django-sticky-uploads Documentation

Release 1.0.0

Caktus Consulting Group

Mar 29, 2018

Contents

django-sticky-uploads is a progressively enhanced file input widget for Django which uploads the file in the background and also retains value on form errors.

CHAPTER 1

Requirements/Installing

django-sticky-uploads requires requires Django 1.11 or 2.0, and a Python that is supported by the chosen version of Django.

The easiest way to install django-sticky-uploads is using pip:

```
pip install django-sticky-uploads
```

CHAPTER 2

Browser Support

This project makes use of progressive enhancement meaning that while all browsers are supported, they will not all have the same user-experience. If the browser does not support the necessary client-side features then it will fall back to the default file upload behavior.

The primary HTML5 dependencies are File API and XHR2 meaning that the following desktop/mobile browsers should get the enhanced experience:

- Chrome 13+
- Firefox 4+
- Internet Explorer 10+
- Safari 6+
- Opera 12+
- iOS Safari 6+
- Android Brower 3+
- Blackberry Broswer 10+
- Opera Mobile 12+
- Chrome for Android 27+
- Firefox for Android 22+

CHAPTER 3

Documentation

Additional documentation on using django-sticky-uploads is available on Read The Docs.

CHAPTER 4

Running the Tests

You can run the tests with via:

```
tox
```

(Possibly after installing `tox` with `pip install tox` or alternative.)

CHAPTER 5

License

django-sticky-uploads is released under the BSD License. See the LICENSE file for more details.

CHAPTER 6

Contributing

If you think you've found a bug or are interested in contributing to this project check out django-sticky-uploads on Github.

Development sponsored by Caktus Consulting Group, LLC.

CHAPTER 7

Contents

7.1 Getting Started with django-sticky-uploads

This will walk you through the basics of getting started with django-sticky-uploads. It assumes that you have already installed django-sticky-uploads via:

```
pip install django-sticky-uploads
```

and have an existing project using a compatible version of Django and Python.

7.1.1 Necessary Settings

After installing you should include `stickyuploads` in your `INSTALLED_APPS` setting. To use the default upload view you must also be using `contrib.auth` to manage users.

```
INSTALLED_APPS = (
    # Required by stickyuploads
    'django.contrib.auth',
    # Required by contrib.auth
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    # Other apps go here
    'stickyuploads',
)
```

This is required so that the built-in `contrib.staticfiles` can find the JS included in the django-sticky-uploads distribution. If you are not using `contrib.staticfiles` then this step is not required but you are on your own to ensure the static files are included correctly.

7.1.2 Including the URLs

django-sticky-uploads includes views for accepting the AJAX file uploads. You'll need to include these in your url patterns:

```
from django.conf.urls import patterns, include, url


urlpatterns = patterns('',
    # Other url patterns go here
    url(r'^sticky-uploads/', include('stickyuploads.urls')),
)
```

The `sticky-uploads/` is there for example purposes and you are free to change it to suit your own needs.

7.1.3 Including the JS

The enhanced upload widget requires a small piece of JavaScript to handle the background upload. Each of the cases assume that you are using `contrib.staticfiles` to manage static dependencies.

First, you can add the following script tag to any page which will use the widget.

```
{% load static from staticfiles %}
<script type="text/javascript" src="{% static 'stickyuploads/js/django-uploader.js' %}
↪"></script>
```

Alternatively, you can minimize the JavaScript and load that, or bundle it with other JavaScript for the page.

Yet another option is to include `{{ form.media }}`, where `form` is whatever form is using the upload widget. The widget includes an inner Media class that lists `'stickyuploads/js/django-uploader.js'` as a dependency, and including `{{ form.media }}` in the template will produce the necessary markup to load it.

7.1.4 Adding the Widget

The final step to use django-sticky-uploads is to use the widget on an existing form with a `FileField`. The `StickyUploadWidget` is a drop-in replacement for the default `ClearableFileInput` and can be used on any Django `Form` including ‘‘ModelForm‘‘s.

```
from django import forms

from stickyuploads.widgets import StickyUploadWidget


class ExampleForm(forms.Form):
    upload = forms.FileField(widget=StickyUploadWidget)
```

Note that to make use of the background upload, the user must be authenticated, so the `StickyUploadWidget` should only be used on forms/views where the user is authenticated.

7.1.5 Next Steps

There are hooks on both the client side and server side for customizing the behavior of the uploads. Continue reading to see how you can adjust the default settings to fit your needs.

7.2 Example project

The source tree contains an example Django project using django-sticky-uploads in the examples directory. It's a quick way to try out django-sticky-uploads.

- Clone the repository locally and change directories into the source tree:

```
$ git clone https://github.com/caktus/django-sticky-uploads
$ cd django-sticky-uploads
```

- Create a virtualenv:

```
$ mkvirtualenv sticky-example
```

- Install django-sticky-uploads and django:

```
$ add2virtualenv .
$ pip install Django
```

- Change into the 'example' directory:

```
$ cd example
```

- Run migrations:

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

- Create a user:

```
$ python manage.py createsuperuser
```

- Run the server:

```
$ python manage.py runserver
```

- Visit http://127.0.0.1:8000/ in a browser.
- Login
- Experiment with the file upload form
- Use the admin at http://127.0.0.1:8000/admin/main/saveduploadd/ to see the files uploaded in the background, and look for them in the `media/uploads` directory.

7.3 Customizing the Server Side

django-sticky-uploads ships with a default view for handling the background file uploads, but you may need or want to customize the behavior such as where files are stored or which users are allowed to upload files.

7.3.1 Changing the Storage

For managing the file uploads, django-sticky-uploads uses the File storage API. This allows you to use any valid storage backend for handing the files. By default the view will use `stickyuploads.storage.TempFileSystemStorage`. This is a subclass of the built-in default `FileSystemStorage` with a few changes.

First the files are stored in `/tmp` (or OS equivalent temp directory) rather than `MEDIA_ROOT`. This storage does not expose a url to serve the temporarily uploaded files.

Note: If you are using a multi-server environment this default will not work for you unless you are able have the load balancer pin the consecutive requests to the same backend server or have the temp directory mounted on a network share available to all backend servers.

The storage used by the upload view is configured by the `storage_class` attribute. This should be the full Python path to the storage class. This can be changed by either sub-classing `stickyuploads.views.UploadView` or by passing it as a parameter to `as_view`.

```
# New view to use S3BotoStorage from django-storages

from stickyuploads.views import UploadView

urlpatterns = patterns('',
    url(r'^custom/$',
        UploadView.as_view(storage_class='storages.backends.s3boto.S3BotoStorage'),
        name='sticky-upload-custom'),
)
```

Note: The storage backend you use should not take any arguments in the `__init__` or should be able to be used with the default arguments.

7.3.2 Changing Allowed Users

By default the `UploadView` will only allow authenticated users to use the background uploads. If you would like to change this restriction then you can subclass `UploadView` and override the `upload_allowed` method.

```
from stickyuploads.views import UploadView


class StaffUploadView(UploadView):
    """Only allow staff to use this upload."""

    def upload_allowed(self):
        return self.request.user.is_authenticated() and self.request.user.is_staff
```

7.3.3 Pointing the Widget to the Customized View

By default the `StickyUploadWidget` will use a view named `sticky-upload-default` for its uploads. If you want to change the url used you can pass the url to the widget.

```
from django import forms
from django.urls import reverse_lazy

from stickyuploads.widgets import StickyUploadWidget


class ExampleForm(forms.Form):
    upload = forms.FileField(widget=StickyUploadWidget(url=reverse_lazy('sticky-
↪upload-custom')))
```

You may also choose to not use the default url patterns and name your own view `sticky-upload-default` in which case that url will be used by default.

7.4 Customizing the Client Side

The uploader has a number of hooks to add additional validation or interactions in the browser.

7.4.1 Accessing the uploader

When the uploader is bound to a file input, it is stored on the element as a property named `djangoUploader` during django-stick-uploads' initialization.

```
var myfield = document.querySelector('input[type=file]#some_id');
var uploader = myfield.djangoUploader;
```

The django-sticky-uploads initialization happens after the DOM has been loaded. A good way to run your own code after that is to load your own code after django-sticky-uploads, and arrange for it also to run after the DOM has been loaded; it should then run after django-sticky-uploads.

You can check whether the uploader is enabled for the current browser with the `enabled` function.

```
console.log(uploader.enabled());
```

(Being "enabled" means the current browser supports the standard features for uploading files that django-sticky-uploads needs.)

7.4.2 AJAX Hooks

There are 3 hooks for interacting with the uploader in the life cycle of a new upload request: `before`, `success`, and `failure`. All of these callbacks are given the scope of the uploader. That is, `this` will access the uploader inside of the callback. Each of these callbacks is set by assigning to `uploader.options`.

before

The `before` function, if set, is called when the file input has been changed, and is passed a single argument which is the file data. You may use this hook to do any validations on the file to be uploaded. If the `before` callback returns `false`, it will prevent the upload. An example is given below:

```
var uploader = myfield.djangoUploader;
uploader.options.before = function (file) {
    if (file.size > 1024 * 1024 * 2) {
        // This file is too big
        return false;
    }
};
```

Note: While this hook can be used to do some basic validations, since it is controlled on the client it can be circumvented by a truly malicious user. Any validations should be replicated on the server as well. This should primarily be used for warnings to the user that data they are about to submit is not going to be valid.

success

The `success` callback is called when the server has completed a successful upload. Successful in this case means that the server gave a 2XX response which could include the case where the server did not validate the file which was uploaded. A successful server response will contain the following info:

```
{
    'is_valid': true, // Response was valid
    'filename': 'filename.txt', // File name which was uploaded
    'url': '', // URL (if any) where this file can be accessed
    'stored': 'XXXXXX' // Serialized stored value
}
```

All callbacks should first check for `is_valid` before continuing any other processing. The other keys are not included when the upload is not valid.

```
var uploader = myfield.djangoUploader;
uploader.options.success = function (response) {
    if (response.is_valid) {
        // Do something
    } else {
        // Do something else
    }
};
```

failure

The `failure` callback is called when the server has returned a 4XX or 5XX response. This might be caused by the user not having permission to do the upload or a server timeout. The callback is given the server response.

```
var uploader = myfield.djangoUploader;
uploader.options.failure = function (response) {
    // Do something
};
```

7.4.3 Handling the Form Submit

Because the file is being uploaded in the background while the user processes the rest of the form, there is a case where the file upload has not completed but the user has submitted the form. In this case the default behavior of the plugin is to abort upload request and submit the form as normal. This means at least part of the file will have been uploaded twice and the effort in the background upload is wasted.

If you choose, you can handle this case differently using the `submit` callback. This callback is passed a single argument which is the form submit event. One example of using this option is given below:

```
var uploader = myfield.djangoUploader;
uploader.options.submit = function (event) {
    var self = this, callback;
    if (this.processing) {
        // Prevent submission
        event.preventDefault();
        var form = event.target;
        callback = function () {
            if (self.processing) {
                // Wait 500 milliseconds and try again
```

```
                setTimeout(callback, 500);
            } else {
                // Done processing so submit the form
                form.submit();
            }
        };
        // Wait 500 milliseconds and try again
        setTimeout(callback, 500);
    }
};
```

7.5 Security Considerations

Any time you allow users to upload files to your web server, you have a potential security hole. This is the case whether you use django-sticky-uploads or not. Below are some things to keep in mind when setting up your project to use django-sticky-uploads. Additionally you should read the notes on Unrestricted File Uploads from the OWASP project for more information on the potential risks and mitigations.

7.5.1 Project Internals

By default django-sticky-uploads takes the follow steps to avoid some of the largest problems with unrestricted file uploads. First, it only allows authenticated users to upload files through the background API. Second, it leverages the existing CSRF protections in Django to help ensure that a user's credentials cannot be used to upload files without their knowledge. Additionally, the temporary uploaded files are stored in the system temp directory and should not be exposed by the webserver until the original form has had a chance to validate the file.

The serialization used for the stored file references uses the cryptographic signing utilities included in Django. This prevents the client from manipulating the value when it is available on the client. This relies on keeping your `SECRET_KEY` a secret. In the case that your `SECRET_KEY` is changed it will invalidate any serialized references used by django-sticky-uploads.

7.5.2 External Measures

In addition to the builtin protections provided by Django and django-sticky-uploads, you can also take steps in configuring your webserver to mitigate possible attacks. These include:

- Limiting the file size of the allowed uploads
- Rate-limiting how often a user is allowed to upload files
- Do not allow "execute" permissions in the uploaded directory
- Installing a virus scanner on the server

More details can be found on the OWASP site.

7.6 Release History

7.6.1 v1.0.0 (Released 2018-03-29)

- Add support for Django 2.0 (Python 3 only)

- Drop support for Django 1.8 and 1.10.
- Result: support for Django 1.11 and 2.0

7.6.2 v0.6.1 (Released 2017-11-27)

- Fix link to docs in README

7.6.3 v0.6.0 (Released 2017-11-14)

- Remove dependency on jQuery.
- *Backwards Incompatible*: there are changes to the interface for customizing how the uploads work. See docs/plugin.rst.

7.6.4 v0.5.0 (Released 2017-11-01)

- Add support for Python 3.5, 3.6
- Drop support for Python 3.2, 3.3
- Add support for Django 1.10, 1.11
- Drop support for Django 1.9 and Django older than 1.8

7.6.5 v0.4.0 (Released 2015-06-15)

- Do not display link for temporary uploads (supported on Django 1.6+)
- Dropped testing support for Python 2.6
- Added testing for Django 1.8
- Updated bundled jQuery version to 1.11.3

7.6.6 v0.3.0 (Released 2014-05-23)

- Added upload progress indicator
- Fixed support for Django 1.7
- Upgraded bundled jQuery version to 1.11.1

7.6.7 v0.2.0 (Released 2013-07-23)

- Security issue related to client changing the upload url specified by the widget for the upload
- Added documentation for plugin extensions and callbacks
- *Backwards Incompatible*: The signatures of the internal `UploadForm.stash`, `serialize_upload`, `deserialize_upload` and `open_stored_file` now require the upload url

7.6.8 v0.1.0 (Released 2013-07-19)

Initial public release includes:

- `StickyUploadWidget` as replacement widget for any `FileField`
- jQuery plugin to process uploads in the background
- Server-side code to process/store temporary uploads
- Full test suite with Travis CI integration
- Documentation covering installation, customization and security notes on Read the Docs
- Example project

CHAPTER 8

Indices and tables

- genindex
- modindex
- search