes that you're using SQLAlchemy with Flask. You can configure the DBMixins to work with other flavours of SQLAlchemy setup's. By default Arrested will attempt to pull the SQLAlchemy db session from you Flask

app's configured extensions. The `get_db_session` method simply needs to return a valid SQLAlchemy session object.

```
def get_db_session(self):
    """Returns the session configured against the Flask appliction instance.
    """
    return my_session
```

5.2 Kim

Kim is a Serialization and Marshaling framework. Arrested provides out of the box integration with Kim, providing you with the ability to serialize and deserialize complex object types.

You can read more about Kim at Read the docs. or check out the source code on GitHub.

Let's refactor the Characters resource from the quickstart example application to integrate with Kim.

5.2.1 Usage

```
from arrested.contrib.kim_arrested import KimEndpoint
from kim import Mapper, field, role


class CharacterMapper(Mapper):
    __type__ = Character

    id = field.Integer(read_only=True)
    name = field.String()
    created_at = field.Datetime(read_only=True)


class CharactersIndexEndpoint(KimEndpoint, GetListMixin, CreateMixin):

    name = 'list'
    many = True
    mapper_class = CharacterMapper

    def get_objects(self):

        characters = db.session.query(Character).all()
        return characters

    def save_object(self, obj):

        db.session.add(obj)
        db.session.commit()
        return obj


class CharacterObjectEndpoint(KimEndpoint, GetObjectMixin,
                              PutObjectMixin, DeleteObjectMixin):

    name = 'object'
    url = '/<string:obj_id>'
    mapper_class = CharacterMapper
```

```
    def get_object(self):

        obj_id = self.kwargs['obj_id']
        obj = db.session.query(Character).filter(Character.id == obj_id).one_or_none()
        if not obj:
            payload = {
                "message": "Character object not found.",
            }
            self.return_error(404, payload=payload)

        return obj

    def update_object(self, obj):

        db.session.add(obj)
        db.session.commit()

        return obj

    def delete_object(self, obj):

        db.session.delete(obj)
        db.session.commit()
```

So what's changed? Firstly we are now using the `KimEndpoint` class when defining our Endpoints. This Custom base Endpoint does the grunt work for us. It defines the custome Kim Response and Request handlers and the get_response_handler_params and get_request_handler_params methods to set them up.

This has had some impact on our CharactersIndexEndpoint and CharacterObjectEndpoint too. We no longer need to manually instantiate the Character model ourselves and we've thankfully removed that really basic validation from the update_object method. Kim now provides robust validation of the data coming into our API ensuring data is present and of the correct type.

5.3 Marshmallow

Marshmallow is a Serialization and Marshaling framework. Arrested provides out of the box integration with Marshmallow, providing you with the ability to serialize and deserialize complex object types.

You can read more about Marshmallow at Read the docs. or check out the source code on GitHub.

Let's refactor the Characters resource from the quickstart example application to integrate with Marshmallow.

5.3.1 Usage

```
from arrested.contrib.marshmallow_arrested import Marshmallow
from marshmallow import Schema, fields


class CharacterSchema(Schema):
    __type__ = Character

    id = fields.Integer(read_only=True)
    name = fields.Str(required=True)
    created_at = field.Datetime(read_only=True)
```

```
class CharactersIndexEndpoint(MarshamallowEndpoint, GetListMixin, CreateMixin):

    name = 'list'
    many = True
    schema_class = CharacterSchema

    def get_objects(self):

        characters = db.session.query(Character).all()
        return characters

    def save_object(self, obj):

        db.session.add(obj)
        db.session.commit()
        return obj


class CharacterObjectEndpoint(MarshmallowEndpoint, GetObjectMixin,
                              PutObjectMixin, DeleteObjectMixin):

    name = 'object'
    url = '/<string:obj_id>'
    schema_class = CharacterSchema

    def get_object(self):

        obj_id = self.kwargs['obj_id']
        obj = db.session.query(Character).filter(Character.id == obj_id).one_or_none()
        if not obj:
            payload = {
                "message": "Character object not found.",
            }
            self.return_error(404, payload=payload)

        return obj

    def update_object(self, obj):

        db.session.add(obj)
        db.session.commit()

        return obj

    def delete_object(self, obj):

        db.session.delete(obj)
        db.session.commit()
```

So what's changed? Firstly we are now using the `MarshmallowEndpoint` class when defining our Endpoints. This Custom base Endpoint does the grunt work for us. It defines the custom Marshmallow Response and Request handlers and the get_response_handler_params and get_request_handler_params methods to set them up.

This has had some impact on our CharactersIndexEndpoint and CharacterObjectEndpoint too. We no longer need to manually instantiate the Character model ourselves and we've thankfully removed that really basic validation from the update_object method. Marshmallow now provides robust validation of the data coming into our API ensuring data is

present and of the correct type.

CHAPTER 6

The API Documentation / Guide

Detailed class and method documentation

6.1 Developer Interface

This part of the documentation covers all the interfaces of Arrested.

6.1.1 API

class arrested.api.**ArrestedAPI**(*app=None*, *url_prefix=''*, *before_all_hooks=None*, *after_all_hooks=None*)

ArrestedAPI defines versions of your api on which *Endpoint* are registered It acts like `Flasks` Blueprint object with a few minor differences.

Constructor to create a new ArrestedAPI object.

Parameters

- **app** – Flask app object.
- **url_prefix** – Specify a url prefix for all resources attached to this API. Typically this is used for specifying the API version.
- **before_all_hooks** – A list containing funcs which will be called before every request made to any resource registered on this Api.
- **after_all_hooks** – A list containing funcs which will be called after every request made to any resource registered on this Api.

Usage:

```
app = Flask(__name__, url_prefix='/v1')
api_v1 = ArrestedAPI(app)
```

init_app(*app*)
Initialise the ArrestedAPI object by storing a pointer to a Flask app object. This method is typically used when initialisation is deferred.

Parameters **app** – Flask application object

Usage:

```
app = Flask(__name__)
ap1_v1 = ArrestedAPI()
api_v1.init_app(app)
```

register_all(*resources*)
Register each resource from an iterable.

Params resources An iterable containing *Resource* objects

Usage:

```
characters_resource = Resource(
    'characters', __name__, url_prefix='/characters'
)
planets_resource = Resource('planets', __name__, url_prefix='/planets')
api_v1 = ArrestedAPI(prefix='/v1')
api_v1.register_all([characters_resource, planets_resource])
```

register_resource(*resource*, *defer=False*)
Register a *Resource* blueprint object against the Flask app object.

Parameters

- **resource** – *Resource* or `flask.Blueprint` object.
- **defer** – Optionally specify that registering this resource should be deferred. This option is useful when users are creating their Flask app instance via a factory.

Deferred resource registration

Resources can optionally be registered in a deferred manner. Simply pass *defer=True* to *ArrestedAPI.register_resource* to attach the resource to the API without calling register_blueprint.

This is useful when you're using the factory pattern for creating your Flask app object as demonstrated below. Deferred resource will not be registered until the ArrestedAPI instance is initialised with the Flask app object.

Usage:

```
api_v1 = ArrestedAPI(prefix='/v1')
characters_resource = Resource(
    'characters', __name__, url_prefix='/characters'
)
ap1_v1.register_resource(characters_resource, defer=True)

def create_app():

    app = Flask(__name__)
    api_v1.init_app(app) # deferred resources are now registered.
```

6.1.2 Resource

class arrested.resource.**Resource**(*name*, *import_name*, *api=None*, *before_all_hooks=None*, *after_all_hooks=None*, **args*, ***kwargs*)

Resource extends Flasks existing Blueprint object and provides some utilty methods that make registering `Endpoint` simpler.

Construct a new Reosurce blueprint. In addition to the normal Blueprint options, Resource accepts kwargs to set request middleware.

Parameters

- **api** – API instance being the resource is being registered against.
- **before_all_hooks** – A list of middleware functions that will be applied before every request.
- **after_all_hooks** – A list of middleware functions that will be applied after every request.

Middleware

Arrested supports applying request middleware at every level of the application stack. Resource middleware will be applied for every Endpoint registered on that Resource.

```
character_resource = Resource(
    'characters', __name__,
    url_prefix='/characters',
    before_all_hooks=[log_request]
)
```

Arrested request middleware works differently from the way Flask middleware does. Middleware registered at the Resource and API level are consumed by the `arrested.Endpoint.dispatch_request` rather than being fired via the Flask app instance. This is so we can pass the instance of the endpoint handling the request to each piece of middleware registered on the API or Resource.

```
def api_key_required(endpoint):

    if request.args.get('api_key', None) is None:
        endpoint.return_error(422)
    else:
        get_client(request.args['api_key'])

character_resource = Resource(
    'characters', __name__,
    url_prefix='/characters',
    before_all_hooks=[api_key_required]
)
```

Please note, Flask's normal App and Blueprint middleware can still be used as normal, it just doesn't recieve the instance of the endpoint registered to handle the request.

add_endpoint(*endpoint*)

Register an *Endpoint* aginst this resource.

Parameters **endpoint** – *Endpoint* API Endpoint class

Usage:

```
foo_resource = Resource('example', __name__)
class MyEndpoint(Endpoint):
```

```
    url = '/example'
    name = 'myendpoint'

foo_resource.add_endpoint(MyEndpoint)
```

init_api(*api*)
Registered the instance of *ArrestedAPI* the *Resource* is being registered against.

Parameters **api** – API instance being the resource is being registered against.

6.1.3 Endpoint

class arrested.endpoint.**Endpoint**
The Endpoint class represents the HTTP methods that can be called against an Endpoint inside of a particular resource.

after_all_hooks = []
A list of functions called after all requests are dispatched

after_delete_hooks = []
A list of functions called after DELETE requests are dispatched

after_get_hooks = []
A list of functions called after GET requests are dispatched

after_patch_hooks = []
A list of functions called after PATCH requests are dispatched

after_post_hooks = []
A list of functions called after POST requests are dispatched

after_put_hooks = []
A list of functions called after PUT requests are dispatched

as_view(*name*, **class_args*, ***class_kwargs*)
Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the *dispatch_request* method on it.

The arguments passed to *as_view* are forwarded to the constructor of the class.

before_all_hooks = []
A list of functions called before any specific request handler methods are called

before_delete_hooks = []
A list of functions called before DELETE requests are dispatched

before_get_hooks = []
A list of functions called before GET requests are dispatched

before_patch_hooks = []
A list of functions called before PATCH requests are dispatched

before_post_hooks = []
A list of functions called before POST requests are dispatched

before_put_hooks = []
A list of functions called before PUT requests are dispatched

delete(**args*, ***kwargs*)
Handle Incoming DELETE requests and dispatch to handle_delete_request method.

dispatch_request(**args*, ***kwargs*)
Dispatch the incoming HTTP request to the appropriate handler.

get(**args*, ***kwargs*)
Handle Incoming GET requests and dispatch to handle_get_request method.

classmethod get_name()
Returns the user provided name or the lower() class name for use when registering the Endpoint with a `Resource`.

Returns registration name for this Endpoint.

Return type string

get_request_handler()
Return the Endpoints defined *`Endpoint.request_handler`*.

Returns A instance of the Endpoint specified `RequestHandler`.

Return type `RequestHandler`

get_request_handler_params(***params*)
Return a dictionary of options that are passed to the specified `RequestHandler`.

Returns Dictionary of `RequestHandler` config options.

Return type dict

get_response_handler()
Return the Endpoints defined *`Endpoint.response_handler`*.

Returns A instance of the Endpoint specified `ResonseHandler`.

Return type `ResponseHandler`

get_response_handler_params(***params*)
Return a dictionary of options that are passed to the specified `ResponseHandler`.

Returns Dictionary of `ResponseHandler` config options.

Return type dict

make_response(*rv*, *status=200*, *headers=None*, *mime='application/json'*)
Create a response object using the `flask.Response` class.

Parameters

- **rv** – Response value. If the value is not an instance of `werkzeug.wrappers.Response` it will be converted into a Response object.
- **status** – specify the HTTP status code for this response.
- **mime** – Specify the mimetype for this request.
- **headers** – Specify dict of headers for the response.

methods = ['GET', 'POST', 'PUT', 'PATCH', 'DELETE']
list containing the permitted HTTP methods this endpoint accepts

name = None
The name used to register this endpoint with Flask's url_map

patch(**args*, ***kwargs*)
Handle Incoming PATCH requests and dispatch to handle_patch_request method.

post (**args*, ***kwargs*)
Handle Incoming POST requests and dispatch to handle_post_request method.

put (**args*, ***kwargs*)
Handle Incoming PUT requests and dispatch to handle_put_request method.

request_handler
A *RequestHandler* class

alias of `RequestHandler`

response_handler
A *ResponseHandler* class

alias of `ResponseHandler`

return_error (*status*, *payload=None*)
Error handler called by request handlers when an error occurs and the request should be aborted.

Usage:

```
def handle_post_request(self, *args, **kwargs):

    self.request_handler = self.get_request_handler()
    try:
        self.request_handler.process(self.get_data())
    except SomeException as e:
        self.return_error(400, payload=self.request_handler.errors)

    return self.return_create_response()
```

url = ''
The URL this endpoint is mapped against. This will build on top of any url_prefix defined at the API and Resource level

6.1.4 Mixins

class arrested.mixins.**GetListMixin**
Base ListMixin class that defines the expected API for all ListMixins

get_objects()

handle_get_request()
Handle incoming GET request to an Endpoint and return an array of results by calling *GetListMixin.get_objects*.

See also:

GetListMixin.get_objects Endpoint.get

list_response (*status=200*)
Pull the processed data from the response_handler and return a response.

Parameters **status** – The HTTP status code returned with the response

class arrested.mixins.**CreateMixin**
Base CreateMixin class that defines the expected API for all CreateMixins

create_response (*status=201*)
Generate a Response object for a POST request. By default, the newly created object will be passed to the specified ResponseHandler and will be serialized as the response body.

handle_post_request()
Handle incoming POST request to an Endpoint and marshal the request data via the specified RequestHandler. *CreateMixin.save_object*. is then called and must be implemented by mixins implementing this interfce.

See also:

CreateMixin.save_object Endpoint.post

save_object(*obj*)
Called by *CreateMixin.handle_post_request* ater the incoming data has been marshalled by the RequestHandler.

Parameters **obj** – The marhsaled object from RequestHandler.

class arrested.mixins.**ObjectMixin**
Mixin that provides an interface for working with single data objects

get_object()
Called by *GetObjectMixin.handle_get_request*. Concrete classes should implement this method and return object typically by id.

Raises NotImplementedError

obj
Returns the value of *ObjectMixin.get_object* and sets a private property called _obj. This property ensures the logic around allow_none is enforced across Endpoints using the Object interface.

Raises werkzeug.exceptions.BadRequest

Returns The result of :meth:ObjectMixin.get_object‘

object_response(*status=200*)
Generic response generation for Endpoints that return a single serialized object.

Parameters **status** – The HTTP status code returned with the response

Returns Response object

class arrested.mixins.**GetObjectMixin**
Base GetObjectMixins class that defines the expected API for all GetObjectMixins

handle_get_request()
Handle incoming GET request to an Endpoint and return a single object by calling GetListMixin.get_object.

See also:

GetListMixin.get_objects Endpoint.get

class arrested.mixins.**PutObjectMixin**
Base PutObjectMixins class that defines the expected API for all PutObjectMixin

handle_put_request()

put_request_response(*status=200*)
Pull the processed data from the response_handler and return a response.

Parameters **status** – The HTTP status code returned with the response

update_object(*obj*)
Called by *PutObjectMixin.handle_put_request* ater the incoming data has been marshalled by the RequestHandler.

Parameters **obj** – The marhsaled object from RequestHandler.

class arrested.mixins.**PatchObjectMixin**
Base PatchObjectMixin class that defines the expected API for all PatchObjectMixin

handle_patch_request()

patch_object(*obj*)
Called by *PatchObjectMixin.handle_patch_request* ater the incoming data has been marshalled by the RequestHandler.

Parameters **obj** – The marhsaled object from RequestHandler.

patch_request_response(*status=200*)
Pull the processed data from the response_handler and return a response.

Parameters **status** – The HTTP status code returned with the response

class arrested.mixins.**DeleteObjectMixin**
Base DeletehObjecttMixin class that defines the expected API for all DeletehObjecttMixins.

delete_object(*obj*)
Called by *DeleteObjectMixin.handle_delete_request*.

Parameters **obj** – The marhsaled object being deleted.

delete_request_response(*status=204*)
Pull the processed data from the response_handler and return a response.

Parameters **status** – The HTTP status code returned with the response

handle_delete_request()

6.1.5 Handlers

class arrested.handlers.**Handler**(*endpoint*, *payload_key='payload'*, ***params*)

handle(*data*, ***kwargs*)
Invoke the handler to process the provided data. Concrete classes should override this method to provide specific capabilties for the choosen method of marshaling and serializing data.

Parameters **data** – The data to be processed by the Handler.

Returns The data processed by the Handler

Return type mixed

Here's an example of a RequestHandler integrating with the Kim library.

```
def handle(self, data):
    try:
        if self.many:
            return self.mapper.many(raw=self.raw,
                                    **self.mapper_kwargs)
↪         .marshal(data, role=self.role)
        else:
            return self.mapper(data=data, raw=self.raw,
                               **self.mapper_kwargs)
↪    .marshal(role=self.role)
    except MappingInvalid as e:
        self.errors = e.errors
```

handle_error(*status*, *errors*)
Generate an error response returning a payload containing validation errors that ocurred during this request.

Parameters

- **status** – HTTP status code
- **errors** – JSON serializable object

process(*data=None*, ***kwargs*)
Process the provided data and invoke *Handler.handle* method for this Handler class.

Params data The data being processed.

Returns self

Return type *Handler*

```
def post(self, *args, **kwargs):
    self.request = self.get_request_handler()
    self.request.process(self.get_data())
    return self.get_create_response()
```

class arrested.handlers.**JSONResponseMixin**
Provides handling for serializing the response data as a JSON string.

get_response_data()
serialzie the response data and payload_key as a JSON string.

Returns JSON serialized string

Return type bytes

class arrested.handlers.**JSONRequestMixin**
Provides handling for fetching JSON data from the FLask request object.

get_request_data()
Pull JSON from the Flask request object.

Returns Deserialized JSON data.

Return type mixed

Raises flask.exceptions.JSONBadRequest

class arrested.handlers.**RequestHandler**(*endpoint*, *payload_key='payload'*, ***params*)
Basic default RequestHandler that expects the will pull JSON from the Flask request object and return it.

process(*data=None*)
Fetch incoming data from the Flask request object when no data is supplied to the process method. By default, the RequestHandler expects the incoming data to be sent as JSON.

class arrested.handlers.**ResponseHandler**(*endpoint*, *payload_key='payload'*, ***params*)
Basic default ResponseHanlder that expects the data passed to it to be JSON serializable without any modifications.

6.1.6 Exceptions

exception arrested.exceptions.**ArrestedException**

Python Module Index

a

Index

I

J

L

M

N

O

P

R

S

U