prompt$_{r}esponsesDocumentation$

Release 0.1b

Paul Grau

Dec 01, 2017

Contents

This is a generic implementation of prompts and responses to collect user data. It comes with a set of APIs and integration with Django Rest Framework to make your life easier. It is also extendable to use your own data and algorithms.

CHAPTER 1

Requirements

- Python 2.7, 3.4, 3.5, 3.6
- Django 1.9, 1.10, 1.11

CHAPTER 2

Index

2.1 Installation

Install with pip

```
$ pip install django-prompt-responses
```

Add it to your *INSTALLED_APPS*

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    ...
    'prompt_responses',
    'sortedm2m',  # for the ability to change the order of Prompts in the Django admin
    ...
)
```

Sync your database

```
$ python manage.py migrate prompt_responses
```

Head to the *usage* section for the next steps.

2.2 Usage

2.2.1 Admin

This package comes with integration for Django Admin. It is the easiest way to create new prompts and order them into sets.

In a standard Django installation, the admin views should be automatically registered.

2.2.2 Views

This package includes one view and one mixin you can use to display prompt instances and create responses.

Read more

2.2.3 Models

This package includes four models: Prompt, PromptSet, Response, and Tag.

Read more

2.2.4 Django Rest Framework

To use the included viewsets in your Django Rest Framework API, simply register them in a router like so:

```
from rest_framework import routers
from prompt_responses.viewsets import PromptViewSet, PromptSetViewSet

router = routers.DefaultRouter()
router.register(r'prompts', PromptViewSet)
router.register(r'prompt-sets', PromptSetViewSet)

urlpatterns = [
    url(r'^api/', include(router.urls))
]
```

This offers read-only API endpoints for prompts and prompt sets and one writable endpoint to create responses.

Read more

2.3 Models

This page gives a discription of the implemented models and their relations.

2.3.1 Prompt

class Prompt

A Prompt is an abstract definition of a task or question that will be presented to a user in order to collect responses.

In order to present a Prompt to a user, it needs to be instantiated by calling **:method:‘get_instance()‘**.

In its simplest form, a Prompt defines some *text* and a *type* for either free-form or rating respones.

A more advanced version can be connected to any other type of object. When instantiated, these Prompts will be populated with one object chosen from the set of objects of the defined type. The default implementation returns one random object, but this can be customized.

The most advanced version defines two types of objects and allows for tagging, i.e. the user is asked to rate the relationship between objects of the first set and objects of the second. When instantiated, these Prompts will be populated with one object chosen of the first type and a number of objects of the second type.

Prompts also offer a range of analysis functions. For the sake of readability, these are documented here.

`type`
The type of the Prompt. Currently supported types: likert, openended, tagging

`text`
The text to be displayed to the user. The string can use the *{object}* placeholder which will be replace by the object the prompt is instantiated with.

`scale_min`
For prompts with scale responsed, the minimum value of the scale. Defaults to 1

`scale_max`
For prompts with scale responsed, the maximum value of the scale.

`prompt_object_type`
An object of class ContentType to define the model of objects this prompt will be populated with when instantiated. One prompt instance will be populated with one object of this type. A response will contain a reference to this object.

`response_object_type`
An object of class ContentType to define the model of objects this prompt will be populated with when instantiated. One prompt instance will be populated with a number of objects of this type. A response will create :class:‘Tag‘s with references to these objects.

`get_instance``()`
Instantiate this Prompt. Will get one or more objects, depending on the type of prompt.

Returns `PromptInstance`

`create_response``()`
Convenience function to create (and save) a `Response` for this prompt.

Pass `rating` or `text`, as well as `user` and `prompt_object` as needed.

To save tagging responses, pass *tags=[(object1, rating1), (object2, rating2), ...]* OR alternatively, *tags=[{‘object_id’: id1, ‘rating’: rating2}, ...]*

Note that responses per se are not unique per user (as some experiments might require asking the same question multiple times). Some of the analytics functions offer a *user_unique* parameter to restrict analysis to the user’s latest response only.

In contrast, tags are ensured to be unique for (prompt, user, prompt_object, response_object). If the user tagged this combination before, the Tag will be updated, incl. its response relation (i.e. the original Response object will no longer be associated with this tag).

This method verifies that the objects match the models defined in the `Prompt` and raises a *ValidationException* on a mismatch.

Returns the newly created `Response`

`get_object``()`
Used to determine the object for instantiating this Prompt. The default implementation is to retrieve

a random object from the queryset. You can override this method to customize this. See `Prompt.prompt_object_type`.

get_queryset()
: The queryset from which the object will be drawn when instantiating this Prompt. The default implementation is to return all objects of type `Prompt.prompt_object_type`.

get_response_objects()
: Used to determine the objects for instantiating this Prompt. The default implementation is to retrieve a random object from the queryset. You can override this method to customize this. See `Prompt.response_object_type`.

get_response_queryset()
: The queryset from which the objects will be drawn when instantiating this Prompt. The default implementation is to return all objects of type `Prompt.response_object_type`.

Scales

For prompts that require rating responses, usually you want to confine the acceptable values to a certain scale.

The *Prompt* model offers some utility functions to create arbitrary scales for displaying them in forms.

TODO

PromptInstance

class **PromptInstance**
: A PromptInstance is not a database model, but created on the fly when a prompt is instantiated. It encapsulates the prompt and any object instances that are needed to display it to the user. It only lives for one request.

prompt

Type `Prompt`

object
: An object with which this prompt has been populated. See `Prompt.prompt_object_type`.

response_objects
: A list of objects with which this prompt has been populated. Can be presented for tagging prompts. See `Prompt.response_object_type`.

__str__()
: The string representation of this class is the prompt's text, formatted with the *object*. Useful for directly printing a *prompt_instance* in a template. See `Prompt.text`.

2.3.2 Response

class **Response**
: A Response can have a rating and/or a text. If the prompt has a `prompt_object_type`, the object obtained during instantiation should be saved as prompt_object.

rating

Type integer

text

Type string

prompt_object
Any object that this response is related to. Its type should match `prompt_object_type`.

prompt
The `Prompt` that this response is related to. This is a required field.

user
The user that this response belongs to. This is a required field.

2.3.3 Tag

class **Tag**
User ratings for associations between two objects. Tags are contained in a `Response`. You shouldn't need to create these objects yourself – rather, refer to `Prompt.create_response()`.

response_object
The object that this tag refers to. Should match the type defined in the `Prompt`. When you instantiate a Prompt, this should be one of the instance's `response_objects`. See `Prompt.response_object_type`.

rating

Type integer

response
The `Response` that this tag is related to. This is a required field.

2.3.4 PromptSet

class **PromptInstance**
You can optionally use PromptSets to organize several prompts together. PromptSets have a *name* and contain an ordered list of `Prompt` objects.

name
A name to identify this set. Should be in slug format.

prompts
A many-to-many field to add any number of `Prompts` to this set. Prompts are orderable. If you use the Django admin and added *sortedm2m* to your *INSTALLED_APPS*, the widget should allow drag and drop. See django-sortedm2m's documentation for details about how this works.

2.4 Views

2.4.1 Mixins

class **PromptInstanceMixin**
Provide prompt and prompt_instance to a view.

Tries to get prompt by looking up the pk parameter from the request URL. Override get_prompt() to choose a different way of obtaining the prompt object.

Example usage

```
from prompt_responses.views import PromptInstanceMixin

class MyView(PromptInstanceMixin, View):
    ...
```

The view will have both *prompt* and *prompt_instance* as attributes. They are also added to the template context.

prompt
The *Prompt* that is displayed in this view

prompt_instance
The *PromptInstance* that is displayed in this view

2.4.2 Class-based Views

class **CreateResponseView** (*PromptInstanceMixin, ...*)
A simple view that can display a template with the instantiated prompt and a form to create a response for this prompt.

You can add it as-is to your URL configuration:

```
from prompt_responses.views import CreateResponseView

urlpatterns = [
    url(r'^prompt/(?P<pk>[0-9]+)/$', CreateResponseView.as_view(), name='create_
↪response'),
]
```

Or have a look at the code to get an idea of making your own view.

For example, you can sub-class *CreateResponseView* and override *get_prompt()* to choose a different way of obtaining the prompt object.

This view requires authentication and uses the user from the current request to create the response. You can also use the *BaseCreateResponseView* and provide an alternative *get_user()* method instead.

2.5 Django Rest Framework

To use the included viewsets in your Django Rest Framework API, simply register them in a router like so:

```
from rest_framework import routers
from prompt_responses.viewsets import PromptViewSet, PromptSetViewSet

router = routers.DefaultRouter()
router.register(r'prompts', PromptViewSet)
router.register(r'prompt-sets', PromptSetViewSet)

urlpatterns = [
    url(r'^api/', include(router.urls))
]
```

This offers read-only API endpoints for prompts and prompt sets and one writable endpoint to create responses.

The API endpoints are hyperlinked together, i.e. they return URLs to other resources. It is recommended to follow these links instead of constructing your own URLs.

2.5.1 Prompt API

Get a list of all prompts:

```
GET api/prompts/
```

Get details about a prompt:

```
GET api/prompts/<prompt_id>/
```

Get an instance of a prompt:

```
GET api/prompts/<prompt_id>/instantiate/
```

Get an instance of a prompt within the context of a prompt set:

```
GET api/prompts/<prompt_id>/instantiate/<prompt_set_name>/
```

2.5.2 Create Response API

Save a response for a prompt:

```
POST api/prompts/<prompt_id>/create-response/
```

This endpoint expects the following data:

TODO

2.5.3 PromptSet API

Get a list of all prompt sets:

```
GET api/prompt-sets/
```

Get details about a prompt set:

```
GET api/prompt-sets/<prompt_set_name>/
```

Traversing an ordered list of prompts

When you use prompt sets, you can follow the links returned in the responses to traverse the list of prompts. Both PromptSet and Prompt API responses will contain a *next_prompt_instance* URL.

2.6 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.6.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/graup/django-prompt-responses/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

prompt_responses could always use more documentation, whether as part of the official prompt_responses docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/graup/django-prompt-responses/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.6.2 Get Started!

Ready to contribute? Here's how to set up *django-prompt-responses* for local development.

1. Fork the *django-prompt-responses* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-prompt-responses.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-prompt-responses
$ cd django-prompt-responses/
$ python setup.py develop
```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 prompt_responses tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

2.6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/graup/django-prompt-responses/pull_requests and make sure that the tests pass for all supported Python versions.

2.6.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_prompt_responses
```

2.7 History

2.7.1 0.1.0 (2017-11-07)

- First release on PyPI.

Index