Django File Picker Documentation

Release 0.5

Caktus Consulting Group LLC

Nov 06, 2017

Contents

django-file-picker is a pluggable Django application used for uploading, browsing, and inserting various forms of media into HTML form fields.

When a user is editing some content, and they want to insert a link to a file or image into their content, possibly uploading the image or file in the process, django-file-picker provides the tools to do that.

Using jQuery Tools, django-file-picker integrates seamlessly into pre-existing pages by installing an overlay that lists file details and, when applicable, image thumbnails. New files can also be uploaded from within the overlay (via AJAX Upload).

django-file-picker provides a few optional extensions to help get started, including `file_picker.uploads`, an app with pre-built Image and File models, and `file_picker.wymeditor`, an app that integrates with WYMeditor, a web-based WYSIWYM (What You See Is What You Mean) XHTML editor. These extensions are provided for convenience and can easily be replaced by custom modules.

For more complete documentation, see http://django-file-picker.readthedocs.org

CHAPTER 1

Dependencies

1.1 Required

- Python 2.7, 3.4, 3.5 and 3.6
- Django 1.8 to 1.11 (inclusive)
- sorl-thumbnail==12.4a1
- jQuery 1.4.x
- jQuery Tools 1.2.x
- AJAX Upload (included)

1.2 Optional

- django.contrib.staticfiles
- WYMeditor 0.5

  If you are using `django.contrib.staticfiles`, then add `file_picker` to your INSTALLED_APPS to include the related css/js.

  Otherwise make sure to include the contents of the static folder in your project's media folder.

CHAPTER 2

Basic Installation

1. Add `file_picker` and `sorl.thumbnail` to INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = (
    'file_picker',
    'file_picker.uploads', # file and image Django app
    'file_picker.wymeditor', # optional WYMeditor plugin
    'sorl.thumbnail',  # required
)
```

   `file_picker.uploads` will automatically create two pickers named 'images' and 'files'.

Note: sorl-thumbnail was out of support for a number of years, but there is now a new pre-release that seems to work, which you can install using:

```
$ pip install --pre sorl-thumbnail==12.4a1
```

Hopefully newer releases will come soon.

1. Add the `file_picker` URLs to urls.py, e.g.:

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

urlpatterns = [
    # ...
    url(r'^file-picker/', include(file_picker.site.urls)),
    # ...
]
```

Development sponsored by Caktus Consulting Group, LLC..

CHAPTER 3

Contents

3.1 Basic Setup

Here's an example of how to support uploading and linking to files and images when editing a text field in a model while in the Django admin interface.

1. Use or create a model to contain the text field(s) to be edited by the user. Here we will use the Post model from `sample_project.article`. It has two text fields, Body and Teaser.
2. The files and images are tracked using their own models. For simplicity here we will use the models in `file_picker.uploads`: Image and File. (You won't see them mentioned in the code below - more on that shortly.)
3. To use the pickers on both the teaser and body fields, use a *formfield_override* in the model's admin class to override the widget with the `file_picker.widgets.SimpleFilePickerWidget`:

```
import file_picker
from django.contrib import admin
from sample_project.article import models as article_models

class PostAdmin(admin.ModelAdmin):
    formfield_overrides = {
        models.TextField: {
            'widget': file_picker.widgets.SimpleFilePickerWidget(pickers={
                'image': "images", # a picker named "images" from file_picker.
↪uploads
                'file': "files", # a picker named "files" from file_picker.uploads
            }),
        },
    }

    class Media:
        js = ("http://cdn.jquerytools.org/1.2.5/full/jquery.tools.min.js",)

admin.site.register(article_models.Post, PostAdmin)
```

There's a lot going on behind the scenes here to make this work. Some of it:

- `file_picker/uploads/file_pickers.py` defines two forms, defines two picker objects to use those forms, and then registers those two pickers to be used with the `Image` and `File` models mentioned earlier.
- `file_picker/uploads/file_pickers.py` needs to be imported for its code to run. That happens because 'file_picker.uploads' has been added to `INSTALLED_APPS` during the basic installation of django-file-picker.

3.1.1 Simple File Picker Widget

class `file_picker.widgets.`**`SimpleFilePickerWidget`**

To use the simple file picker widget, override the desired form field's widget. It takes a dictionary with expected keys *"image"* and/or *"file"* which define which link to use, i.e. "Add Image" and/or "Add File".

3.2 Detailed Instructions

The situation where django-file-picker is useful is when you have a user editing a text field in a form, and you want them to be able to insert the paths to images and/or files, either ones that have previously been uploaded, or new ones they upload at the time.

So to start with, you'll have a form with a text field:

```
class MyForm(forms.Form):
    my_text = forms.CharField(widget=forms.TextArea)
```

To insert django-file-picker, we'll tell Django we want to use a different widget for the `my_text` field. This widget will be an instance of `file_picker.widgets.FilePickerWidget` or a subclass. For this example, we'll use `file_picker.widgets.SimpleFilePickerWidget` because it has some buit-in appearance things already set, but later you can customize the appearance by using your own widget:

```
from django import forms
from file_picker.widgets import SimpleFilePickerWidget

class MyForm(forms.Form):
    my_text = forms.CharField(
        widget = SimpleFilePickerWidget(...)
```

File picker widgets have a required argument, `pickers`, which should be a dictionary with keys `"image"`, `"file"`, or both:

```
from django import forms
from file_picker.widgets import SimpleFilePickerWidget

class MyForm(forms.Form):
    my_text = forms.CharField(
        widget = SimpleFilePickerWidget(
            pickers = {
                'image': 'images',
                'file': 'files',
        )
```

The values of the items in the pickers dictionary are the names under which pickers have previously been registered. In this case, we're relying on the knowledge that we've added `file_picker.uploads` to our `INSTALLED_APPS`, and the uploads app's file_picker.py file registered two pickers. That code looks like this:

```
import file_picker

file_picker.site.register(Image, ImagePicker, name='images')
file_picker.site.register(File, FilePicker, name='files')
```

That raises the question, what is a picker? A picker is a previously registered name that has linked to it a model and a subclass of `file_picker.ImagePickerBase` or `file_picker.FilePickerBase`. In the code above, `Image` and `File` are models, and the two `XxxxPicker` identifiers name such classes.

Let's look first at the models:

```
class File(BaseFileModel):
    """ Basic file field model """
    file = models.FileField(upload_to='uploads/files/')


class Image(BaseFileModel):
    """ Basic image field model """
    file = models.ImageField(upload_to='uploads/images/')
```

We can see that each one has a `file` column with a Django FileField or ImageField. They both inherit from abstract model `BaseFileModel`, which adds a bunch of handy fields like `name`, `description`, `file_type`, etc.

The purpose of the models is to track the uploaded files, and this looks reasonable for that. But how does django-file-picker use these models? Or in other words, how much of this code is required by django-file-picker and how much can we do what we like with?

For that, we need to look at the picker classes. Here's what the `FilePicker` class looks like:

```
class FilePicker(file_picker.FilePickerBase):
    form = FileForm
    columns = ('name', 'file_type', 'date_modified')
    extra_headers = ('Name', 'File type', 'Date modified')
```

The `columns` are field names from the file tracking model, and the `extra_headers` the corresponding headers for those fields. So it seems reasonable to guess that the picker widget is going to display those columns when it is listing the uploaded files, and we can see that that part is completely customizable.

There's more about pickers in the docs elsewhere - look for the part about writing custom pickers.

What about the form? You can leave the `form` off of your picker class and file-picker will generate a form on the fly, but in this case, the code provides its own form. Here's `FileForm`:

```
class FileForm(forms.ModelForm):
    file = forms.CharField(widget=forms.widgets.HiddenInput())

    class Meta(object):
        model = File
        fields = ('name', 'description')

    def save(self, commit=True):
        image = super(FileForm, self).save(commit=False)
        # Strip any directory names from the filename
        file_path = os.path.basename(self.cleaned_data['file'])
        fh = ContentFile(open(self.cleaned_data['file'], 'rb').read())
        image.file.save(file_path, fh)
        if commit:
            image.save()
        return image
```

This is more complicated. This is the form that will be used when the user wants to upload a new file. From `Meta.fields`, we can see that the user will only have to enter the name and description. But there's also a hidden `file` field, of type CharField. It looks like this is going to end up with a path to a temporary file, and when the form is saved, it'll copy that into the file field of the new model instance and save it.

The `ImageForm` is practically identical apart from the model it's associated with. I don't know why all the common code isn't factored out into a base form class provided by file_picker.

Again, you don't have to write your own form class if you don't want to. You can leave out the form attribute when creating your picker class, and it'll generate the necessary form on the fly.

Your template must load jquery, jquery-ui, and jquery.tools, as well as any media associated with your form by file-picker. Your head section might look like:

```
<head>
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
  <script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>
  <script src="http://cdn.jquerytools.org/1.2.5/full/jquery.tools.min.js"></script>
  {{ form.media }}
</head>
```

Then you can use your form in your template in the usual way. You can start with:

```
<form action="." method="POST" >
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Submit</button>
</form>
```

For more information, I recommend reading the code of django-file-picker, especially the `uploads` app, `forms.py` which has the base class used to generate new forms for pickers, and the `views.py` file which handles the uploads.

3.3 The Uploads App

The uploads app is designed to make it easy to get File Picker up and running without having to add models or register them with `file_picker`. The uploads app includes two simple pickers which can be attached to your own project's text fields. For installation instructions, check out *Basic Setup*

3.3.1 FilePicker

class `file_picker.uploads.file_pickers.`**`FilePicker`**

This is a subclass of `file_picker.FilePickerBase` which is connected to the *File* model and can be found in the Uploads admin section.

3.3.2 ImagePicker

class `file_picker.uploads.file_pickers.`**`ImagePicker`**

This is a subclass of `file_picker.ImagePickerBase` which is connected to the *Image* model and can be found in the Uploads admin section.

3.3.3 Simple File Picker Widget

These pickers can be used like any other. Below is an example of how to add them on a single text field:

```
body = forms.TextField(
            widget=file_picker.widgets.SimpleFilePickerWidget(pickers={
                'file': "files",
                'image': "images",
            }))
```

The *"file"* and *"image"* keywords add classes to the inputs, so that the links for the overlay can be added. If you're using the Django admin, they can also be added to all fields in a form by using the *formfield_overrides* on the admin class, as in:ref:*setup*.

3.4 The WYMeditor App

The WYMeditor app is included to make it easy to integrate a File Picker with a popular WYSIWYG interface. WYMeditor is a Javascript based editor. Its documentation can be found here. This application offers an extra form widget for applying WYMeditor to a text field with buttons for files and images if desired.

3.4.1 WYMeditorWidget

class `file_picker.wymeditor.widgets.`**`WYMeditorWidget`**

To use the WYMeditorWidget, override the desired form field's widget. It takes in a dictionary with expected keys *"image"* and/or *"file"* which define which button is used to call the overlay, either an image or a paper clip icon respectively.

Example TextField Override

An example using a *formfield_override* in an admin class using WYMeditor and a File Picker for each *TextField* in the form:

```
class PostAdmin(admin.ModelAdmin):
        formfield_overrides = {
            models.TextField: {
                'widget': file_picker.wymeditor.widgets.WYMeditorWidget(
                    pickers={
                        'image': "images",
                        'file': "files",
                    }
                ),
            },
        }

        class Media:
            js = ("http://cdn.jquerytools.org/1.2.5/full/jquery.tools.min.js",)
```

3.5 Creating Custom Pickers

File Picker offers many ways to create custom pickers and assign them to your own models.

3.5.1 The FilePickerBase Class

The base file picker class has a mixture of class based views and helper functions for building the colorbox on the page. File pickers should be included in the `file_pickers.py` file in the root directory of any app so that the auto-discovery process can find it.

Attributes

Each picker can take a set of attributes for easy customization.:

```
from myapp.models import CustomModel
from myapp.forms import CustomForm
import file_pickers

class CustomPicker(file_picker.FilePickerBase):
    form = CustomForm
    page_size = 4
    link_headers = ['Insert File',]
    columns = ['name', 'body', 'description',]
    extra_headers = ['Name', 'Body', 'Description',]
    ordering = '-date'

file_picker.site.register(CustomModel, CustomPicker, name='custom')
```

None of these attributes are required and they all have sane defaults.

- *form*- If not set, a *ModelForm* is created using the model defined in the register function. This is used to build the form on the Upload tab.
- *link_headers*- Defines the headers for the first set of columns which are used to insert content into the textbox or WYSIWYG widget of your choice. By default, it converts _ to ‘ ‘ and capitalizes the first letter of the field’s name.
- *columns*- Defines the fields you want to be included on the listing page and their ordering.
- *extra_headers*- This list is used to display the headers for the columns and needs to be in the same order as *columns*.
- *ordering*- Defines the order of items on the listing page. In this example, the code would run `query_set.order_by('-date')`. If no ordering is provided, we’ll order by `-pk`.

Methods

The three main methods consist of *append*, *list*, and *upload_file*. *List* and *upload_file* take a request object and act as views, while *append* takes a model instance and builds the JSON output for list. Other methods are available but typically do not need to be modified.

append(obj)

This method takes *obj* which is a model instance and returns a dictionary to be used by *list*. This dictionary is of the form:

```
{
    'name': 'Name for the object.',
    'url': 'The url to the file.',
    'extra': {
```

```
        'column_name_1': 'value',
        'column_name_2': 'value',
    },
    'insert': [
        'text or html to insert if first link is clicked',
        'text or html to insert if second link is clicked',
    ],
    'link_content': [
        'string to show on first insert link',
        'string to show on second insert link',
    ],
}
```

The *link_content* list, *insert* list, and *extra* dictionary must all be the same length, and must match the length of the *link_headers* of your custom FilePicker.

list(request)

This takes a *request* object and returns:

```
{
    'page': 1-based integer representing current page number,
    'pages': List of pages,
    'search': The current search term,
    'result': List returned from *append*,
    'has_next': Boolean telling paginator whether to show the next button,
    'has_previous': Boolean telling paginator whether to show the previous button.,
    'link_headers': List of link headers defined by the Picker attribute (or␣
↪generated if not defined),
    'extra_headers': List of the extra_headers specified by the Picker attribute,
    'columns': List of column names specified by the Picker attribute,
}
```

upload_file(request)

This takes a *request* object and builds the upload file form, which is used to upload files in two steps: first the file, and then the other form parameters.

If called without POST data it returns a JSON dictionary with the key `form` containing the HTML representation of the form.

If called with a file and then with the POST data, it performs a two step process. If the form validates successfully on the second step it returns the result of *append* for the object which was just created.

3.6 Screenshots

3.7 Motivation

The deep concern while building a file picker has been flexibility. Too many projects focus on wrapping everything together so that they can make deep connections.

Our main goal has been to build an application that is easy to plug into a wide variety of applications and models.

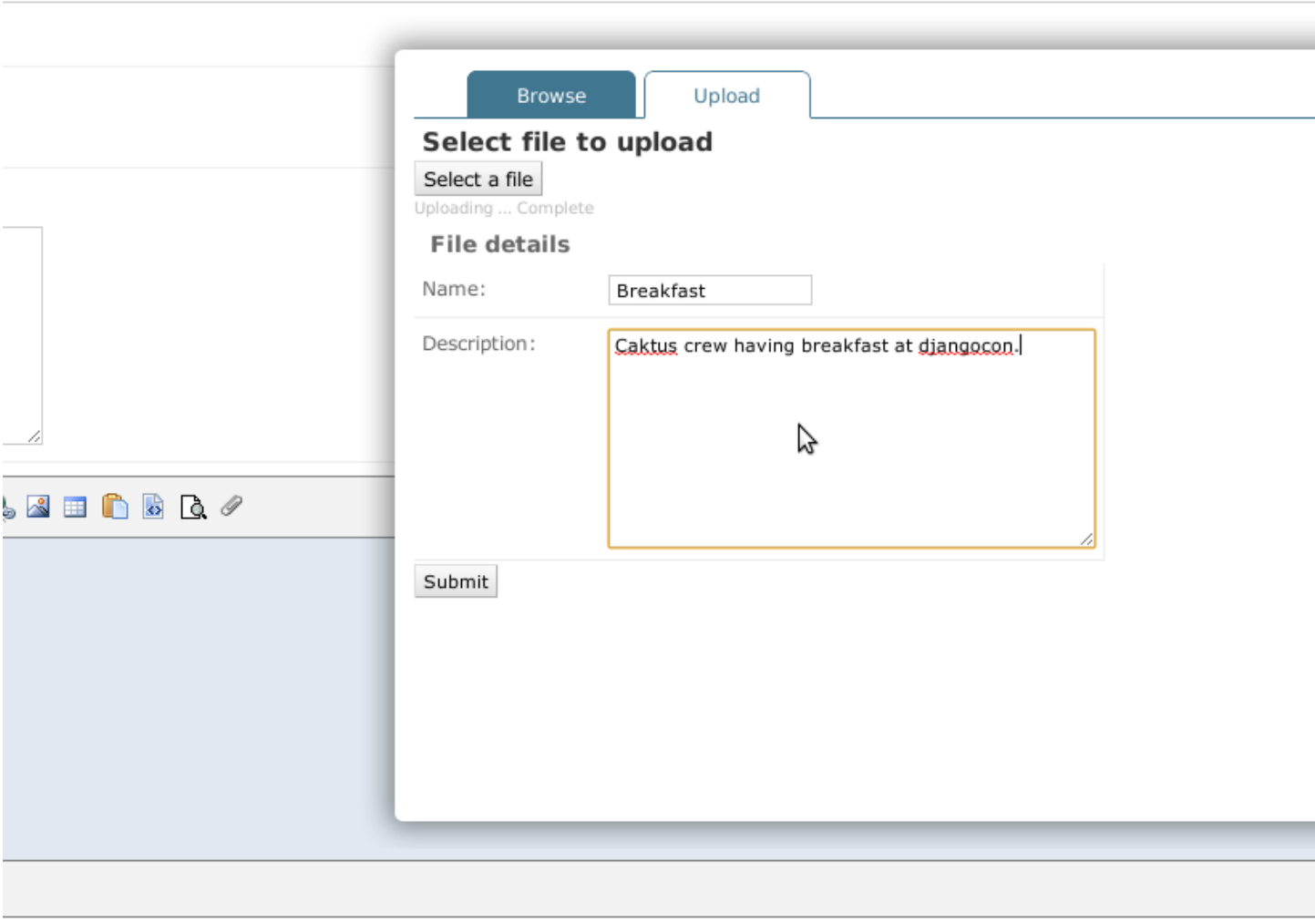

Fig. 3.1: The upload pane from the sample project.

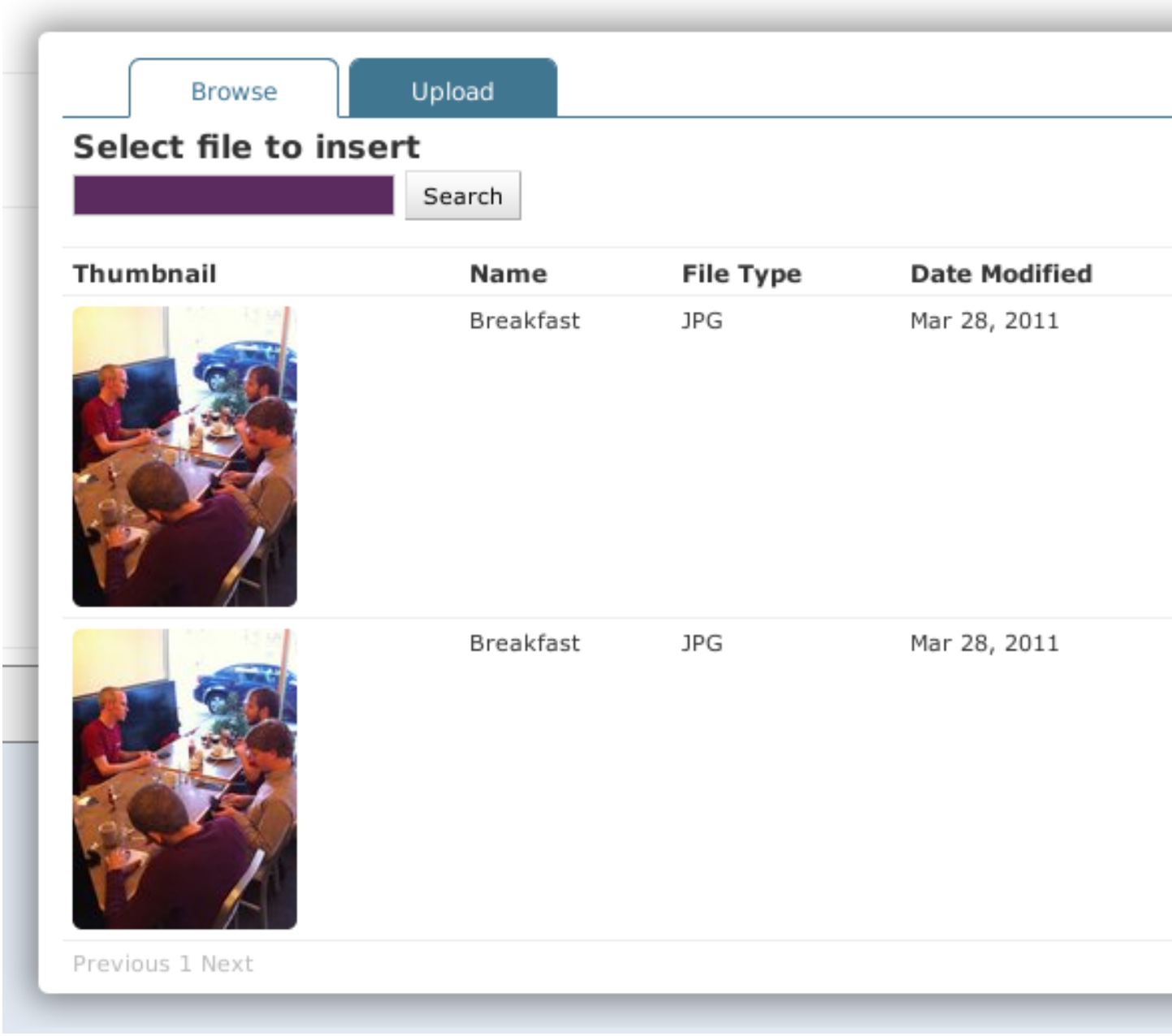

Fig. 3.2: The browser pane from the sample project.

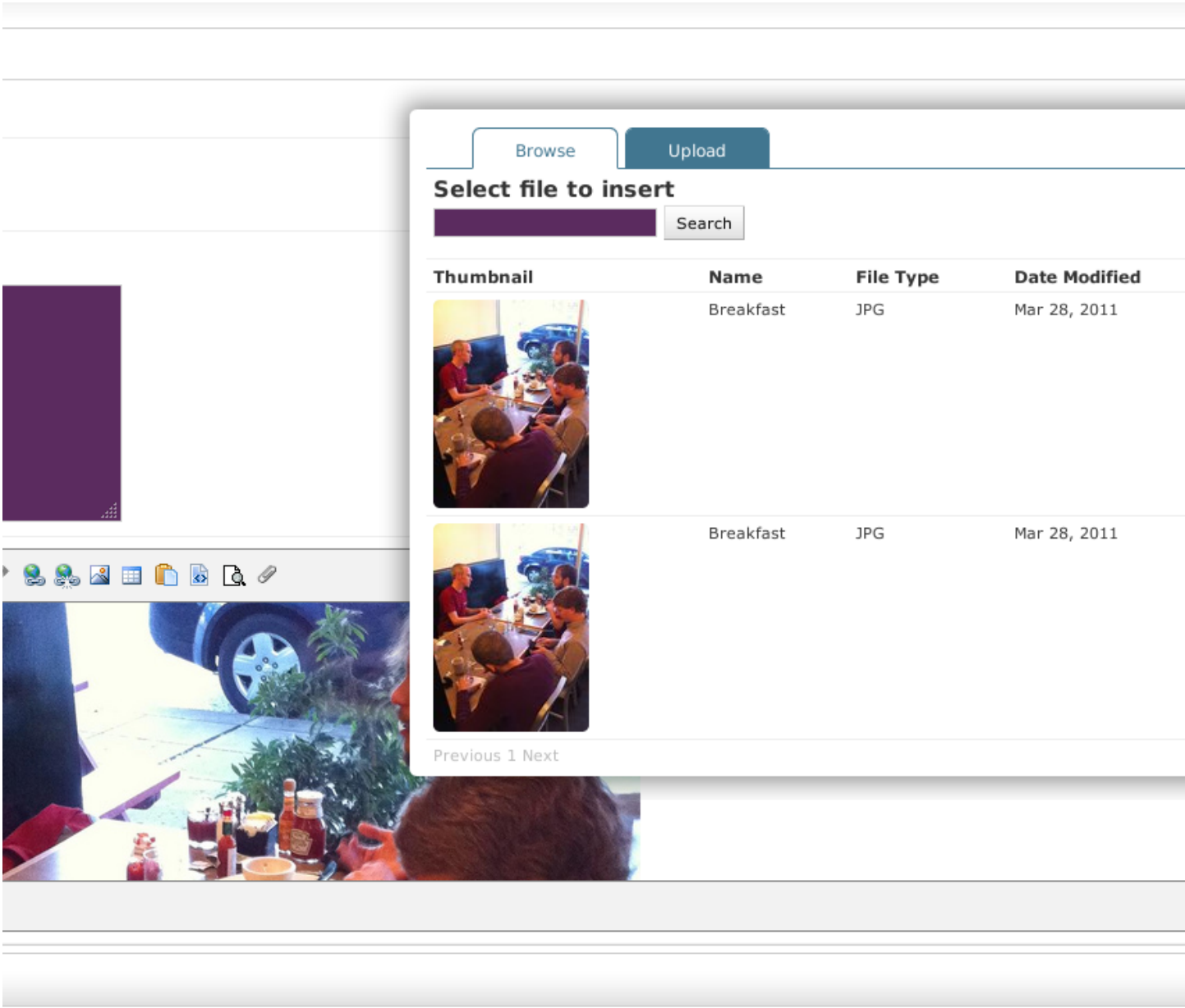

Fig. 3.3: An example of inserting an image with File Picker.

3.8 Running the Sample Project

django-file-picker includes a sample project to use as an example. You can run the sample project like so:

```
~$ mkvirtualenv filepicker-sample
(filepicker-sample)~$ pip install "django>=1.8,<2.0"
(filepicker-sample)~$ git clone git://github.com/caktus/django-file-picker.git
(filepicker-sample)~$ cd django-file-picker/
(filepicker-sample)~/django-file-picker$ python setup.py develop
(filepicker-sample)~/django-file-picker$ cd sample_project/
(filepicker-sample)~/django-file-picker/sample_project$ ./manage.py migrate
(filepicker-sample)~/django-file-picker/sample_project$ ./manage.py createsuperuser
(filepicker-sample)~/django-file-picker/sample_project$ ./manage.py runserver
```

Then go to the admin, log in, and add an Article Post. There will be 2 links to 'Insert File' and 'Insert Image' which will pop up the File Picker dialog.

3.9 Release History

3.9.1 0.9.1 (released 2017-11-06)

- Fixed bug when trying to read binary content.
- Greatly expanded documentation explaining how to get started with django-file-picker.

3.9.2 0.9.0 (released 2017-10-31)

- Added Python 3 support

3.9.3 0.8.0 (released 2017-10-27)

- Added support for Django 1.11

3.9.4 0.7.0 (released 2017-10-26)

- Added support for Django 1.8, 1.9 and 1.10
- Dropped support for Django < 1.8

3.9.5 Previous versions

- See commit history for clues.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

F