JSON Data Ferret

Oct 26, 2022

Contents

CHAPTER 1

Introduction

This is a Django application that manages JSON data with a full history of changes and moderation facilities.

1.1 What this provides

This Django application provides models and helper code to help a Django app manage some data.

There can be several types of data stored.

Each type can have:

- A JSON Schema file against which data will be validated, as a way to provide a web form to edit the data.
- A spreadsheet guide form file that lets users download spreadsheets with the existing data, edit them and import them.

Each type can hold a number of Records. Each record has a public ID (slug type) and one block of JSON.

The system keeps a history of all changes to the data, by way of Events. Each event has one or more Edits attached. Each event can optionally be linked to a user account and have a comment attached.

The system also provides a way for edits to be suggested, and a moderator to approve or reject these edits.

Each Edit can provide a whole new block of JSON to replace the current value with, or a smaller JSON value that will be merged into the current JSON value.

Edit's can be approved straight away, or a future events can approve or reject edits.

1.2 Provided via a web interface

The application provides a admin web interface which any Django user with the correct permission can access.

This gives them access to call any operation on the data and a handy way to look at the current state of the data.

1.3 Provided via Python API's

But it is anticipated most applications will include this code as a library and then provide their own application that will provide user-friendly interfaces and can build meaning on top of the JSON data.

In this repository an example application is included to illustrate this.

CHAPTER 2

Explanation

2.1 Models

2.1.1 Type

The app can process data of several different types.

Types should be created as records in the database.

However most of the configuration of types happens in the Django configuration.

Each type is identified by it's *public_id*, which should be unique.

2.1.2 Record

Each type can have multiple records.

Each record is identified by it's *public_id*, which should be unique in it's type.

Records are not created manually, but are instead created for you as soon as an Edit happens on a new *public_id*.

A record contains only basic information; it is a object/database row for Edits to link to.

It however also contains some columns of cached information, designed to make it easy for other apps to get data from the system.

These are:

- *cached_exists* - boolean. A record is said to exist if there has ever been any data approved for it.
- *cached_data* - JSON. The latest version of the actual data.
- *cached_jsonschema_validation_errors* - If a JSON Schema is specified, this will contain JSON Schema errors for the latest data.

2.1.3 Event

Changes happen to data by way of Events. Think of them as a commit in git.

Each event can be linked to one or more edits in several different ways. See below.

Note:

- an event may only contain new data to be moderated, and thus may not always change the actual data in the system.
- an event may contain edits that are created and approved in the same event - data that was not moderated but approved instantly, in other words.

Each event is identified by it's *public_id*, which should be unique - but these are set automatically for you.

2.1.4 Edit

Each edit contains actual data:

- *mode* - Replace or Merge. This is how the edit is applied to the record.
- *data_key* - Not currently used - leave as the default of /
- *data* - The actual JSON data to merge or replace.

Each edit is linked to at least one Event by:

- *creation_event* - The event that created it. This must be set.

Each edit can also have one (but not both) of these links set:

- *approval_event* - The event that approved this data. (Note this can be the same event as the *creation_event*.)
- *refusal_event* - The event that rejected this data.

Each edit is identified by it's *public_id*, which should be unique - but these are set automatically for you.

CHAPTER 3

How to Guides

3.1 Use as a library in your app

3.1.1 Include library

Add as a requirement:

```
-e git+https://github.com/OpenDataServices/json-data-ferret.git@v0.3.0
↪#egg=jsondataferret
```

(Choosing the version you want)

In your Django settings file, add this to INSTALLED_APPS:

```
INSTALLED_APPS = [
    ...
    "jsondataferret.apps.JsondataferretConfig",
    ...
]
```

3.1.2 Set up Types

Now you need to set up the types you want to use. You do this by creating *Type* models. You can do this by any usual Django means - logging in to admin interface as a super user is probably easiest.

In your Django settings file you may also want to add a *JSONDATAFERRET_TYPE_INFORMATION* setting with extra information. *See the Configuration reference for more*

3.1.3 Use from your custom code

You can now use the Python API and read the models of this library as you require. *See the Python API reference for more*

3.1.4 Web UI

If you want people to be able to use the Web UI, you must first enable it and give the relevant user accounts permission.

In your Django app's urls file add:

```
urlpatterns = [
    ...
    path("jsondataferret/", include("jsondataferret.urls")),
    ...
]
```

You need to set the correct permissions for each user of the web UI. You can do this by any means Django allows - e.g. logging into the admin interface as a superuser. *See the reference for more*

CHAPTER 4

Reference

4.1 Configuration

4.1.1 Types

Some configuration options for each different type can be set in the normal Django configuration.

```
JSONDATAFERRET_TYPE_INFORMATION = {
    "type1": {
        "json_schema": {...},
        "spreadsheet_form_guide": "full filename"
    },
    "type2": {
        "json_schema": {...},
        "spreadsheet_form_guide": "full filename"
        "fields": [...],
    },
}
```

The key for each type (eg *type1*, *type2*) should match the *public_id* field in the *Type* record in the database.

JSON Schema

If this is set,

- Any data will be validated against the JSON Schema and any errors or success will be shown to the user in the Web UI.
- The user will be able to edit the data in a web browser using a JSON Schema widget.

The value should be the actual JSON Schema as a python dictionary. You will probably load it in the settings module.

```
with open(...) as fp:
    org_json_schema = json.load(fp)

JSONDATAFERRET_TYPE_INFORMATION = {
    "org": {
        "json_schema": org_json_schema,
    },
```

This is optional; if not set basic operations will still work.

Spreadsheet Guide Form

If this is set,

- The user will be able to download and import a spreadsheet in the Web UI.

The value should be the filename of the spreadsheet. Ideally make it a absolute filename.

See the documentation for the Spreadsheet forms library for more on guide forms.

This is optional; if not set basic operations will still work.

Fields

If this is set,

- The user will see a list of data from the JSON data presented as fields.

The value should be a list of python dictionaries.

If there is only one value in the data, the dictionary should look like:

```
{"key": "/project_name/value", "title": "Project Name (value)"},
```

The *key* should be the JSON path to the value and the *title* is what is shown to the user.

Where the data contains a list of dictionaries, you can also specify that. In this mode, you specify where the list is in the data then specify fields for each item in the list.

```
{
    "type": "list",
    "key": "/outcomes",
    "title": "Outcomes",
    "fields": [
        {"key": "/outcome", "title": "Outcome"},
        {"key": "/definition", "title": "Definition"},
    ],
},
```

This is optional; if not set basic operations will still work.

4.2 User Accounts

Normal Django user accounts are used.

The permission *Admin - Can Admin All Data Managed by JSON Data Ferret* is required for a user to access the admin web interface for the app. This will give them full permissions to change and moderate data.

This permission is not needed if a user calls some custom code in another app that calls one of the libraries Python API's. In that case, it's up to the calling code to check any user permissions as required.

4.3 Python API's

4.3.1 Read from models directly

There are Django models with information, and currently you are encouraged to read from them to look up certain information.

In particular, the *Record* model contains some cached columns of the latest data.

4.3.2 Write to models directly

You can write to the *Type* model, to set up new types.

Do NOT write to the *Record*, *Event* or *Edit* models directly. Instead, use the Python API's below (in particular *jsondataferret.pythonapi.newevent*) to write new data to the system.

4.3.3 jsondataferret.pythonapi.newevent

The function *newEvent* is used to write new data to the system.

It should be passed:

- *datas* - an array of objects, described below.
- *user* - A Django User
- *comment* - A text comment

The items in the *datas* array should be instances of one of the following classes:

- *NewEventData* - used to add new data to the system.
- *NewEventApproval* - used to approve an edit that has previously been written to the system (moderate it successfully)
- *NewEventRejection* - used to reject an edit that has previously been written to the system (moderate it and fail)

4.3.4 jsondataferret.pythonapi.purge

The function *purge_record* is used to delete a record and all associated data from the system permanently. There is no undo.

4.3.5 jsondataferret.pythonapi.runevents

The function *clear_data_and_run_all_events* clears all caches on Records, then tries to updates them all to the latest value.

This should not have to be run in normal operations, but may be needed to clear a problem.

4.4 Vagrant for Developers

A vagrant box exists to help developers.

Simply run *vagrant up*.

After *vagrant ssh*, run *cd /vagrant* and *source .ve/bin/activate*.

4.4.1 Run Web Server

```
python manage.py runserver 0:8000
```

Go to *http://localhost:8000*

4.4.2 Set up app for the first time

Run normal Django database migrations.

Create a superuser via the normal django command line tool:

    python manage.py createsuperuser

Run the webserver.

Log into */admin*.

Add some *Types* records in the *Jsondataferret* section, for use with the exaple app :

- public id: *project*, title: *Project*
- public id: *org*, title: *Organisation*

4.4.3 Python Packages Upgrade

```
pip-compile --upgrade
pip-compile --upgrade requirements_dev.in
```

4.4.4 Tests

Run tests (with Vagrant DB credentials):

```
JSONDATAFERRET_DATABASE_NAME=test JSONDATAFERRET_DATABASE_USER=test JSONDATAFERRET_
↪DATABASE_PASSWORD=test python manage.py test
```

4.4.5 Code Quality

Clean up code before commit:

```
isort --recursive djangoproject/ jsondataferret jsondataferretexampleapp/ setup.py␣
↪docs/
black djangoproject/ jsondataferret jsondataferretexampleapp/ setup.py docs/
flake8 djangoproject/ jsondataferret jsondataferretexampleapp/ setup.py docs/
```

4.4.6 Reset Database

sudo su postgres psql -c “DROP DATABASE app” psql -c “CREATE DATABASE app WITH OWNER app ENCODING ‘UTF8’ LC_COLLATE=’en_GB.UTF-8’ LC_CTYPE=’en_GB.UTF-8’ TEMPLATE=template0 ” exit python manage.py migrate