Learning Objectives

- **Explain what `django-versatileimagefield` does**
- **Define ppoi**
- **Add a hero image**
- **Create a thumbnail image at a specified size**
- **Add images to a serializer**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

django-versatileimagefield

Intro

Django has its own `ImageField` model field, designed to make it easy to attach an image to your model. Users can upload an image (for example, through the Django Admin) and it's stored as a file in the media directory for the project (as defined by the `MEDIA_ROOT` setting for your project). The `ImageField` itself just stores a path to the file in the database.

django-versatileimagefield is a library that provides a drop-in replacement for `ImageField`: `VersatileImageField` (`versatileimagefield.fields.VersatileImageField`). It provides helper methods to generate thumbnails, crop images, and apply filters.

Along with `VersatileImageField` is `PPOIField` (`versatileimagefield.fields.PPOIField`), or *Primary Point of Interest*. This is a field that stores the coordinates of the "point of interest" of the image. If this is set, then when the `VersatileImageField` crops or thumbnails an image it will center the crop around the primary point of interest instead of the center of the image.

To use an `ImageField` in a template, we'd access its `url` attribute. In Blango we're going to add a `hero_image` field to the `Post` model, so accessing it in a template would be done like this:

```
<img src="{{ post.hero_image.url }}"/>
```

This works the same with a `VersatileImageField`, but we can also perform cropping like this:

```
<img src="{{ post.hero_image.crop.100x100.url }}"/>
```

Where `100x100` is the size of the cropped image. It will be cropped around the PPOI, if set, otherwise the center of the image.

Alternatively, we can thumbnail the image:

```
<img src="{{ post.hero_image.thumbnail.100x100.url }}"/>
```

What’s the difference between cropping and thumbnailing? When the image is cropped, it is cropped to the exact size given. When thumbnailing, it’s resized, not cropped, to fit into the given dimensions. For non-square images, this means that the maximum dimension in our example would be 100 pixels, while the other dimensions would be proportionally smaller.

Try It Out

`VersatileImageField` sounds pretty straightforward, but we need a couple of adjustments to Blango before we can upload and serve images. We’ll then add the `hero_image` and `ppoi` fields to the `Post` model.

We’ll start by getting `django-versatileimagefield` installed, with `pip`:

```
pip3 install django-versatileimagefield
```

▼ **Mac and Windows**

If you’re working outside of the Codio platform, you will need to install some additional modules. On macOS, you’ll need to install libmagic, for example, with Homebrew:

```
brew install libmagic
```

On Windows, you’ll have to install `python-magic-bin`:

```
pip install python-magic-bin==0.4.1
```

Once installed, your first `settings.py` change is to add `versatileimagefield` to your `INSTALLED_APPS`.

Open settings.py

Next, you’ll need to add the settings to enable media saving and serving. The first is `MEDIA_ROOT`, which defines where uploaded files are saved. We’ll put them in a directory called `media`, inside the main `blango` project directory. To set this up, add this setting to the `Dev` class in `settings.py`:

```
    MEDIA_ROOT = BASE_DIR / "media"
```

Then we need to tell Django which URL/path to serve media from. In our case, it will be `/media/`. This is set with the `MEDIA_URL` setting. Add this to the `Dev` class too:

```
    MEDIA_URL = "/media/"
```

Now that you have those settings, we need to configure a URL pattern to serve static files from the `MEDIA_ROOT`. We'll use the `django.conf.urls.static.static` function which returns a list of URL patterns, and serves from the given `document root`. Open `blango/urls.py`, then add this import:

Open urls.py

```
from django.conf.urls.static import static
```

You should already have a check for `settings.DEBUG` in this file, and you'll only be adding the `debug_toolbar.urls` patterns if in `DEBUG` mode. Add the media `static` pattern inside this check too, so the whole thing looks like this:

```
if settings.DEBUG:
    urlpatterns += [
        path("__debug__/", include(debug_toolbar.urls)),
    ] + static(settings.MEDIA_URL,
        document_root=settings.MEDIA_ROOT)
```

You'll need to create the `media` directory too, directly inside the `blango` project directory (the outer directory, not the inner `blango` directory that contains `urls.py` and `settings.py`, etc). You can check that media serving is set up correctly by also creating a file such as a `test.txt` file inside the `media` directory and checking if you can download it (the path will be, for example, `/media/test.txt`). The text file is not needed. Feel free to delete it if you want. Keeping it will not affect the project.

Assuming you have media serving working, let's start using the `VersatileImageField`. As mentioned, we'll be adding `hero_image` and `ppoi` fields to the `Post` model. Open up `blog/models.py`.

Open models.py

You'll first need to import the `versatileimagefield` fields first, so add this import:

```
from versatileimagefield.fields import VersatileImageField,
        PPOIField
```

Then add the `hero_image` (a `VersatileImageField`) and `ppoi` (a `PPOIField`) to the `Post` class.

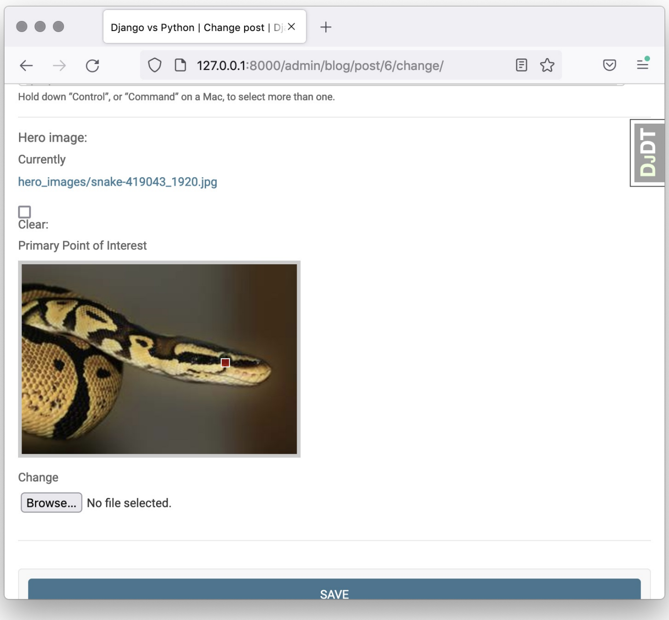
```
class Post(models.Model):
    # existing fields omitted
    hero_image = VersatileImageField(
        upload_to="hero_images", ppoi_field="ppoi", null=True,
        blank=True
    )
    ppoi = PPOIField(null=True, blank=True)
```

We're using the `upload_to` argument so that hero images will be automatically uploaded to the directory `hero_images` inside the `MEDIA_ROOT`. The `ppoi_field` argument sets the name of a `PPOIField` on the model. This is so the `hero_image` knows how to find the PPOI when performing a crop.

Run the `makemigrations` and `migrate` commands with `manage.py` to apply these changes to the database.

Now log in to the Django Admin site and edit a `Post`. You should see the *Hero image* field. Try uploading an image to it. You'll see that once an image is set, you'll be able to set the PPOI by dragging the red square around the image. See the next screenshot with the picture of a python and PPOI on its eye.

View Blog


Django vs Python | Change post | Dj
127.0.0.1:8000/admin/blog/post/6/change/
DjDT
Browse...
SAVE

Hold down "Control", or "Command" on a Mac, to select more than one.

Hero image:
Currently
hero_images/snake-419043_1920.jpg

Clear:
Primary Point of Interest

Change
No file selected.

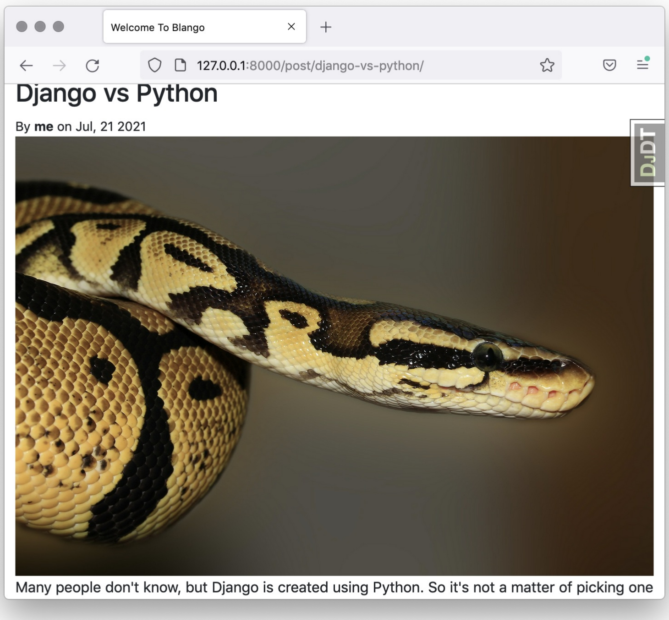
Figure 1

Save the `Post` (you can upload images to multiple `Posts` if you like), then we'll make some changes to see our image, first in the post detail view. Open `blog/templates/blog/post-detail.html`. We want to include the hero_image if it is set on the Post. Add a new row under the row that contains the byline include:

Open post-detail.html

```
{% if post.hero_image %}
    {% row %}
        {% col %}
            <img class="img-fluid" src="{{ post.hero_image.url
        }}">
        {% endcol %}
    {% endrow %}
{% endif %}
```

Here we're not using any kind of VersatileImageField special tricks, just including the URL. Open up a detail page for Post that you've added an image to and check that the image shows up.



Django vs Python

By **me** on Jul, 21 2021

Many people don't know, but Django is created using Python. So it's not a matter of picking one

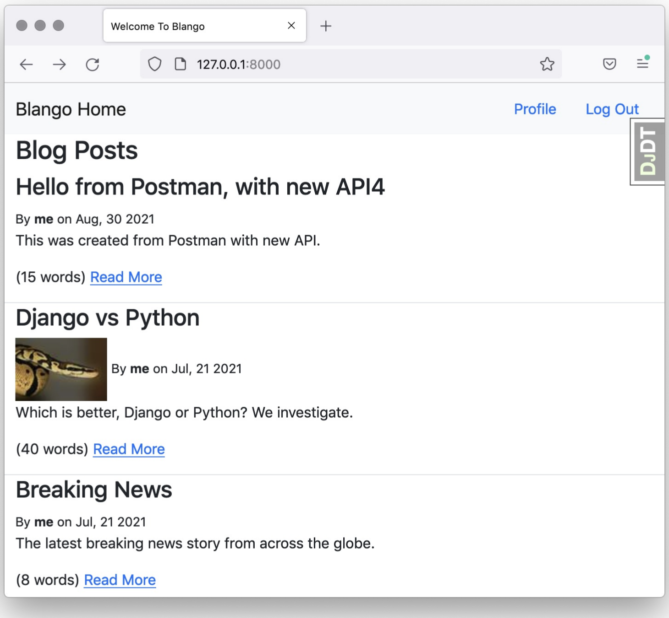
image in post detail

Next we'll use the thumbnail feature. Open blog/templates/blog/index.html. Underneath the <h3> element that contains the title, add this to show a thumbnail of the hero_image.

Open index.html

```
{% if post.hero_image %}
    <img src="{{ post.hero_image.thumbnail.100x100.url }}"/>
{% endif %}
```

We're making the thumbnail have a maximum dimension of 100 pixels (this number is chosen somewhat arbitrarily). Open up the post list page in your browser and you should see a thumbnailed image next to any post that has a `hero_image` set.



Blango Home Profile Log Out

Blog Posts

Hello from Postman, with new API4

By **me** on Aug, 30 2021
This was created from Postman with new API.

(15 words) Read More

Django vs Python

By **me** on Jul, 21 2021
Which is better, Django or Python? We investigate.

(40 words) Read More

Breaking News

By **me** on Jul, 21 2021
The latest breaking news story from across the globe.

(8 words) Read More

thumbnail in post list

If you're curious about how `VersatileImageField` "caches" images, you can look inside the `media` directory. Inside is a `__sized__` directory which is a replica of the structure of your `media` directory – inside is another directory called `hero_images` which contains any resized hero images that have been generated. If you have more `VersatileImageFields` in use, then more directories will be created to contain the resized images.

Now that we've integrated a `VersatileImageField`, using it with Django Rest Framework is actually quite straightforward. Let's look at that now.

Use with Django Rest Framework

Use with Django Rest Framework

`VersatileImageField`, by default, works just like a normal `ImageField` when included on a serializer. It will serialize to the URL of the image. For example, on the `PostSerializer`, since we're including all the fields (by using `fields = "__all__"`) we'll have it shown automatically:

```
{
    "count": 11,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 6,
            "author":
        "http://127.0.0.1:8000/api/v1/users/ben@example.com",
            "hero_image":
        "http://127.0.0.1:8000/media/hero_images/snake-
        419043_1920.jpg",
            "ppoi": "(0.74, 0.52)",
            ...
        },
        ...
    ]
}
```

Notice that `ppoi` is also included, as you might expect.

To add different sizes that can be fetched, a `VersatileImageFieldSerializer` (`versatileimagefield.serializers.VersatileImageFieldSerializer`) field needs to be added to the serializer. To control what image sizes are available, this can be instantiated with the `sizes` parameter. This is a list of two-element tuples, each containing a key/identifier of the image size, and a string which `VersatileImageField` can parse to determine how to generate the image.

Here's an example `sizes` list:

```
[
    ("full_size", "url"),
    ("thumbnail", "thumbnail__100x100"),
    ("square_crop", "crop__200x200"),
]
```

This gives the client three options for fetching a resized image:

- `full_url`: The URL for the full-sized image.
- `thumbnail`: The URL for a thumbnailed image of max dimension 100 pixels.
- `square_crop`: The URL for a cropped image, 200 x 200 pixels around the PPOI.

Of course, we could define any number of image sizes that we want.

Here it is in use on the `PostDetailSerializer`:

```python
class PostDetailSerializer(PostSerializer):
    hero_image = VersatileImageFieldSerializer(
        sizes=[
            ("full_size", "url"),
            ("thumbnail", "thumbnail__100x100"),
            ("square_crop", "crop__200x200"),
        ],
        read_only=True,
    )
    # other attributes and methods omitted
```

In the API response, these are contained in a dictionary. For example, in the `Post` detail:

```
{
    "id": 6,
    "hero_image": {
        "square_crop":
        "http://127.0.0.1:8000/media/__sized__/hero_images/snake-
        419043_1920-crop-c0-74__0-52-200x200-70.jpg",

        "full_size":
        "http://127.0.0.1:8000/media/hero_images/snake-
        419043_1920.jpg",
        "thumbnail":
        "http://127.0.0.1:8000/media/__sized__/hero_images/snake-
        419043_1920-thumbnail-100x100-70.jpg"

    },
    ...
}
```

In order for these images to be available, `VersatileImageField` pre-generates them when the serializer is used, that is, as soon as the `Post` object is fetched, either as part of the list or as the detail. For this reason, you should avoid having too many different size options as part of the serialized list because you'll be generating a lot of images which might not be used (although this will only happen once). Instead, use different `VersatileImageField` configurations on the list and detail serializers. For example, our blog API clients might want to show a thumbnail in the list of `Posts` that they display, so we'll just include the thumbnail URL in the list response. We'll include more options in the detail response, so they can show an image that's an appropriate size when showing the post detail.

Try It Out

Now you'll add some options for fetching different-sized hero images to your Blango API. We already have separate serializers for the `Post` list view and the `Post` detail view, so it's easy for us to have different `hero_image` sizes for each view.

In the `blog/api/serializers.py` file, import `VersatileImageFieldSerializer`:

```
from versatileimagefield.serializers import
        VersatileImageFieldSerializer
```

The first change is to `PostSerializer`, instead of including all the fields we want to exclude `ppoi`, as it doesn't mean anything to most clients. Change the line:

```
        fields = "__all__"
```

to:

```
        exclude = ["ppoi"]
```

Next we'll add an option to fetch the thumbnail image, so add this serializer field to `PostSerializer`:

```
    hero_image = VersatileImageFieldSerializer(
        sizes=[
            ("full_size", "url"),
            ("thumbnail", "thumbnail__100x100"),
        ],
        read_only=True,
    )
```

We include the `full_size` option since it's already generated and basically "free" to include.

On `PostDetailSerializer`, we'll add one extra size for a cropped 200x200 image. Add this field:

```
    hero_image = VersatileImageFieldSerializer(
        sizes=[
            ("full_size", "url"),
            ("thumbnail", "thumbnail__100x100"),
            ("square_crop", "crop__200x200"),
        ],
        read_only=True,
    )
```

Remember `PostDetailSerializer` inherits from `PostSerializer` so the `ppoi` field is already excluded from it.

Now try viewing your `Post` list or detail endpoints. For `Posts` without a `hero_image` set, you'll get an empty dictionary. Otherwise, you'll get a dictionary with `full_size` and `thumbnail` keys in both list and detail views, with the addition of a `square_crop` key in the detail endpoint.

View Blog

Try also fetching one of the image URLs that are generated, to verify they're working properly.

Wrap Up

django-versatileimagefield is a very useful library if you’re working with images, whether using DRF or not. If you’d like to read more about the filter it offers and other options, you can check out the official documentation.

That bring us to the end of this module on third-party libraries for DRF. In the next module we’re going to take a step away from Django and Python and look at how to write some JavaScript code and use the React framework to build an interactive UI, that fetches data from our API.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish django-versatileimagefield"
```

- Push to GitHub:

```
git push
```