Audrey Documentation

Release 0.0.1

Sam Brauer

Sep 27, 2017

Contents

Audrey is a minimal framework for creating Pyramid applications that use MongoDB for persistence, ElasticSearch for full text search (optional), traversal for resource/view lookup, and colander for schema declaration and user input validation.

Audrey also provides views that implement a RESTful API. In an attempt to satisfy the hypermedia constraint (HATEOAS), GET responses use the HAL+JSON mediatype. In a further attempt to be self-describing, links to JSON Schema documents (generated automatically from your types' colander schemas) are provided which describe the bodies for POST and PUT requests. Audrey doesn't provide any HTML views, but it does include HAL-Browser which can be used to explore the RESTful API. (Please be aware that the included API is tightly coupled to your resource models. Such an API may be handy for prototypes or even in cases where you control all client use, but it is **definitely not recommended for use by third-parties**. Creating your own custom versioned API is highly recommended.)

My goal is to keep Audrey otherwise unopinionated. For example, Audrey intentionally does nothing regarding authentication, authorization, permissions, etc. A developer building on Audrey can make those decisions as appropriate for their app and implement them using standard Pyramid facilities.

Warning: Audrey is a pet project serving as a playground to explore some ideas. Too soon to tell whether it will mature or not.

CHAPTER 1

Introduction

Let's say you just installed Audrey and used its starter scaffold to create a new project (as described in *Creating a new project*). You'd have two example object types (`Person` and `Post`) to play with. (As you read this section, if you find yourself wondering things like how the types and their schemas are defined, you may want to jump ahead to the *Resource Modelling* section.)

Let's take a look around in a pshell session:

```
$ pshell development.ini#main
Python 2.7.3 (default, Nov 13 2012, 15:00:33)
[GCC 4.4.5] on linux2
Type "help" for more information.

Environment:
  app          The WSGI application.
  registry     Active Pyramid registry.
  request      Active request object.
  root         Root of the default resource tree.
  root_factory Default root factory used to create `root`.

>>> root
<myproject.resources.Root object at 0x9d35a6c>
>>> root.get_collection_names()
['people', 'posts']
```

OK. So we have a Root object with two collections named "people" and "posts". Let's check out one of those:

```
>>> people = root['people']
>>> people
<myproject.resources.People object at 0xa26c04c>
>>> people.get_children()
[]
```

Look's like there aren't any people yet. So let's create one:

```
>>> from myproject import resources
>>> person = resources.Person(request)
>>> print person
{'_created': None,
 '_etag': None,
 '_id': None,
 '_modified': None,
 'firstname': None,
 'lastname': None,
 'photo': None}
```

Kinda boring. But let's see what would happen if we tried to save it (by adding it to the `people` collection):

```
>>> people.add_child(person)
... traceback omitted ...
Invalid: {'firstname': u'Required', 'lastname': u'Required'}
```

That's a `colander.Invalid` exception letting us know that schema validation failed. Let's set the required attributes and try again:

```
>>> person.firstname = 'Audrey'
>>> person.lastname = 'Horne'
>>> people.add_child(person)
>>> print person
{'_created': datetime.datetime(2012, 12, 24, 1, 52, 45, 281718, tzinfo=<UTC>),
 '_etag': '52779a9953bd01defd439bd29874c3d4',
 '_id': ObjectId('50d7b56dbf90af0e96bc8433'),
 '_modified': datetime.datetime(2012, 12, 24, 1, 52, 45, 281718, tzinfo=<UTC>),
 'firstname': 'Audrey',
 'lastname': 'Horne',
 'photo': None}
```

The object has been persisted in MongoDB and now has an ObjectId, creation and modification timestamps and an Etag. (It was also indexed in ElasticSearch.) Let's check the children of the `People` collection again:

```
>>> people.get_children()
[<myproject.resources.Person object at 0xa26cbac>]
```

As sort of an aside, we can traverse to the new Person object by the string version of its ID like this:

```
>>> root['people']['50d7b56dbf90af0e96bc8433']
<myproject.resources.Person object at 0xa1f4d6c>
>>> person.__name__
'50d7b56dbf90af0e96bc8433'
>>> person.__parent__
<myproject.resources.People object at 0xa26c04c>
```

Note: Using the ID as the __name__ is the behavior of the base Audrey `Object` and `Collection` types. There exist subclasses `NamedObject` and `NamingCollection` that allow for explicit control over naming. Whether you use one or the other depends on your use case. For this introduction, I opted to keep it minimal and use the base classes.

Let's add a couple more Person objects to make things a little more interesting. We can pass kwargs to the object constructor to initialize attributes:

```
>>> people.add_child(resources.Person(request, firstname='Laura', lastname='Palmer'))
>>> people.add_child(resources.Person(request, firstname='Dale', lastname='Cooper'))
>>> [child.get_title() for child in people.get_children()]
[u'Dale Cooper', u'Audrey Horne', u'Laura Palmer']
```

The order of the children is arbitrary. Let's explicitly sort them:

```
>>> [child.get_title() for child in people.get_children(sort=[('_created',1)])]
[u'Audrey Horne', u'Dale Cooper', u'Laura Palmer']
```

Did you notice the `photo` attribute earlier? Let's set a photo for Dale. First let's retrieve his object:

```
>>> obj = people.get_child({'firstname':'Dale'})
>>> print obj
{'_created': datetime.datetime(2012, 12, 24, 2, 10, 14, 856000, tzinfo=<UTC>),
 '_etag': u'a8ee673c5490be625bd720375add252f',
 '_id': ObjectId('50d7b986bf90af0e96bc8434'),
 '_modified': datetime.datetime(2012, 12, 24, 2, 10, 14, 856000, tzinfo=<UTC>),
 'firstname': u'Dale',
 'lastname': u'Cooper',
 'photo': None}
```

Now we'll open a file, add it to Audrey's GridFS, update the Person and then save it:

```
>>> with open("dale-cooper.jpg") as f:
...     obj.photo = root.create_gridfs_file(f, "dale-cooper.jpg", "image/jpeg")
>>> obj.save()
>>> print obj
{'_created': datetime.datetime(2012, 12, 24, 2, 10, 14, 856000, tzinfo=<UTC>),
 '_etag': '080b9d79d888e5d6714acc8cfb07d6ae',
 '_id': ObjectId('50d7b986bf90af0e96bc8434'),
 '_modified': datetime.datetime(2013, 1, 3, 1, 7, 31, 134749, tzinfo=<UTC>),
 'firstname': u'Dale',
 'lastname': u'Cooper',
 'photo': <audrey.resources.file.File object at 0xaa2190c>}
```

`photo` is an instance of *`audrey.resources.file.File`*. This is simply a wrapper around the ObjectId of a GridFS file. To access the GridFS file (which can be read like a normal Python file and also has a few extra attributes like `name` and `content_type`), call `get_gridfs_file()`:

```
>>> gf = obj.photo.get_gridfs_file(request)
>>> gf.name
u'dale-cooper.jpg'
>>> gf.length
66953
>>> gf.content_type
u'image/jpeg'
```

We've covered creating and updating objects. Now let's delete one:

```
>>> obj = people.get_child({'firstname': 'Laura'})
>>> people.delete_child(obj)
>>> [child.get_title() for child in people.get_children()]
[u'Dale Cooper', u'Audrey Horne']
```

Note: `Collection` also has methods `delete_child_by_id()` and `delete_child_by_name()`. This introduction doesn't try to demonstrate every method and parameter. Refer to the *API* section for more.

Now let's switch our focus to the web api. (If you're running locally, you can explore the api with HAL-Browser by visiting http://127.0.0.1:6543/hal-browser/ in your web browser.) For our current purposes, I'll use curl and Python's super-handy json.tool:

```
$ curl http://127.0.0.1:6543/ | python -mjson.tool
{
    "_links": {
        "audrey:upload": {
            "href": "http://127.0.0.1:6543/@@upload"
        },
        "curie": {
            "href": "http://127.0.0.1:6543/relations/{rel}",
            "name": "audrey",
            "templated": true
        },
        "item": [
            {
                "href": "http://127.0.0.1:6543/people/{?sort}",
                "name": "people",
                "templated": true
            },
            {
                "href": "http://127.0.0.1:6543/posts/{?sort}",
                "name": "posts",
                "templated": true
            }
        ],
        "search": {
            "href": "http://127.0.0.1:6543/@@search?q={q}{&sort}{&collection*}",
            "templated": true
        },
        "self": {
            "href": "http://127.0.0.1:6543/"
        }
    }
}
```

Note: These are just the default views that Audrey provides. You can override and reconfigure to suit your needs, or ignore them entirely and create your own views from scratch.

Note: This view-related documentation needs to be updated to reflect the current state of the views. Everything described here will still work, but the responses may differ slightly. For example, listing views have new "embed" and "fields" options which didn't appear in the templated urls described here. Try out http://127.0.0.1:6543/people?embed=1 or http://127.0.0.1:6543/people?embed=1&fields=firstname,lastname

This is a HAL+JSON document representing the root. Since the root has no state of its own, the document just has a number of links keyed by link relation ("rel") names. Besides "self" which is obligatory for HAL, Audrey tries to stick to relations from the IANA list.

Here we see "item" used to list the children of root (the "people" and "posts" collections). These urls are templated, in this case indicating that you may use an optional "sort" parameter. In a moment, we'll follow one of these links.

There's also a link to a "search" endpoint (again with a URL template) and another to the "upload" endpoint. Since there was no IANA rel that seemed suitable for the upload endpoint (which as you may have guessed is a factory for uploading files into the system), Audrey uses a namespaced URI. Applying the "curie" template, "audrey:upload" expands to "http://127.0.0.1:6543/relations/upload"; visiting that url returns some HTML documentation of the endpoint including the expected request and response details.

Now let's GET the "people" collection using the "sort" parameter to sort by creation time:

```
$ curl http://127.0.0.1:6543/people?sort=_created | python -mjson.tool
{
    "_factory": {
        "method": "POST",
        "schemas": [
            "person"
        ]
    },
    "_links": {
        "audrey:schema": [
            {
                "href": "http://127.0.0.1:6543/people/@@schema/person",
                "name": "person"
            }
        ],
        "collection": {
            "href": "http://127.0.0.1:6543/"
        },
        "curie": {
            "href": "http://127.0.0.1:6543/relations/{rel}",
            "name": "audrey",
            "templated": true
        },
        "item": [
            {
                "href": "http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/",
                "name": "50d7b56dbf90af0e96bc8433",
                "title": "Audrey Horne"
            },
            {
                "href": "http://127.0.0.1:6543/people/50d7b986bf90af0e96bc8434/",
                "name": "50d7b986bf90af0e96bc8434",
                "title": "Dale Cooper"
            }
        ],
        "self": {
            "href": "http://127.0.0.1:6543/people/?sort=_created"
        }
    },
    "_summary": {
        "batch": 1,
        "per_batch": 20,
        "sort": "_created",
        "total_batches": 1,
        "total_items": 2
    }
}
```

The Collection view has some similarities with the Root view. Again we see the obligatory "self" link and a list of "item" links (this time the items are the two `Person` instances we created earlier). The "collection" rel is used to indicate a link to the container of the current resource, which in this case is the root. Finally there's a custom namespaced "schema" rel. As the documentation at http://127.0.0.1:6543/relations/schema explains, the "schema" rel is a list of links to JSON Schema documents; there's one such link for each object type that can be created in the current Collection.

We also see two custom properties: "_factory" and "_summary".

The first identifies the HTTP method to be used to create new resources inside the collection. Here it's POST since People is a base Collection and assigns names automatically. If it was a NamingCollection, the method would be PUT indicating that clients should specify new resource names by doing a PUT to a new url (such as "/people/harry-truman").

The "_summary" property contains some metadata about the current item listing. Here we see that there are 2 items total. Since the batch size is 20, there's only one batch. If there were more than 20 people, the "item" link array would only include a batch of up to 20 and there may be links with the rel "next" and/or "prev" with the urls for the next and previous batches (as appropriate).

Now let's follow the first "item" link:

```
$ curl http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/ | python -mjson.tool
{
    "_created": "2012-12-24T01:52:45.281000+00:00",
    "_etag": "52779a9953bd01defd439bd29874c3d4",
    "_id": {
        "ObjectId": "50d7b56dbf90af0e96bc8433"
    },
    "_links": {
        "audrey:file": [],
        "audrey:reference": [],
        "collection": {
            "href": "http://127.0.0.1:6543/people/"
        },
        "curie": {
            "href": "http://127.0.0.1:6543/relations/{rel}",
            "name": "audrey",
            "templated": true
        },
        "describedby": {
            "href": "http://127.0.0.1:6543/people/@@schema/person"
        },
        "self": {
            "href": "http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/"
        }
    },
    "_modified": "2012-12-24T01:52:45.281000+00:00",
    "_object_type": "person",
    "_title": "Audrey Horne",
    "firstname": "Audrey",
    "lastname": "Horne",
    "photo": null
}
```

Finally, something with some state data; here we see the schema properties "firstname", "lastname" and "photo", as well as various metadata properties which I've used the convention of starting with an underscore. Now let's look at the ubiquitous links.

There's "self" of course. The "collection" link refers to the current object's container. The "describedby" link refers to a JSON Schema for the object. Finally there are two custom rels "file" and "reference".

The "file" rel is used to indicate a list of links to (you guessed it) files referenced by this resource object. In this case, if "photo" wasn't null there would be a link to the photo file. (Keep reading and we'll upload a photo file and update this person to refer to it.)

The "reference" rel is used to indicate a list of links to other object resources referenced by this one. The `Person` type doesn't have any reference attributes in its schema, so this will always be an empty list for this class.

Now let's demonstrate POSTing a new `Person`:

```
$ curl -i -XPOST http://127.0.0.1:6543/people/ -d '{
      "_object_type": "person",
      "firstname": "Shelly",
      "lastname": "Johnson"
  }'

HTTP/1.1 201 Created
Content-Length: 2
Content-Type: application/json; charset=UTF-8
Date: Mon, 24 Dec 2012 18:25:35 GMT
Location: http://127.0.0.1:6543/people/50d89e1fbf90af0d7169df5d/
Server: waitress

{}
```

Cool... Audrey responds with the `201 Created` success status and "Location" header with the URL of the new resource.

You might wonder what would happen if we tried to POST an invalid request. First let's try POSTing an empty JSON document:

```
$ curl -i -XPOST http://127.0.0.1:6543/people/ -d '{}'
HTTP/1.1 400 Bad Request
Content-Length: 45
Content-Type: application/json; charset=UTF-8
Date: Mon, 24 Dec 2012 18:27:34 GMT
Server: waitress

{"error": "Request is missing _object_type."}
```

Uh oh... we got `400 Bad Request` and an error message in the body with the reason. So now let's POST a document that just contains an "_object_type":

```
curl -i -XPOST http://127.0.0.1:6543/people/ -d '{"_object_type": "person"}'
HTTP/1.1 400 Bad Request
Content-Length: 92
Content-Type: application/json; charset=UTF-8
Date: Mon, 24 Dec 2012 18:27:57 GMT
Server: waitress

{"errors": {"lastname": "Required", "firstname": "Required"}, "error": "Validation␣
↪failed."}
```

Another 400 error and another "error" message. Since this one's a validation error, the JSON document in the response also includes an "errors" key with the field-specific errors (courtesy of colander).

Now let's upload a photo:

```
$ curl -F file=@audrey.jpg http://127.0.0.1:6543/@@upload
{"file": [{"FileId": "50d8a64bbf90af0d7169df5e"}]}
```

The server creates a GridFS file in MongoDB for each file from the request and responds with a JSON document containing a list of the file ObjectIds for each parameter name from the request.

Let's update Audrey Horne's record with the new photo file:

```
$ curl -i -XPUT http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/ -d '{
      "_object_type": "person",
      "firstname": "Audrey",
      "lastname": "Horne",
      "photo":  {"FileId": "50d8a64bbf90af0d7169df5e"}
  }'
HTTP/1.1 412 Precondition Failed
Content-Length: 75
Content-Type: application/json; charset=UTF-8
Date: Mon, 24 Dec 2012 20:04:37 GMT
Server: waitress

{"error": "Requests must supply If-Unmodified-Since and If-Match headers."}
```

What's going on here? The views implement optimistic concurrency control in an effort to avoid silent data loss. PUT requests to update an existing resource and DELETE requests to remove an existing resource must include "If-Unmodified-Since" and "If-Match" headers whose values must match the "Last-Modified" and "Etag" headers from the response to a GET of that same resource. Let's examine the response headers to get those two values:

```
$ curl -i http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/
HTTP/1.1 200 OK
Content-Length: 660
Content-Type: application/hal+json; charset=UTF-8
Date: Mon, 24 Dec 2012 20:13:42 GMT
Etag: "52779a9953bd01defd439bd29874c3d4"
Last-Modified: Mon, 24 Dec 2012 01:52:45 GMT
Server: waitress

{"_links": {"audrey:file": [], "self": {"href": "http://127.0.0.1:6543/people/
↪50d7b56dbf90af0e96bc8433/"}, "collection": {"href": "http://127.0.0.1:6543/people/"}
↪, "curie": {"href": "http://127.0.0.1:6543/relations/{rel}", "name": "audrey",
↪"templated": true}, "audrey:reference": [], "describedby": {"href": "http://127.0.0.
↪1:6543/people/@@schema/person"}}, "photo": null, "firstname": "Audrey", "lastname":
↪"Horne", "_modified": "2012-12-24T01:52:45.281000+00:00", "_created": "2012-12-
↪24T01:52:45.281000+00:00", "_title": "Audrey Horne", "_id": {"ObjectId":
↪"50d7b56dbf90af0e96bc8433"}, "_etag": "52779a9953bd01defd439bd29874c3d4", "_object_
↪type": "person"}
```

Now let's try that PUT again with the two headers for OCC:

```
$ curl -i -H 'If-Unmodified-Since:Mon, 24 Dec 2012 01:52:45 GMT' \
-H 'If-Match:"52779a9953bd01defd439bd29874c3d4"' \
-XPUT http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/ -d '{
    "_object_type": "person",
    "firstname": "Audrey",
    "lastname": "Horne",
    "photo":  {"FileId": "50d8a64bbf90af0d7169df5e"}
}'
HTTP/1.1 204 No Content
Content-Length: 0
Location: http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/
Content-Type: application/json; charset=UTF-8
Date: Mon, 24 Dec 2012 20:19:23 GMT
Server: waitress
```

Success! Let's confirm the change by doing another GET:

```
$ curl http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/ | python -mjson.tool
{
    "_created": "2012-12-24T01:52:45.281000+00:00",
    "_etag": "3c418f678d1cb636fca4cadc599bf725",
    "_id": {
        "ObjectId": "50d7b56dbf90af0e96bc8433"
    },
    "_links": {
        "audrey:file": [
            {
                "href": "http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/
↪@@download/50d8a64bbf90af0d7169df5e",
                "name": "50d8a64bbf90af0d7169df5e",
                "type": "image/jpeg"
            }
        ],
        "audrey:reference": [],
        "collection": {
            "href": "http://127.0.0.1:6543/people/"
        },
        "curie": {
            "href": "http://127.0.0.1:6543/relations/{rel}",
            "name": "audrey",
            "templated": true
        },
        "describedby": {
            "href": "http://127.0.0.1:6543/people/@@schema/person"
        },
        "self": {
            "href": "http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/"
        }
    },
    "_modified": "2012-12-24T20:19:23.660000+00:00",
    "_object_type": "person",
    "_title": "Audrey Horne",
    "firstname": "Audrey",
    "lastname": "Horne",
    "photo": {
        "FileId": "50d8a64bbf90af0d7169df5e"
    }
}
```

The "photo" is no longer null and the list of "file" links now contains one item with type="image/jpeg" and name="50d8a64bbf90af0d7169df5e". A client could match up that name with the value of the photo FileId.

Try viewing the photo by hitting http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/@@download/50d8a64bbf90af0d7169df5e

You could also traverse to the `photo` attribute like so: http://127.0.0.1:6543/people/50d7b56dbf90af0e96bc8433/photo

As our final stop before ending this introduction, let's try out the most basic usage of the search api. We'll do a search for "dale":

```
$ curl http://127.0.0.1:6543/@@search?q=dale | python -mjson.tool
{
    "_links": {
        "item": [
            {
                "href": "http://127.0.0.1:6543/people/50d7b986bf90af0e96bc8434/",
                "name": "people:50d7b986bf90af0e96bc8434",
                "title": "Dale Cooper"
            }
        ],
        "self": {
            "href": "http://127.0.0.1:6543/@@search?q=dale"
        }
    },
    "_summary": {
        "batch": 1,
        "collections": [],
        "per_batch": 20,
        "q": "dale",
        "sort": null,
        "total_batches": 1,
        "total_items": 1
    }
}
```

The search found Dale’s `Person` object. As you might guess, if there were lots of results they would be batched with “next” and “prev” links.

Well that wraps up this introduction. It didn’t cover everything, but hopefully it provided a sufficient taste.

CHAPTER 2

Resource Modelling

Audrey provides some base resource classes for a developer to subclass to model the objects specific to their application. The main classes are:

1. *audrey.resources.object.Object* - This is the fundamental building block of an Audrey application. Objects have methods to save/load/remove themselves in MongoDB and index/reindex/unindex themselves in ElasticSearch.

2. *audrey.resources.collection.Collection* - Collections are sets of Objects. They correspond to MongoDB collections and have various methods to access and manipulate their child objects.

3. *audrey.resources.root.Root* - Root is the container of Collections and represents the root of your app. It also provides various "global" services (such as search, cross-collection references and file uploads/downloads).

As the application developer, you define your Object classes (using colander to define the schema for each class), your Collection classes (specifying which Object classes can be created in each Collection), and a Root class (specifying the list of Collections). Audrey then provides what I hope is a comfortable and Pythonic interface that handles the boring, repetitve yet error-prone details of interacting with MongoDB and ElasticSearch, validating your schemas, etc.

Note: The base `Object` and `Collection` classes don't allow explicit control of the `__name__` attribute used for traversal. For cases where you need such control, use the *audrey.resources.object.NamedObject* and *audrey.resources.collection.NamingCollection* classes instead.

After you create a new project using the `audrey` scaffold (as described in *Creating a new project*), you'll have a couple of example Objects and Collections defined in the file `resources.py` inside your package directory (which will be the same name as your project name, but in lowercase). You'll of course want to replace these examples with your own Objects and Collections, and may even want to split the single file up into a `resources` sub-package.

Let's take a close look at the example `resources.py` file and see how it subclasses the base Audrey classes. The file should have content similar to the following:

```python
import audrey
import colander

# The following are just some example resource classes to get
# you started using Audrey.

class Person(audrey.resources.object.Object):
    _object_type = "person"

    _schema = colander.SchemaNode(colander.Mapping())
    _schema.add(colander.SchemaNode(colander.String(), name='firstname'))
    _schema.add(colander.SchemaNode(colander.String(), name='lastname'))
    _schema.add(colander.SchemaNode(audrey.types.File(), name='photo',
                default=None, missing=None))

    def get_title(self):
        parts = []
        for att in ('firstname', 'lastname'):
            val = getattr(self, att, '')
            if val:
                parts.append(val)
        if parts:
            return " ".join(parts)
        else:
            return "Untitled"

class People(audrey.resources.collection.Collection):
    _collection_name = 'people'
    _object_classes = (Person,)

# A deferred schema binding.  Used to populate the missing attribute
# of Post.dateline at runtime.
@colander.deferred
def deferred_datetime_now(node, kw):
    return audrey.dateutil.utcnow(zero_seconds=True)

class Post(audrey.resources.object.Object):
    _object_type = "post"

    _schema = colander.SchemaNode(colander.Mapping())
    _schema.add(colander.SchemaNode(colander.String(), name='title'))
    _schema.add(colander.SchemaNode(colander.DateTime(), name='dateline',
                missing=deferred_datetime_now))
    _schema.add(colander.SchemaNode(colander.String(), name='body',
                is_html=True))
    _schema.add(colander.SchemaNode(
                audrey.types.Reference(collection='people'),
                name='author', default=None, missing=None))

    @classmethod
    def get_class_schema(cls, request=None):
        return cls._schema.bind()

    def get_title(self):
        return getattr(self, 'title', None) or 'Untitled'

class Posts(audrey.resources.collection.Collection):
    _collection_name = 'posts'
    _object_classes = (Post,)

class Root(audrey.resources.root.Root):
    _collection_classes = (People, Posts, )
```

```
63
64 def root_factory(request): # pragma: no cover
65     return Root(request)
```

Starting at line 7, a `Person` class is defined that subclasses *`audrey.resources.object.Object`*.

At line 8, the class attribute `_object_type` is overridden. The value of this attribute should be a string that uniquely identifies the Object type (within the context of your project). It's used in many places as a key to lookup a given Object class. There are no restrictions on the characters it may contain, so feel free to make it human-friendly (using spaces instead of underscores to separate words, for example).

In lines 10-14, the class attribute `_schema` is overridden. The value of this attribute should be a colander schema representing the user-editable attributes for the Object type (the sort of attributes that might be shown as fields in an edit form). This is standard colander stuff, and you can use all the colander types (including Mapping and Sequence). Additionally Audrey defines a couple of its own colander types:

1. *`audrey.types.File`* - This type represents an uploaded file which will be stored in the MongoDB GridFS. As an example, see line 15 where a File attribute with the name `photo` is defined for the `person` type.

2. *`audrey.types.Reference`* - This type represents a reference to another Object (possibly in another collection). As an example, see lines 46-48 where a Reference attribute with the name `author` is defined to allow a reference from the `post` type to the `person` type.

In lines 16-25, the method `get_title()` is overridden. This method should return a string suitable for use as a human-friendly title of an Object instance (as might be shown as the text in a link to the object). If you don't override this method, it will return the object's `__name__` by default. The implementation of `Person.get_title()` is a little long since it tries to be flexible and handle cases where the "firstname" and "lastname" attributes may be missing. The implementation of `Post.get_title()` at line 45 is a one-liner suitable for types that have a single attribute that's a natural fit for a title.

For a lot of object types, that's all you'll need to override. It should go without saying that since these are just Python classes, you're free to override other methods and add your own to suit your specific needs.

Moving on, lines 27-29 define a `People` class that subclasses *`audrey.resources.collection.Collection`*. This is pretty short and sweet.

Line 28 overrides the `_collection_name` class attribute. The value of this attribute is a string that uniquely identifies the Collection within the content of your project. It's used as a key/name to traverse from the root of the app to a singleton instance of the Collection.

Line 29 overrides the `_object_classes` class attribute. The value of this attribute is a sequence of Object classes representing the types of Objects that may exist in the Collection. In this case, the People Collection is homogenous and only contains Person Objects. You can, however, define Collections that may contain multiple Object types (presumably with some common sub-schema). When creating Object types that will be in a non-homogenous Collection, be sure to set the `audrey.resources.object.Object._save_object_type_to_mongo` class attribute to `True`; otherwise the Collection will raise an exception while deserializing from MongoDB since it won't be able to determine the correct Object class to construct.

Lines 31-61 define another Object type and another homogenous Collection. The `Post` class demonstrates overriding the *`audrey.resources.object.Object.get_class_schema()`* class method to do deferred schema binding at runtime.

Lines 63-64 define a `Root` class that subclasses *`audrey.resources.root.Root`* and overrides the `_collection_classes` class attribute. The value of this attribute is a sequence of Collection classes representing all the Collections in use in the app.

Lines 66-67 define a `root_factory()` function which returns an instance of `Root` for a request. This function is used by Audrey to configure the Pyramid application to find the traversal root.

If you haven't read the *Introduction* section yet, you may want to now. It demonstrates some of the functionality Audrey provides using the `Person` and `People` classes defined here as examples.

You may also want to explore the *API* documentation to discover further functionality and details.

CHAPTER 3

Installation

Note: I'm developing Audrey on Linux. I'm assuming that the instructions below would work just as well under OS X, but can't say for sure. No idea about Windows.

Prerequisites

1. Python 2.7 and virtualenv

   If you don't already have these, refer to the Pyramid docs for instructions.

2. MongoDB

   If you don't already have a MongoDB server, install the latest production release from http://www.mongodb.org/downloads

   If you just want to quickly try out Audrey, here's a recipe for running a MongoDB server in the foreground under your non-root user account:

   ```
   wget http://fastdl.mongodb.org/linux/mongodb-linux-x86_64-2.0.6.tgz
   tar xfz mongodb-linux-x86_64-2.0.6.tgz
   ln -s mongodb-linux-x86_64-2.0.6 mongodb
   cd mongodb
   mkdir -p data
   bin/mongod --dbpath=data --rest
   ```

3. ElasticSearch (optional; full text and cross-collection search won't work without it)

   If you don't already have an ElasticSearch server, install the latest production release from http://www.elasticsearch.org/download/

   If you just want to quickly try out Audrey, here's a recipe for running an ElasticSearch server in the foreground under your non-root user account:

```
wget http://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-
↪0.20.2.tar.gz
tar xfz elasticsearch-0.20.2.tar.gz
ln -s elasticsearch-0.20.2 elasticsearch
cd elasticsearch
bin/plugin -install elasticsearch/elasticsearch-transport-thrift/1.4.0
bin/elasticsearch -f
```

Setup Audrey

1. Create and activate a Python virtual environment. For example:

   ```
   virtualenv myenv
   cd myenv
   source bin/activate
   ```

2. Move or clone the `Audrey` directory from Github into `myenv`. Then:

   ```
   cd Audrey
   python setup.py install
   ```

   Wait patiently for all of the dependencies to download and install.

   [FIXME: Upload Audrey to PyPI so you can just pip install it.]

Creating a new project

Audrey includes a scaffold to bootstrap a new project. Run it from the root directory of your virtualenv like this example:

```
cd $VIRTUAL_ENV
pcreate -s audrey MyProject
cd MyProject
python setup.py develop
```

As examples to get you started, `myaudreyproject/resources.py` defines two object types (Person and Post) as well as two collections (People and Posts) and a Root class to tie it all together. You will of course want to replace these classes with your own app-specific types, but for now just refer to them as you walk through the *Resource Modelling* section and see how everything fits together.

If you aren't running MongoDB and ElasticSearch on default ports on localhost (as described in the *Prerequisites* section above), edit `development.ini` and adjust the connection settings `mongo_uri`, `mongo_name`, `elastic_uri` and `elastic_name`.

Startup commands

You can now use the usual Pyramid commands, such as these examples.

Start the web server:

```
pserve development.ini --reload
```

Start an interactive shell:

```
pshell development.ini#main
```

Run tests:

```
nosetests --cover-package=myproject --cover-erase --with-coverage --cover-html
```

CHAPTER 4

API

audrey.resources

object

class `audrey.resources.object.`**`NamedObject`**(*request*, ***kwargs*)

A subclass of `Object` that has an editable `__name__` attribute.

`get_nonschema_values`()

Get the names and values of "non-schema" attributes.

Return type dictionary with the same keys as returned by `Object.get_nonschema_values()` plus:

- "__name__": string or None

class `audrey.resources.object.`**`Object`**(*request*, ***kwargs*)

Base class for objects that can be stored in MongoDB and indexed in ElasticSearch.

Developers extending Audrey should create their own subclass(es) of Object that:

- override class attribute `_object_type`; this string should uniquely identify the Object type within the context of an Audrey application
- override either the `_schema` class attributes or the `get_class_schema()` class method. Either way, `get_class_schema()` should return a colander schema for the type.
- override `get_title()` to return a suitable title string for an instance of the type.

If the type has some non-schema attributes that you store in Mongo, override `get_nonschema_values()` and `set_nonschema_values()`. When overriding, be sure to call the superclass methods since the base `Object` type uses these methods for metadata (`_id`, `_created`, etc).

If the type is to be part of a non-homogenous collection, override the class attribute `_save_object_type_to_mongo` to `True`. This is defaulted to `False` under the assumption that homogenous collections are the norm, in which case storing the same `_object_type` in every document is redundant.

If ElasticSearch indexing isn't desired, override the class attribute `_use_elastic` to `False`.

`dereference`(*reference*)

Return the object referred to by `reference`.

Parameters **`reference`** (*`audrey.resources.reference.Reference`* or `None`) – a reference

Return type *`Object`* or `None`

`generate_etag`()

Compute an Etag based on the object's schema values.

Return type string

`get_all_files`()

Returns a list of all the File objects that this object refers to (via schema or non-schema attributes).

Return type list of *`audrey.resources.file.File`* instances

`get_all_referenced_objects`()

Returns a list of all the Objects that this object refers to (via schema or non-schema attributes).

Return type list of *`Object`* instances

`get_all_references`()

Returns a list of all the Reference objects that this object refers to (via schema or non-schema attributes).

Return type list of *`audrey.resources.reference.Reference`* objects

`get_all_values`()

Returns a dictionary of both schema and nonschema values.

Return type dictionary

classmethod `get_class_schema`(*request=None*)

Return a colander schema describing the user-editable attributes for this Object type.

Parameters **`request`** (`pyramid.request.Request`) – the current request, possibly `None`

Return type `colander.SchemaNode`

Audrey makes use of the following custom `SchemaNode` kwargs for `colander.String` nodes:

- `include_in_text`: boolean, defaults to True; if True, the value will be included in Elastic's full text index.
- `is_html`: boolean, defaults to False; if True, the value will be stripped of html markup before being indexed in Elastic.

The default implementation of this method simply returns the class attribute `_schema`.

If a `request` is passed, you may opt to use it to get access to various interesting bits of data like the current user, a context object, etc. You could use that to set default values, vocabulary lists, etc. If you opt to customize the schema for each request, be sure to start by creating a deepcopy of `_schema`, or ditch the use of that class attribute altogether and construct the schema inside this method.

`get_dbref`(*include_database=False*)

Return a DBRef for this object.

Parameters **`include_database`** (*`boolean`*) – Should the database name be included in the DBRef?

Return type `bson.dbref.DBRef`

`get_elastic_connection`()
Return a connection to the ElasticSearch server. May return `None` if the class attribute `_use_elastic` is `False` for this Object type or the Collection, or if no ElasticSearch connection is configured for the app.

Return type `pyes.es.ES` or `None`

`get_elastic_doctype`()
Return the ElasticSearch document type for this object.

Note that Audrey uses Collection names as the Elastic doctype. This is just a convenience method that returns the type from the collection.

Return type string

`get_elastic_index_doc`()
Returns a dictionary representing this object suitable for indexing in ElasticSearch.

Return type dictionary

`get_elastic_index_name`()
Return the name of the ElasticSearch index for this object.

Note that all objects in an Audrey app will use the same Elastic index (the index name is analogous to a database name). This is just a convenience method that returns the name from the root.

Return type string

`get_fulltext_to_index`()
Returns a string containing the "full text" for this object.

Text is found by walking over the schema values looking for `colander.String` nodes that don't have the attribute `include_in_text` set to `False`. (If the attribute is missing, it defaults to `True`.)

If the schema node has the attribute `is_html` set to `True`, the text value will be stripped of HTML markup. (If the attribute is missing, it defaults to `False`.)

Return type string

`get_gridfs_file`(*file*)
Return the GridFS file referred to by `file`.

Parameters **`file`** (*`audrey.resources.file.File`* or `None`) – a file

Return type `gridfs.grid_file.GridOut` or `None`

`get_mongo_collection`()
Return the MongoDB Collection that contains this object's document.

Return type `pymongo.collection.Collection`

`get_mongo_save_doc`()
Returns a dictionary representing this object suitable for saving in MongoDB.

Return type dictionary

`get_nonschema_values`()
Get the names and values of "non-schema" attributes.

Return type

dictionary with the keys:

- "_id": ObjectId or None

- "_created": datetime.datetime (UTC) or None
- "_modified": datetime.datetime (UTC) or None
- "_etag": string or None

`get_schema`()

Return the colander schema for this `Object` type.

Return type `colander.SchemaNode`

`get_schema_names`()

Return the names of the top-level schema nodes.

Return type list of strings

`get_schema_values`()

Return a dictionary of this object's schema names and values.

Return type dictionary

`get_title`()

Return a "title" for this object. This should ideally be a human-friendly string such as might be displayed as the text of a link to the object.

The default implementation boringly returns the object's `__name__` or "Untitled".

Return type string

`index`()

Index (or reindex) this object in ElasticSearch.

Note that this is a no-op when use of ElasticSearch is disabled (for this Object, its collection or the app).

`load_mongo_doc`(*doc*)

Update the object's attribute values using values from `doc`.

Note that as appropriate, ObjectIds and DBRefs will be converted to *`audrey.resources.reference.Reference`* or *`audrey.resources.file.File`* instances.

Parameters **`doc`** (*`dictionary`*) – a MongoDB document (such as returned by `pymongo.collection.Collection.find_one()`)

`save`(*validate_schema=True*, *index=True*, *set_modified=True*, *set_etag=True*)

Save this object in MongoDB (and optionally ElasticSearch).

Parameters

- **`validate_schema`** (*`boolean`*) – Should the colander schema be validated first? If `True`, may raise `colander.Invalid`.
- **`index`** (*`boolean`*) – Should the object be (re-)indexed in ElasticSearch?
- **`set_modified`** (*`boolean`*) – Should the object's last modified timestamp (`_modified`) be updated?
- **`set_etag`** (*`boolean`*) – Should the object's Etag be updated?

`set_all_schema_values`(***kwargs*)

Set attribute values from `kwargs` for **all** top-level schema nodes. Schema nodes that are missing in `kwargs` will be set to `None`.

`set_nonschema_values`(***kwargs*)

Set this instance's non-schema values from `kwargs`.

set_schema_values(***kwargs*)
Set attribute values for the top-level schema nodes present in `kwargs`.

unindex()
Unindex this object in ElasticSearch.

Return type integer

Returns the number of items affected (normally this will be 1, but it may be 0 if use of ElasticSearch is disabled or if the object wasn't indexed to begin with).

validate_schema()
Runs the instance's schema attribute values thru a serialize-deserialize roundtrip. This will raise a `colander.Invalid` exception if the schema fails to validate. Otherwise it will have the effect of applying `default` and `missing` values.

collection

class `audrey.resources.collection.`**Collection**(*request*)
A set of Objects. Corresponds to a MongoDB Collection (and an ElasticSearch "type").

Developers extending Audrey should create their own subclass(es) of Collection that:

- override class attribute `_collection_name`; this string is used for traversal to a Collection from Root, as the name of the MongoDB collection, and as the name of the ElasticSearch doctype.
- override either the `_object_classes` class attribute or the *`get_object_classes()`* class method. Either way, *`get_object_classes()`* should return a sequence of the *`audrey.resources.object.Object`* classes that can be stored in this collection.

If Mongo indexes are desired for the collection, override the class method *`get_mongo_indexes()`*.

If an ElasticSearch mapping is desired, override the class method *`get_elastic_mapping()`*.

If ElasticSearch indexing isn't desired, override the class attribute `_use_elastic` to `False`.

add_child(*child*, *validate_schema=True*)
Add a child object to this collection. Note that this will ultimately call the child's *`audrey.resources.object.Object.save()`* method, persisting it in Mongo (and indexing in Elastic). If `validate_schema` is `True`, a `colander.Invalid` exception may be raised if schema validation fails.

Parameters

- **child** (*`audrey.resources.object.Object`*) – a child to be added to this collection
- **validate_schema** (*boolean*) – Should we validate the schema before adding the child?

clear_elastic()
Delete all documents from Elastic for this Collection's doctype.

construct_child_from_mongo_doc(*doc*)
Given a MongoDB document (presumably from this collection), construct and return an Object.

Parameters **doc** (*dictionary*) – a MongoDB document (such as returned by `pymongo.collection.Collection.find_one()`)

Return type *`audrey.resources.object.Object`*

delete_child(*child_obj*)
Remove a child object from this collection.

Parameters **child** (*audrey.resources.object.Object*) – a child to be added to this collection

delete_child_by_id (*id*)
Remove a child object (identified by the given `id`) from this collection.

Parameters **name** (`bson.objectid.ObjectId`) – ID of the child to remove

Return type integer indicating number of children removed. Should be 1 normally, but may be 0 if no child was found with the given `id`.

delete_child_by_name (*name*)
Remove a child object (identified by the given `name`) from this collection.

Parameters **name** (*string*) – name of the child to remove

Return type integer indicating number of children removed. Should be 1 normally, but may be 0 if no child was found with the given `name`.

get_child (*spec=None*, *sort=None*, *fields=None*)
Return the first child matching the query parms.

Parameters

- **spec** (dictionary or `None`) – a MongoDB query spec (as used by `pymongo.collection.Collection.find()`)
- **sort** (a list of (key, direction) tuples or `None`) – a MongoDB sort parameter
- **fields** (list of strings or dict with boolean values or `None`) – a list of field names to retrieve or `None` for all fields. May also be a dict to exclude fields (example: `fields={'body':False}`).

Return type *audrey.resources.object.Object* or `None`

get_child_by_id (*id*, *fields=None*)
Return the child object for the given `id`.

Parameters

- **id** (`bson.objectid.ObjectId`) – an ObjectId
- **fields** (list of strings or dict with boolean values or `None`) – a list of field names to retrieve or `None` for all fields. May also be a dict to exclude fields (example: `fields={'body':False}`).

Return type *audrey.resources.object.Object* class or `None`

get_child_by_name (*name*, *fields=None*)
Return the child object for the given `name`.

Parameters

- **name** (*string*) – an object name
- **fields** (list of strings or dict with boolean values or `None`) – a list of field names to retrieve or `None` for all fields. May also be a dict to exclude fields (example: `fields={'body':False}`).

Return type *audrey.resources.object.Object* class or `None`

get_child_names (*spec=None*, *sort=None*, *skip=0*, *limit=0*)
Return the child names matching the query parameters.

Parameters

- **spec** (dictionary or `None`) – a MongoDB query spec (as used by `pymongo.collection.Collection.find()`)
- **sort** (a list of (key, direction) tuples or `None`) – a MongoDB sort parameter
- **skip** (*integer*) – number of documents to omit from start of result set
- **limit** (*integer*) – maximum number of children to return

Return type a sequence of __name__ strings

get_child_names_and_total(*spec=None*, *sort=None*, *skip=0*, *limit=0*)
Query for children and return the total number of matching children and a list of the child names (or a batch of child names if the `limit` parameter is non-zero).

Parameters

- **spec** (dictionary or `None`) – a MongoDB query spec (as used by `pymongo.collection.Collection.find()`)
- **sort** (a list of (key, direction) tuples or `None`) – a MongoDB sort parameter
- **skip** (*integer*) – number of documents to omit from start of result set
- **limit** (*integer*) – maximum number of children to return

Return type

dictionary with the keys:

- "total" - an integer indicating the total number of children matching the query `spec`
- "items" - a sequence of __name__ strings

get_children(*spec=None*, *sort=None*, *skip=0*, *limit=0*, *fields=None*)
Return the children matching the query parameters.

Parameters

- **spec** (dictionary or `None`) – a MongoDB query spec (as used by `pymongo.collection.Collection.find()`)
- **sort** (a list of (key, direction) tuples or `None`) – a MongoDB sort parameter
- **skip** (*integer*) – number of documents to omit from start of result set
- **limit** (*integer*) – maximum number of children to return
- **fields** (list of strings or dict with boolean values or `None`) – a list of field names to retrieve or `None` for all fields. May also be a dict to exclude fields (example: `fields={'body':False}`).

Return type a sequence of `audrey.resources.object.Object` instances

get_children_and_total(*spec=None*, *sort=None*, *skip=0*, *limit=0*, *fields=None*)
Query for children and return the total number of matching children and a list of the children (or a batch of children if the `limit` parameter is non-zero).

Parameters

- **spec** (dictionary or `None`) – a MongoDB query spec (as used by `pymongo.collection.Collection.find()`)
- **sort** (a list of (key, direction) tuples or `None`) – a MongoDB sort parameter
- **skip** (*integer*) – number of documents to omit from start of result set
- **limit** (*integer*) – maximum number of children to return

- **`fields`** (list of strings or dict with boolean values or `None`) – a list of field names to retrieve or `None` for all fields. May also be a dict to exclude fields (example: `fields={'body':False}`).

Return type

dictionary with the keys:

- "total" - an integer indicating the total number of children matching the query `spec`
- "items" - a sequence of `audrey.resources.object.Object` instances

`get_children_lazily`(*spec=None*, *sort=None*, *fields=None*)

Return child objects matching the query parameters using a generator. Great when you want to iterate over a potentially large number of children and don't want to load them all into memory at once.

Parameters

- **`spec`** (dictionary or `None`) – a MongoDB query spec (as used by `pymongo.collection.Collection.find()`)
- **`sort`** (a list of (key, direction) tuples or `None`) – a MongoDB sort parameter
- **`fields`** (list of strings or dict with boolean values or `None`) – a list of field names to retrieve or `None` for all fields. May also be a dict to exclude fields (example: `fields={'body':False}`).

Return type a generator of `audrey.resources.object.Object` instances

`get_elastic_connection`()

Return a connection to the ElasticSearch server. May return `None` if the class attribute `_use_elastic` is `False`, or if no ElasticSearch connection is configured for the app.

Return type `pyes.es.ES` or `None`

`get_elastic_doctype`()

Return the ElasticSearch document type for this collection.

Note that Audrey uses the `_collection_name` as the doctype.

Return type string

`get_elastic_index_name`()

Return the name of the ElasticSearch index.

Note that all objects in an Audrey app will use the same Elastic index (the index name is analogous to a database name). This is just a convenience method that returns the name from the root.

Return type string

classmethod `get_elastic_mapping`()

Return a dictionary representing ElasticSearch mapping properties for this collection. Refer to http://www.elasticsearch.org/guide/reference/mapping/

Return type dictionary

`get_mongo_collection`()

Return the MongoDB Collection for this collection.

Return type `pymongo.collection.Collection`

classmethod `get_mongo_indexes`()

Return a list of data about the desired MongoDB indexes for this collection. The first item of each tuple is the ensure_index `key_or_list` parm. The second item of each tuple is a dictionary that will be passed as kwargs.

Return type sequence of two-item tuples, each with the two parameters to be passed to a call to `pymongo.collection.Collection.ensure_index()`

The default implementation returns an empty list, meaning that no indexes will be ensured.

get_object_class(*object_type*)
Return the class that corresponds to the `object_type` string.

Parameters **object_type** (*string*) – name of an object type

Return type *audrey.resources.object.Object* class or `None`

classmethod get_object_classes()
Returns a sequence of the Object classes that this Collection manages.

Return type sequence of *audrey.resources.object.Object* classes

get_object_types()
Return the `_object_types` that this collection manages.

Return type list of strings

has_child_with_id(*id*)
Does this collection have a child with the given `id`?

Parameters **id** (`bson.objectid.ObjectId`) – an ObjectId

Return type boolean

has_child_with_name(*name*)
Does this collection have a child with the given `name`?

Parameters **name** (*string*) – an object name

Return type boolean

reindex_all(*clear=False*)
Reindex all this collection's objects in Elastic. Returns a count of the objects reindexed.

Parameters **clear** (*boolean*) – Should we clear the index first?

Return type integer

veto_add_child(*child*)
Check whether the collection will allow the given `child` to be added. If there is some objection, return a string describing the objection. Else return `None` to indicate the child is OK.

Parameters **child** (*audrey.resources.object.Object*) – a child to be added to this collection

Return type string or `None`

class audrey.resources.collection.**NamingCollection**(*request*)
A subclass of *Collection* that allows control over the `__name__` attribute.

rename_child(*name*, *newname*, *validate=True*)
Rename a child of this collection. May raise a *audrey.exceptions.Veto* exception if `validate` is `True` and the `newname` is vetoed.

Parameters

- **name** (*string*) – name of the child to rename
- **newname** (*string*) – new name for the child
- **validate** (*boolean*) – Should we validate the new name first?

Return type integer indicating number of children renamed. Should be 1 normally, but may be 0 if `newname` == `name`.

`validate_name_format`(*name*)

Is the given name in an acceptable format? If so, return `None`. Otherwise return an error string explaining the problem.

Parameters **`name`** (*`string`*) – name of a child object

Return type string or `None`

If you want to restrict the format of names (legal characters, etc) override this method to check the name and return an error if needed.

Note that this method doesn't have to test whether the name is empty or already in use.

`veto_child_name`(*name*, *unique=True*)

Check whether the collection will allow a child with the given `name` to be added. If there is some objection, return a string describing the objection. Else return `None` to indicate the child name is OK.

Parameters

- **`name`** (*`string`*) – name of a child object
- **`unique`** (*`boolean`*) – Should we check that the name is unique (not already in use)?

Return type string or `None`

root

class `audrey.resources.root.`**`Root`**(*request*)

The root of the application (starting point for traversal) and container of Collections.

Developers extending Audrey should create their own subclass of Root that:

- overrides either the `_collection_classes` class attribute or the *`get_collection_classes()`* class method. Either way, *`get_collection_classes()`* should return a sequence of *`audrey.resources.collection.Collection`* classes.

`basic_fulltext_search`(*search_string=''*, *collection_names=None*, *skip=0*, *limit=10*, *sort=None*, *highlight_fields=None*, *object_fields=None*)

A functional basic full text search. Also a good example of using the other search methods.

All parms are optional. Calling the method without specifying any parms queries for anything and everything.

Parameters

- **`query`** (*`string`*) – a query string that may contain wildcards or boolean operators
- **`collection_names`** (list of strings, or `None`) – restrict search to specific Collections
- **`skip`** (*`integer`*) – number of results to omit from start of result set
- **`limit`** (*`integer`*) – maximum number of results to return
- **`sort`** (string or `None`) – a *`audrey.sortutil.SortSpec`* string; default sort is by relevance
- **`highlight_fields`** (list of strings, or `None`) – a list of Elastic mapping fields in which to highlight `search_string` matches. For example, to highlight matches in Audrey's default full "text" field: `['text']`

- **`object_fields`** – like `fields` param to *`audrey.resources.collection.Collection.get_children()`*)

Return type dictionary

Returns a dictionary like *`get_objects_and_highlights_for_raw_search_results()`* when `highlight_fields`. Otherwise returns a dictionary like *`get_objects_for_raw_search_results()`*.

`clear_elastic`()
Delete all documents from Elastic for all Collections.

`create_gridfs_file`(*file*, *filename*, *mimetype*, *parents=None*)
Create a new GridFS file.

Parameters

- **`file`** (*`a file-like object (providing a read() method) or a string`*) – file content/data
- **`filename`** (*`string`*) – a filename
- **`mimetype`** (*`string`*) – a mime-type
- **`parents`** (list of `bson.dbref.DBRef` instances, or `None`) – list of references to the Objects that “own” the file

Return type *`audrey.resources.file.File`*

`create_gridfs_file_from_fieldstorage`(*fieldstorage*, *parents=None*)
Create a new GridFS file from the given `fieldstorage`.

Parameters

- **`fieldstorage`** (`cgi.FieldStorage`) – a FieldStorage (such as found in WebOb request.POST for each file upload)
- **`parents`** (list of `bson.dbref.DBRef` instances, or `None`) – list of references to the Objects that “own” the file

Return type *`audrey.resources.file.File`*

`get_collection`(*name*)
Return the Collection for the given `name`. The returned Collection will have the Root object as its traversal `__parent__`.

Parameters **`name`** (*`string`*) – a collection name

Return type *`audrey.resources.collection.Collection`* class or `None`

classmethod `get_collection_classes`()
Returns a sequence of the Collection classes in this app.

Return type sequence of *`audrey.resources.collection.Collection`* classes

`get_collection_names`()
Get the names of the collections.

Return type list of strings

`get_collections`()
Get all the collections.

Return type list of *`audrey.resources.collection.Collection`* instances

`get_elastic_connection``()`
Return a connection to the ElasticSearch server. May return `None` if no ElasticSearch connection is configured for the app.

Return type `pyes.es.ES` or `None`

`get_elastic_index_name``()`
Return the name of the ElasticSearch index.

Note that all objects in an Audrey app will use the same Elastic index (the index name is analogous to a database name).

Return type string

`get_gridfs``()`
Return the MongoDB GridFS for the app.

Return type `gridfs.GridFS`

`get_mongo_collection``(`*coll_name*`)`
Return the collection identified by `coll_name`.

Return type `pymongo.collection.Collection`

`get_mongo_connection``()`
Return a connection to the MongoDB server.

Return type `pymongo.connection.Connection`

`get_mongo_db``()`
Return the MongoDB database for the app.

Return type `pymongo.database.Database`

`get_mongo_db_name``()`
Return the name of the MongoDB database.

Return type string

`get_object_for_collection_and_id``(`*collection_name*, *id*, *fields=None*`)`
Return the Object identified by the given `collection_name` and `id`.

Parameters

- **`collection_name`** (*`string`*) – name of a collection
- **`id`** (`bson.objectid.ObjectId`) – an ObjectId
- **`fields`** – like `fields` param to *`audrey.resources.collection.Collection.get_children()`*

Return type *`audrey.resources.object.Object`* class or `None`

`get_object_for_reference``(`*reference*, *fields=None*`)`
Return the Object identified by the given `reference`.

Parameters

- **`reference`** (*`audrey.resources.reference.Reference`*) – a reference
- **`fields`** – like `fields` param to *`audrey.resources.collection.Collection.get_children()`*

Return type *`audrey.resources.object.Object`* class or `None`

get_objects_and_highlights_for_query(*query=None*, *doc_types=None*, *object_fields=None*, ***query_parms*)
A convenience method that returns the result of calling `get_objects_and_highlights_for_raw_search_resu` on `search_raw()` with the given parameters.

get_objects_and_highlights_for_raw_search_results(*results*, *object_fields=None*)
Given a `pyes` result dictionary (such as returned by `search_raw()`) return a new dictionary with the keys:

- "total": total number of matching hits
- "took": search time in ms
- "items": a list of dictionaries, each with the keys "object" and highlight"

Parameters **object_fields** – like `fields` param to `audrey.resources.collection.Collection.get_children()`)

get_objects_for_query(*query=None*, *doc_types=None*, *object_fields=None*, ***query_parms*)
A convenience method that returns the result of calling `get_objects_for_raw_search_results()` on `search_raw()` with the given parameters.

get_objects_for_raw_search_results(*results*, *object_fields=None*)
Given a `pyes` result dictionary (such as returned by `search_raw()`) return a new dictionary with the keys:

- "total": total number of matching hits
- "took": search time in ms
- "items": a list of `audrey.resources.object.Object` instances

Parameters **object_fields** – like `fields` param to `audrey.resources.collection.Collection.get_children()`)

reindex_all(*clear=False*)
Reindex all documents in Elastic for all Collections. Returns a count of the objects reindexed.

Parameters **clear** (*boolean*) – Should we clear the index first?

Return type integer

search_raw(*query=None*, *doc_types=None*, ***query_parms*)
A thin wrapper around `pyes.ES.search_raw()`

Parameters

- **query** – a `pyes.query.Search` or a `pyes.query.Query` or a custom dictionary of search parameters using the query DSL to be passed directly
- **doc_types** (list of strings or `None`) – which doc types to search
- **query_parms** – extra kwargs

Return type dictionary

The returned dictionary is like that returned by `pyes.ES.search_raw()`

Keys are ['hits', '_shards', 'took', 'timed_out'].

result['took'] is the search time in ms

result['hits'] has the keys: ['hits', 'total', 'max_score']

result['hits']['total'] is total number of hits result['hits']['hits'] is a list of hit dictionaries, each with the keys: ['_score', '_type', '_id', '_source', '_index', 'highlight'] Although if the `fields` kwarg is a list of field names (instead of the default value `None`), instead of a '_source' key, each hit will have a '_fields' key whose value is a dictionary of the requested fields.

The "highlight" key will only be present if the query has highlight fields and there was a match in at least one of those fields. In that case, the value of "highlight" will be dictionary of strings. Each dictionary key is a field name and each string is an HTML fragment where the matched term is in an `<em>` tag.

`serve_gridfs_file_for_id`(*id*)
Attempt to serve the GridFS file referred to by `id`.

Parameters **`id`** (`bson.objectid.ObjectId`) – an ObjectId

Return type `pyramid.response.Response` if a matching file was found in the GridFS, otherwise `pyramid.httpexceptions.HTTPNotFound`

Warning: The following sections are incomplete and poorly formatted. Should improve as I continue to work on the docstrings in the source.

file

class `audrey.resources.file.`**`File`**(*_id*)
Wrapper around a GridFS file. Instances of Object use this File type for attributes that refer to files in the GridFS.

`get_gridfs_file`(*request*)
Returns an instance of `gridfs.grid_file.GridOut` for the GridFS file that this File object refers to by ID. If no match in GridFS is found, returns `None`.

`serve`(*request*)
Serve the GridFS file referred to by this object. Returns a `pyramid.response.Response` if a matching file was found in the GridFS. Otherwise returns `pyramid.httpexceptions.HTTPNotFound`.

reference

class `audrey.resources.reference.`**`IdReference`**(*collection*, *id*)
Just a little syntactic sugar around *`Reference`* with `serialize_id_only` = `True`.

class `audrey.resources.reference.`**`Reference`**(*collection*, *id*, *serialize_id_only=False*)
Represents a reference to a document in MongoDB. Similar to Mongo's standard DBRef, but has an option to serialize only the ID, which can be more space/bandwidth efficient when the reference will always be to the same collection.

`dereference`(*context*)
Return the `Object` this Reference refers to. `context` can be any resource and is simply used to find the root (which in turn is used to resolve the reference).

audrey.types

class `audrey.types.`**`File`**
colander type representing an audrey.resources.file.File Serializes to/from a dict with the key "FileId" whose value is a string representation of the file's ID in the GridFS.

class `audrey.types.`**`Reference`**(*collection=None*)

colander type representing an audrey.resources.reference.Reference.

This type constructor accepts one argument:

`collection` The name of the *`audrey.resources.collection.Collection`* that this reference will always refer to. May be None if this reference may refer to multiple collections.

When collection is None, the Reference is serialized to and deserialized from a dict with the keys: "ObjectId" "collection"

When collection is not None, the dictionary will only have the "ObjectId" key.

audrey.views

`audrey.views.`**`str_to_bool`**(*s*, *default=None*)

Interpret the given string `s` as a boolean value. `default` should be one of `None`, `True`, or `False`.

`audrey.views.`**`str_to_list`**(*s*, *default=None*)

Interpret the given string `s` as a comma-delimited list of strings.

audrey.colanderutil

class `audrey.colanderutil.`**`SchemaConverter`**

Converts a colander schema to a JSON Schema (expressed as a data structure consisting of primitive Python types, suitable for serializing to JSON).

audrey.dateutil

`audrey.dateutil.`**`convert_naive_datetime`**(*dt*, *tz_from*, *tz_to*)

Convert a naive datetime.datetime "dt" from one timezone to another. tz_from and tz_to may be either pytz.timezone instances, or timezone strings. Examples: Convert UTC datetime to US/Eastern: convert_datetime(datetime.datetime.utcnow(), pytz.utc, 'US/Eastern')

Convert US/Eastern datetime to UTC: convert_datetime(datetime.datetime.now(), 'US/Eastern', pytz.utc)

Convert US/Eastern datetime to US/Pacific: convert_datetime(datetime.datetime.now(), 'US/Eastern', 'US/Pacific')

`audrey.dateutil.`**`utcnow`**(*zero_seconds=False*)

Returns a timezone aware version of utcnow. (datetime.datetime.utcnow() returns a naive version.) If zero_seconds, the datetime will be rounded down to the minute.

audrey.sortutil

class `audrey.sortutil.`**`SortSpec`**(*sort_string=None*)

It seems that everything that supports sorting has a different way of specifying the sort parameters. SortSpec tries to be a generic way to specify sort parms, and has methods to convert to sort specifications used by other systems in Audrey (MongoDB and Elastic).

SortSpec's preferred way to represent sort parms is as a comma-delimited string. Each part of the string is a field name optionally prefixed with a plus or minus sign. Minus indicates descending order; plus (or the absence of a sign) indicates ascending.

Example: "foo,-bar" or "+foo,-bar" both indicate primary sort by "foo" ascending and secondary sort by "bar" descending.

Rationale: Strings are easy to sling around as HTTP query parameters (compared for example to Mongo's sequence of two-tuples). This string format is as simple, concise and understandable (even for normal folks) as I could come up with (contrast with Elastic's more verbose ":desc" suffixes).

audrey.exceptions

exception `audrey.exceptions.`**`Veto`**(*msg*)

May be raised when an attempt is made to do something that the app doesn't allow, such as adding an object to a folder with a name that's already in use. Higher level code (such as views) may want to catch these Veto exceptions and present them to end users in a friendly manner.

Python Module Index

Index