sandman2 Documentation

Release 0.0.1

Jeff Knupp

Dec 21, 2020

Contents

Contents:

CHAPTER 1

Quickstart

1.1 Install

The easiest way to install sandman2 is using `pip`:

```
$ pip install sandman2
```

1.2 `sandman2ctl`

Once installed, sandman2 provides a command-line utility, `sandman2ctl`, that takes your database's URL as a command-line argument and starts a RESTful API server immediately. Database URLs are RFC-1738-style URLs. For more information, please read the SQLAlchemy documentation on the matter.

Note: When using **SQLite**, use `pysqlite` as the driver name (i.e. `sqlite+pysqlite:///relative/path/to/db`).

By default, all database tables will be introspected and made available as API resources (don't worry if this is not the behavior desired; there are easy ways to configure the exact behavior of `sandman2`, discussed later in the documentation). The default URL for each table is a slash followed by the table's name in all lower case (e.g. an "Artist" table would be found at `localhost:5000/artist/`).

1.3 Using Your New REST API

If you've successfully pointed `sandman2ctl` at your database, you should see output like the following:

```
$ sandman2ctl 'sqlite+pysqlite:///path/to/sqlite/database'
 * Running on http://0.0.0.0:5000/
```

The API service is available on port 5000 by default (though this is configurable). You can interact with your service using `curl` or any other HTTP client.

CHAPTER 2

Basics

sandman2 exposes each table in your database as a `resource` in a RESTful API service. You access and manipulate resources by sending HTTP requests with various `methods` (e.g. `GET`, `POST`, `DELETE`, etc.). Here's a quick list of common tasks and the HTTP method to use to complete them:

2.1 Retrieve all records in a table

In REST-terms, a table in your database represents a `resource`. A *group* of resources (i.e. data returned from a `SELECT * FROM foo` statement) is called a `collection`. To retrieve a `collection` of resources from your API, make a HTTP `GET` call to the resource's base URL with a trailing `/`. The `/` indicates that this URL represents a `collection` rather than a single `resource`. By default, a `resource`'s base URL is set to `/<table_name_in_lowercase>/`. If you had an `Artist` table in your database, you would use the following `curl` command to retrieve the collection of all `Artist` resources:

```
$ curl http://127.0.0.1:5000/artist/
```

The implied HTTP method in this case is `GET`. The response would be a JSON list of all the artist resources in the database.

2.1.1 Possible HTTP status codes for response

- `200 OK` if the resource is found
- `404 Not Found` if the resource can't be found

2.2 Retrieve a single row

To retrieve a single resource (i.e. row in your database), use the `GET` method while specifying the value of the resource's primary key field. If our `Artist` table used the column `ArtistId` (an `integer`) as a primary key, we could retrieve a single resource like so:

```
$ curl http://127.0.0.1:5000/artist/3
```

Again, the implied HTTP method is `GET`.

2.2.1 Possible HTTP status codes for response

- `200 OK` if the resource is found
- `404 Not Found` if the resource can't be found

2.3 Add a new row to a table

To add a resource to an existing collection, use the HTTP `POST` method on the collection's URL (e.g. `/artist/` in our example). All required fields should be sent as JSON data, and the `Content-type` header should be set to `application/json`. Here's how we would create a new `artist` resource:

```
$ curl -X POST -d '{"Name": "Jeff Knupp"}' -H "Content-Type: application/json" http://
↪127.0.0.1:5000/artist/
```

In this case, the primary key field (`ArtistId`) was not sent, since it is an auto-incremented `integer`. The response shows the assigned `ArtistId`:

```
{
    "ArtistId": 276,
    "Name": "Jeff Knupp"
}
```

We know based on our knowledge of how `sandman2` works that this new resource can be retrieved at `/artist/276` (i.e. <resource name>/<primary key value>). If we didn't know how `sandman2` worked, however, how would we know *where* the new resource was located? The `Link` HTTP response header always indicates the location a resource can be reached at, among other things.

2.3.1 Possible HTTP status codes for response

- `201 Created` if a new resource is properly created
- `400 Bad Request` if the request is malformed or missing data

2.3.2 Error conditions

If we send a field not in the record's definition, we are alerted by sandman2 in the HTTP response:

```
$ curl -X POST -d '{"Name": "Jeff Knupp2", "Age": 32}' -H "Content-Type: application/
↪json" http://127.0.0.1:5000/artist
{
    "message": "Unknown field [Age]"
}
```

Similarly, if we miss a required field, sandman2 helpfully lets us know which field(s) we missed. Imagine we had an `Album` table that contains albums for each artist. Each row has the album's title in the `Title` column and the associated `Artist`'s `ArtistId` in the `ArtistId` column. If we try to create a new album with only a `Title` set, the following is returned:

```
$ curl -X POST -d '{"Title": "For Those About To Rock We Salute You"}' -H "Content-
↪Type: application/json" http://127.0.0.1:5000/album
{
    "message": "[ArtistId] required"
}
```

2.4 Delete a single row from a table

To remove a resource from a collection, use the HTTP `DELETE` method while specifying a value for the primary key field:

```
$ curl -X DELETE http://127.0.0.1:5000/artist/276
```

2.4.1 Possible HTTP status codes for response

- `204 No Content` if the resource was found and deleted
- `404 Not Found` if the resource could not be found

2.5 Update an existing row

To update a row using the so-called "delta", i.e. just the fields that must be changed, send an HTTP `PATCH` request to the service. To change the `ArtistId` associated with an album, you could send the following `PATCH` request:

```
$ curl -X PATCH -d '{"ArtistId": 3}' -H "Content-Type: application/json" http://127.0.
↪0.1:5000/album/6
```

This updates the `Album` with ID `6` to refer to the `Artist` with ID `3`.

2.5.1 Possible HTTP status codes for response

- `200 OK` if the resource was found and updated
- `400 Bad Request` if the request is malformed or missing data
- `404 Not Found` if the resource could not be found

2.6 "Upsert" a row in a table

Some database engines support an "upsert" action where a full row is provided, including a value for the primary key. If no record with that primary key exists, the row is inserted as normal. If there *is* an existing row with the same primary key value, the operation is changed to an "update", and the existing row is updated with the new values.

The HTTP `PUT` method works in much the same way. A full copy of a resource is sent in the request. The primary key value is determined by the URL the request is sent to (i.e. a `PUT` to `/artist/3` implies an `ArtistId` of `3`). Any existing resource is overwritten with the new values.

An important property of the HTTP `PUT` is *idempotency*. An *idempotent* operation always gives the same result, regardless of how many times or in which order it is applied. You can always be sure of the state of a resource after a successful `PUT` request.

2.6.1 Possible HTTP status codes for response

- `200 OK` if the resource was found and updated
- `201 Created` if the resource was not found and a new resource was created
- `400 Bad Request` if the request is malformed or missing data

CHAPTER 3

Searching, Filtering, and Sorting

3.1 Pagination

When you perform an HTTP GET `request` on a collection, the default behavior is to return *all* of the results in the collection, though these results are *paginated*. _Pagination_ breaks up a long list of results into identically-sized chunks, called *pages*. Each page contains a predetermined number of results (`sandman2` uses a default size of `20`). Each paged response also contains a `Link` header that will include the URL for the previous, next, first, and last page of results (where applicable).

3.1.1 Controlling pagination

Pagination options are controlled by a number of optional URL parameters:

- `page (integer)`: Return the *Nth* page of results. With 100 results and a page size of 20, `page=2` would result in resources 21-40 being returned.
- `limit (integer)`: Set the number of results per page to *N*. With 100 results and a page size of 20, `limit=10` would return only the first 10 results.

3.2 Filtering

One can *filter* the resources that will be returned from a `GET` request for a collection. Filtering is done on the value of one or more of the fields in a resource. For example, suppose we had a `Person` resource with `first_name`, `last_name`, and `age` fields. To request only the resources where the person's `first_name` is "Jeff", you would make an HTTP `GET` request to the following URL:

```
/person/?first_name=Jeff
```

Notice that `person` is followed by a `/`, indicating it is a collection and not a resource. `first_name` is simply set as a URL parameter with the value set to "Jeff".

3.2.1 Combining filters

When more than one filter is specified on a request, the filters are combined and taken to be a set of clauses joined by AND. That is, a resource must match *all* filters to be returned, not just one. This behavior is useful for further refining results:

```
/person/?first_name=Jeff&last_name=Knupp
```

A `GET` request to this URL will return only the resources where `first_name` is "Jeff" *and* `last_name` is "Knupp". We could also specify an age to filter on, though only exact matches (e.g. `age=33` vs `age<40`) are currently supported for non-text fields.

3.2.2 Filtering text fields with pattern matching

Suppose we want `Person` resources not with a specific `first_name`, but where the `first_name` begins with `J`. We can specify a "like" parameter (named after the SQL `LIKE` keyword) by sending a `GET` request to:

```
/person/?%name=J%%
```

(Note that the `%` character must be URL-escaped)

The double `%` s mean "match any series of characters", so our filter is "first_name starting with J and followed by any series of characters."

3.3 Sorting

The last type of operation that controls how/which resources are returned from a collection is *sorting*. A collection can be sorted by any field on the resource

CHAPTER 4

Using your own database models

One of the most common questions is, “Does `sandman2` support using my own database models?”. The answer, of course, is YES! Here’s an example, pulled right from the unit tests, of how to “bring your own models” to `sandman2`:

```python
import datetime

from sandman2.model import db, Model

from tests.resources import (
    GET_ERROR_MESSAGE,
    INVALID_ACTION_MESSAGE,
    )

class User(db.Model, Model):

    """A user of the blogging application."""

    __tablename__ = 'user'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    email = db.Column(db.String, unique=True)

    @staticmethod
    def is_valid_get(request, resource):
        """Return error message in all cases (just for testing)."""
        return INVALID_ACTION_MESSAGE

    @staticmethod
    def is_valid_post(request, resource):
        """Return error message in all cases (just for testing)."""
        return INVALID_ACTION_MESSAGE

    @staticmethod
    def is_valid_patch(request, resource):
```

(continues on next page)

(continued from previous page)

```
        """Return error message in all cases (just for testing)."""
        return INVALID_ACTION_MESSAGE

    @staticmethod
    def is_valid_put(request, resource):
        """Return error message in all cases (just for testing)."""
        return INVALID_ACTION_MESSAGE

    @staticmethod
    def is_valid_delete(request, resource):
        """Return error message in all cases (just for testing)."""
        return INVALID_ACTION_MESSAGE


class Blog(db.Model, Model):

    """An online weblog."""

    __tablename__ = 'blog'

    id = db.Column(db.String, primary_key=True)
    name = db.Column(db.String)
    subheader = db.Column(db.String, nullable=True)
    creator_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    creator = db.relationship(User)


class Post(db.Model, Model):

    """An individual blog post."""

    __tablename__ = 'post'

    id = db.Column(db.Numeric, primary_key=True)
    title = db.Column(db.String)
    content = db.Column(db.String)
    posted_at = db.Column(db.DateTime, default=datetime.datetime.now)
    author_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    author = db.relationship(User)
```

CHAPTER 5

The Admin Interface

In addition to a RESTful API service, sandman2 provides a HTML-based *admin interface* that allows you to view and manipulate the data in your tables. To access the admin interface, simply navigate to `/admin` once your service is started.

On the left sidebar, you'll see a list of all the services you specified (i.e. the database tables you chose to include). Clicking one will show the contents of that table, paginated. You can edit a record by clicking on the pencil icon or delete a record using the trashcan icon.

You may notice that *foreign keys* are displayed with their default Python representation (i.e. `<flask_sqlalchemy.Artist object @ 0xdeadbeef>`). To show foreign keys in a more useful way, you can define your own extensions to the classes reflected in the database and add a `__unicode__` member function.

Imagine we have a simple blog application consisting of Blog, Post, and User models. As expected, each post belongs to a specific user and the model has the requisite foreign key to the User table. When we view a Post's assoicated user in the admin site, however, we see the following:

```
<flask_sqlalchemy.user object at 0x10d3cea10>
```

To provide a more useful representation in the admin, we *extend* the reflected class by creating a `models.py` file and adding functionality to our model classes. Deriving from `sandman2.AutomapModel` accomplishes this:

```
from sandman2 import AutomapModel

class User(AutomapModel):

    """A user of the blogging application."""

    __tablename__ = 'user'

    def __unicode__(self):
        return self.name


class Blog(AutomapModel):
```

(continues on next page)

(continued from previous page)

```
    """An online weblog."""

    __tablename__ = 'blog'

    def __unicode__(self):
        return self.name

class Post(AutomapModel):

    """An individual blog post."""

    __tablename__ = 'post'

    def __unicode__(self):
        return self.title
```

Notice that you can refer to attributes of the class that you know to be present (like `user.name`) without defining the `name` column; all other columns/properties are reflected. You're meerly *extending* the existing model class.

CHAPTER 6

API

sandman2 provides a minimal API to control the behavior and inclusion of resources from the database.

sandman2's main module.

class sandman2.model.**Model**

The sandman2 Model class is the base class for all RESTful resources. There is a one-to-one mapping between a table in the database and a `sandman2.model.Model`.

`__methods__ = {'DELETE', 'GET', 'HEAD', 'OPTIONS', 'PATCH', 'POST', 'PUT'}`

The HTTP methods this resource supports (default=all).

`__url__ = None`

The relative URL this resource should live at.

`__version__ = '1'`

The API version of this resource (not yet used).

classmethod description()

Return a field->data type dictionary describing this model as reported by the database.

Return type dict

links()

Return a dictionary of links to related resources that should be included in the *Link* header of an HTTP response.

Return type dict

classmethod optional()

Return a list of all nullable columns for the resource's table.

Return type list

classmethod primary_key()

Return the key of the model's primary key field.

Return type string

classmethod required()

Return a list of all columns required by the database to create the resource.

Parameters **cls** – The Model class to gather attributes from

Return type list

resource_uri()

Return the URI to this specific resource.

Return type str

to_dict()

Return the resource as a dictionary.

Return type dict

update(*attributes*)

Update the current instance based on attribute->value items in *attributes*.

Parameters **attributes** (*dict*) – Dictionary of attributes to be updated

Return type *sandman2.model.Model*

class sandman2.service.**Service**

The *Service* class is a generic extension of Flask's *MethodView*, providing default RESTful functionality for a given ORM resource.

Each service has an associated *__model__* attribute which represents the ORM resource it exposes. Services are JSON-only. HTML-based representation is available through the admin interface.

delete(*resource_id*)

Return an HTTP response object resulting from a HTTP DELETE call.

Parameters **resource_id** – The value of the resource's primary key

get(*resource_id=None*)

Return an HTTP response object resulting from an HTTP GET call.

If *resource_id* is provided, return just the single resource. Otherwise, return the full collection.

Parameters **resource_id** – The value of the resource's primary key

methods = {'DELETE', 'GET', 'PATCH', 'POST', 'PUT'}

patch(*resource_id*)

Return an HTTP response object resulting from an HTTP PATCH call.

Returns `HTTP 200` if the resource already exists

Returns `HTTP 400` if the request is malformed

Returns `HTTP 404` if the resource is not found

Parameters **resource_id** – The value of the resource's primary key

post()

Return the JSON representation of a new resource created through an HTTP POST call.

Returns `HTTP 201` if a resource is properly created

Returns `HTTP 204` if the resource already exists

Returns `HTTP 400` if the request is malformed or missing data

put (*resource_id*)

Return the JSON representation of a new resource created or updated through an HTTP PUT call.

If resource_id is not provided, it is assumed the primary key field is included and a totally new resource is created. Otherwise, the existing resource referred to by *resource_id* is updated with the provided JSON data. This method is idempotent.

Returns `HTTP 201` if a new resource is created

Returns `HTTP 200` if a resource is updated

Returns `HTTP 400` if the request is malformed or missing data

The *sandman2.exception* module contains an Exception, expressible in JSON, for each HTTP status code. In general, code should raise these exceptions directly rather than making a call to Flask's `abort()` method.

JSON-based Exception classes which generate proper HTTP Status Codes.

exception `sandman2.exception.`**BadRequestException**(*message=None*, *payload=None*)

Raised when a request contains illegal arguments, is missing arguments, can't be decoded properly, or the request is trying to do something that doesn't make sense.

code = 400

exception `sandman2.exception.`**ConflictException**(*message=None*, *payload=None*)

Similar to a ServerErrorException (HTTP 500) but there is some action the client may take to resolve the conflict, after which the request can be resubmitted. A request to reprocess a job not marked for reprocessing, for example, could cause this exception to be raised.

code = 409

exception `sandman2.exception.`**EndpointException**(*message=None*, *payload=None*)

Base class for all Exceptions.

to_dict()

Return a dictionary representation of the exception.

exception `sandman2.exception.`**ForbiddenException**(*message=None*, *payload=None*)

Raised when a request asks us to do something that we won't do because it violates the application logic. *Does not refer to an authentication failure.* Rather, it means the action requested is forbidden by the application.

code = 403

exception `sandman2.exception.`**NotAcceptableException**(*message=None*, *payload=None*)

Raised when the client does not Accept any of the Content-Types we are capable of generating.

code = 406

exception `sandman2.exception.`**NotFoundException**(*message=None*, *payload=None*)

Raised when the endpoint (or a resource it refers to) is not found. Can also be used if a resource referred to in a request (e.g. a specific job in a /job_status request) can not be found.

code = 404

exception `sandman2.exception.`**NotImplementedException**(*message=None*, *payload=None*)

Raised when the application does not implement the functionality being requested. Note that this doesn't refer to an HTTP method not being implemented for a given endpoint (which would be a 405 error).

code = 501

exception `sandman2.exception.`**ServerErrorException**(*message=None*, *payload=None*)

Raised when the application itself encounters an error not related to the request itself (for example, a database error).

code = 500

exception `sandman2.exception.`**`ServiceUnavailableException`**(*message=None*, *payload=None*)

Raised when a resource is temporarily unavailable (e.g. not being able to get a database connection). Setting the *Retry-After* header gives the length of the delay, if it is known. Otherwise, this is treated as a 500 error.

code = 503

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index