django-fbv

Release 0.5.1

Adam Hill

Jan 01, 2023

CONTENTS

`django-fbv` includes utilities to make function-based views cleaner, more efficient, and better tasting.

CHAPTER

ONE

WHY?

The Django community continues to be split about whether to use function-based views or class-based views. This library is intended to provide solutions that address some of the annoyances of function-based views.

If you want to read a more detailed critique of class-based views, https://spookylukey.github.io/django-views-the-right-way/ is excellent.

CHAPTER

TWO

INSTALLATION

`poetry add django-fbv` OR `pip install django-fbv`

The *decorators* and *views* can be used by just importing them. The middleware *needs to be installed like typical Django middleware*.

CHAPTER

THREE

FEATURES

3.1 decorators

- `fbv.decorators.render_html`: decorator to specify the template that a view function response should use when rendering
- `fbv.decorators.render_view`: decorator to specify the template and content type that a view function response should use when rendering
- `fbv.decorators.render_json`: decorator to automatically render dictionaries, Django Models, or Django QuerySets as a JSON response

3.2 views

- `fbv.views.html_view`: directly render a template from `urls.py`
- `fbv.views.favicon_file`: serve an image file as the favicon.ico
- `fbv.views.favicon_emoji`: serve an emoji as the favicon.ico

3.3 middleware

- `fbv.middleware.RequestMethodMiddleware`: adds a boolean property to the `request` for the current request's HTTP method

CHAPTER

FOUR

PRIOR ART

- The `render_view` decorator was forked from `render_to` in the delightful https://github.com/skorokithakis/django-annoying library.
- The `favicon_file` and `favicon_emoji` code is from the superb https://adamj.eu/tech/2022/01/18/how-to-add-a-favicon-to-your-django-site/ blog post.

4.1 Decorators

4.1.1 render_html

Decorator that provides a convienent way to render HTML from a function-based view.

`render_html` with `template_name` argument in `views.py`

```
from fbv.decorators import render_html

@render_html("sample-html-template.html")
def sample_html_view(request):
    return {"data": 123}
```

`render_html` with `TEMPLATE` dictionary key in `views.py`

```
from fbv.decorators import render_html

@render_html()
def sample_html_view(request):
    return {"TEMPLATE": "sample-html-template.html", "data": 123}
```

`render_html` with no template specified in `views.py`

This will use the current module and function name as the template name.

For example, the following would look for a `views/sample_html_view.html` template.

```
from fbv.decorators import render_html

@render_html()
def sample_html_view(request):
    return {"data": 123}
```

Note: `render_html` is just an alias for `render_view` below that sets the content type to `text/html; charset=utf-8`.

4.1.2 render_view

Includes all the functionality of the `html_view`, but allows the response content type to be set.

`render_view` in `views.py`

```
from fbv.decorators import render_view

@render_view("sample-xml-template.xml", content_type="application/xml")
def sample_xml_view(request):
    return {"data": 123}
```

4.1.3 render_json

Decorator that provides a convienent way to return a `JSONResponse` from a function-based view. `dictionary`, Django `Model`, and Django `QuerySet` objects are all rendered automatically by `render_json`.

Note: By default, the rendered JSON won't have whitespaces between keys and values for the most compact representation as possible. However, you can override that functionality by passing in a `tuple` as `(item_separator, key_separator)`.

`render_json` in `views.py`

```
from fbv.decorators import render_json

@render_json()
def sample_json_view(request):
    return {"data": 123, "test": 456}
```

Default JSON response

```
{"data":123,"test":456}
```

`render_json` with `separators` in `views.py`

```
from fbv.decorators import render_json

@render_json(separators=(", ", ": "))
def sample_json_view(request):
    return {"data": 123, "test":456}
```

Separators JSON response

```
{"data": 123, "test": 456}
```

dictionary

render_json in views.py

```
from fbv.decorators import render_json

@render_json()
def sample_json_view(request):
    return {"data": 123}
```

dictionary JSON response

```
{ "data": 123 }
```

Django Model

render_json Django Model in views.py

```
from django.contrib.auth.models import User
from fbv.decorators import render_json

@render_json()
def sample_json_model_view(request):
    user = User.objects.get(id=1)

    return user
```

Django Model JSON response

```
{
  "pk": 1,
  "username": "testuser1",
  "first_name": "Test 1",
  "last_name": "User 1",
  "email": "testuser1@test.com"
}
```

Specifying model fields

To only return some of the model fields, pass in a `fields` kwarg with a `tuple` of field names.

render_json Django Model in views.py

```
from django.contrib.auth.models import User
from fbv.decorators import render_json

@render_json(fields=("username", ))
def sample_json_model_view(request):
    user = User.objects.get(id=1)

    return user
```

Django Model JSON response

```
{
  "username": "testuser"
}
```

Django QuerySet

render_json Django QuerySet in views.py

```
from django.contrib.auth.models import User
from fbv.decorators import render_json

@render_json()
def sample_json_queryset_view(request):
    users = User.objects.all()

    return users
```

Django QuerySet JSON response

```
[
  {
    "pk": 1,
    "username": "testuser1",
    "first_name": "Test 1",
    "last_name": "User 1",
    "email": "testuser1@test.com"
  },
  {
    "pk": 2,
    "username": "testuser2",
    "first_name": "Test 2",
    "last_name": "User 2",
    "email": "testuser2@test.com"
  }
]
```

Specifying QuerySet model fields

To only return some of the QuerySet's model fields, pass in a `fields` kwarg with a `tuple` of field names.

render_json Django QuerySet with fields in views.py

```
from django.contrib.auth.models import User
from fbv.decorators import render_json

@render_json(fields=("username",))
def sample_json_queryset_view(request):
    users = User.objects.all()

    return users
```

Django QuerySet with fields JSON response

```
[
  {
    "username": "testuser1"
  },
  {
    "username": "testuser2"
  }
]
```

Using `QuerySet.values()`

To only return some of the QuerySet's model fields, call `QuerySet.values()` with the field names.

render_json Django QuerySet.values() in views.py

```
from django.contrib.auth.models import User
from fbv.decorators import render_json

@render_json()
def sample_json_queryset_view(request):
    users = User.objects.all().values("first_name")

    return users
```

Django QuerySet.values() JSON response

```
[
  {
    "first_name": "Test "
  },
  {
    "first_name": "Test 2"
  }
]
```

4.2 Views

4.2.1 html_view

Directly render a template from `urls.py`, similar to using the generic `TemplateView`.

```
# urls.py
from fbv.views import html_view

urlpatterns = (
    path("sample-html-view", html_view, {"template_name": "sample-html-view-template.html
↪"}),
)
```

4.2.2 redirect_view

Redirect to a named pattern from `urls.py`, similar to using the `RedirectView`. Can also specify whether the redirect is permanent or not (i.e. a 301 or 302). Defaults to a 302.

```
# urls.py
from fbv.views import redirect_view

urlpatterns = (
    path("sample-html-view", redirect_view, {"pattern_name": "some-pattern-name"}),
    path("another-html-view", redirect_view, {"pattern_name": "another-pattern-name",␣
↪permanent=True}),
)
```

4.2.3 favicon_file

Serves an image file as `favicon.ico`.

Note: Even though the extension is `ico`, browsers will use other image formats. `PNG` images are widely supported and work great for this purpose. More details at: https://adamj.eu/tech/2022/01/18/how-to-add-a-favicon-to-your-django-site/#what-the-file-type.

```
# urls.py
from fbv.views import favicon_file

urlpatterns = (
    path("favicon.ico", favicon_file, {"file_path": "static/img/favicon.png"}),
)
```

Note: `file_path` is relative to Django's `settings.BASE_DIR` path. i.e. the example above would use the file located at `/www/sample-project/static/img/favicon.png` if `settings.BASE_DIR` is `Path("/www/sample-project/")`.

4.2.4 favicon_emoji

Serves an emoji as `favicon.ico`.

Note: Even though the extension is `ico`, browsers will use other image formats. The emoji will be rendered as an `SVG` which is supported by most modern browsers (other than Safari). More details at: https://adamj.eu/tech/2022/01/18/how-to-add-a-favicon-to-your-django-site/#what-the-file-type.

```
# urls.py
from fbv.views import favicon_emoji

urlpatterns = (
    path("favicon.ico", favicon_emoji, {"emoji": ""}),
)
```

4.3 Middleware

4.3.1 RequestMethodMiddleware

Adds a boolean property to the `request` for the current request's HTTP method.

Installation

Add `"fbv.middleware.RequestMethodMiddleware"` to the `MIDDLEWARE` list in `settings.py`. It doesn't matter where in the list it is added, but it probably shouldn't be first.

```
# settings.py
MIDDLEWARE = [
    # other middleware
    "fbv.middleware.RequestMethodMiddleware",
    # other middleware
]
```

`request` properties

Once the middleware is installed every `request` object will now have a boolean property for each of the following HTTP methods:

- `is_post`
- `is_get`
- `is_patch`
- `is_head`
- `is_put`
- `is_delete`
- `is_connect`
- `is_trace`

```
# views.py
from fbv.decorators import render_html

@render_html("sample-html-template.html")
def sample_html_view(request):
    if request.is_get:  # instead of `request.method == "GET"`
        return {"http_method": "GET"}
    elif request.is_post:  # instead of `request.method == "POST"`
        return {"http_method": "POST"}
```