Banana Documentation

Release 2.0

Gijs Molenaar, John Swinbank, Tim Staley

January 15, 2016

Contents

Contents:

Introduction

This is Banana - the web frontend for a database populated by the TRAnsients Pipeline (TRAP).

Technically it is a Django project.

CHAPTER 2

Installation

2.1 Requirements

Banana depends on various 3rd party Python libraries which are defined in the **requirements.txt** file. You can install the dependencies using pip:

```
$ pip install astropy
$ pip install -r requirements.txt
```

Pip cant figure out dependencies correctly in some cases, so you need to manually install astropy first.

2.2 Quick configuration

copy the example config file:

```
$ cp project/settings/local_example.py project/settings/local.py
```

Now open `project.settings.local` in your favorite editor and configure your database settings.

2.3 Runnig the server

You can run a Django testing webserver serving the banana project using:

```
$ ./manage.py runserver
```

2.4 Deployment

If you want a more permanent implementation and serve Banana so several users it is adviced to deploy your setup with a dedicated webserver. The Django project itself has extended documentation on how to do this.

CHAPTER 3

Configuration

All settings are defined in the python module `project.settings`. banana first loads `project.settings.base` followed by `project.settings.local`. When you start using banana for the firs time, there will be no **local** file. You should copy `project.settings.local_example` to `project.settings.local` and adjust it to your environment. The **base** should contains all settings which are required by Banana and you should not modify this file. You should override, append or define new settings variables in **local**.

CHAPTER 4

Project Layout

Contents:

4.1 Banana Project

This is the Banana Django Project. At the moment the split between the Banana app and the banana project is quite ambiguous since for now they are never used separately. The Django Project is a placeholder for site specific logic or media types, like custom settings, templates and logo's. The app should contain all code that can be reused in an other Django project, together with other apps with a specific independent purpose.

You can read more about this distinction in the Django documentation resusable apps section.

4.2 Banana

This is the Django Banana app. To learn more about what an app is please read the *Banana Project* section.

4.3 Artwork

This folder contains banana artwork.

4.4 Documentation

the **doc** folder inside the banana project contains a Sphinx documentation project which is used to generate the documentation your are now reading.

CHAPTER 5

Models

banana.models contains the Django ORM models. These Object Oriented representation of a TRAP database. Since you don't initialise the TRAP database using Django, we need to manually keep the Django models in sync with the TRAP schema. Below we describe a procedure on how to do this.

5.1 Updating the model

You need to update the **banana/models.py** file to reflect the new database structure. The easy way to do this is as follows:

- Generate a new database with the schema version you want to upgrade to (using, eg, *tkp-manage.py initdb*). Either MonetDB or Postgres is fine.
- Get a Banana installation which is able to connect to your database. You'll need to edit **project/settings/local.py** to set the appropriate hostname, port and password for MonetDB and/or for Postgres. Banana will build a list of all the databases on the host you specify, based on the assumption that the database name, username and password are all the same.
- Within your banana directory, dump a set of models representing your new database by running:

```
$ ./manage.py inspectdb --database=<dbname> > banana/models_new.py
```

- Using your favourite tool, update **banana/models.py** to reflect the additions in **banana/models_new.py**. Note that **banana/models.py** has a bunch of useful customization which we don't want to lose. **don't** replace it with the new version, but rather carefully compare it with the new models and merge only the relevant changes.
- Update the **schema_version** variable defined in **banana/models.py** to reflect the new schema.
- Check for any templates (stored in **banana/templates**) which are using model fields which you've just removed or renamed, and modify them to use the new models.
- Commit your changes, submit a pull request, and have a cup of tea.

Testing

You should be careful when running the test suite. Default behavior for Django is to take your database configuration (which you defined in `project.settings.local`, append **_test** to the database name and attempt to create and destroy this database configuration. You probably don't want to do this in production. We created a seperate `testing` subproject that takes the **banana** configuration but overrides the database settings to use a safe **sqlite** based database configuration.

6.1 Running the test suite

To run the banana test suite run:

```
$ ./manage.py test --settings=testing.settings
```

6.2 Updating the fixtures

Always regenerate the fixtures when you altered the model. You should do this by populating a TKP database with Mock data.

- (re)create a database
- initialise schema with tkp-manage.py initdb
- run **banana/util/create_content.py** to create mock data. Configure the connection using the TKP_DB* environment variables
- configure the Banana project to use this database
- dump the db content:

```
    $ ./manage.py dumpdata --database=%{TK_DBNAME} --indent=1 banana > testing/fixtures/initial_data
```

6.3 Travis

For every commit to every branch or every issued pull request the travis build system is triggered and will try to run the test suite for that branch. It will update the github status page of the branch or pull request according to the test run output (failed or not).

CHAPTER 7

Banana specific Django View Mixins

To fit our specific requirements we created Django View Mixins that extend the default behavior. They are used in various views in the banana app.

7.1 The mixins

class `banana.views.mixins.`**`DatasetMixin`**

Mixin view that adds the 'dataset' request variable to the context.

class `banana.views.mixins.`**`FluxViewMixin`**

Mixin view that adds the 'flux_prefix' request variable to the context.

class `banana.views.mixins.`**`HybridTemplateMixin`**

Assigns a default `template_name`, and checks the request for a format.

If the format specified in the querystring is json or csv, this will change the `content_type` and `template_name` accordingly.

If template name is not explicitly set, we assign one based on the object or model in the view. We derive the template path as:

<app_label>/<object_name.lower()><template_name_suffix><extension>

where `template_name_suffix` is something like '_list' or '_detail' (inherited from the Django standard class views) e.g.:

banana/extractedsource_list.html

banana/extractedsource_detail.html

class `banana.views.mixins.`**`SortListMixin`**

View mixin which provides sorting for ListView.

CHAPTER 8

Banana and multiple databases

The way we deploy TRAP and Banana at the University of Amsterdam is that various scientists create multiple PostgreSQL and MonetDB databases and populate these with data. We want to be able to visualise the content of all these databases.

We've created various helper functions (`project.settings.database`) that assist in automatically populating the Django configuration with our site specific configuration. It is adviced **not** to use these in production, but rather build a manual configuration.

The database which is used is based on the URL, specifically the URL variable. We've crafted a combination of Django middleware and Django database routing that makes Django use the desired database. Below is the module documentation for that logic.

8.1 Module documentation

Select database based on URL variable

Inspired by this Django snipped.

It's assumed that any view in the system with a cfg keyword argument passed to it from the urlconf may be routed to a separate database. for example:

```
url( r'^(?P<db>\w+)/account/$', 'views.account' )
```

The middleware and router will select a database whose alias is <db>, **default** if no db argument is given and raise a 404 exception if not listed in **settings.DATABASES**, all completely transparent to the view itself.

class `project.multidb.`**`MultiDbRouter`**

The multiple database router.

Add this to your Django database router configuration, for example:

```
DATABASE_ROUTERS += ['project.multidb.MultiDbRouter']
```

class `project.multidb.`**`MultiDbRouterMiddleware`**

The Multidb router middelware.

he middleware process_view (or process_request) function sets some context from the URL into thread local storage, and process_response deletes it. In between, any database operation will call the router, which checks for this context and returns an appropriate database alias.

Add this to your middleware, for example:

```
MIDDLEWARE_CLASSES += ['project.multidb.MultiDbRouterMiddleware']
```

`project.multidb.`**`multidb_context_processor`**(*request*)

This context processor will add a db_name to the request.

Add this to your Django context processors, for example:

```
TEMPLATE_CONTEXT_PROCESSORS +=[
    'project.multidb.multidb_context_processor']
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search