DRF Magic

Release 0.1.0

Dan Starner

Oct 28, 2020

CONTENTS

Django REST Framework Magic (DRF Magic) is a Django application that builds off our all of my personal best practices of boiler-plates of working with a RESTful Django application. It makes a fair amount of subjective coding decisions, but I hope that others see the benefit and ease-of-development that they provide.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Authorization & Permissions

1.2 Model & Data Serialization

Serializers are a Django Rest Framework concept that describe converting models to and from JSON, and under what circumstances. They provide a middle man that when paired with views and viewsets can automatically parse and unparse data between requests and responses. They provide a common interface for describing how (de)serialization works through the project and its routes.

DRF Magic provides tooling to make working with serialization even easier by including class decorators to automatically tie serializer classes to the models that they represent. Note that the more you buy into the different features of DRF Magic, the more powerful the automatic serialization features become.

1.2.1 Model Core Serializers

For most models, there is usually one core serializer that is used for a majority of the routes, usually at least for **listing** of a group of instances and **retrieving** a specific instance. This means that it may be useful to create a mapping from a model to its main serializer class, so that instances and data of that model/request type can easily and quickly be converted back and forth while having the serializer model mapping declared only once, providing good *D.R.Y* programming habits.

Registering a Main Serializer

DRF Magic has a `register_main_serializer` function that allows us to easily create this serializer-to-model mapping when we define our serializers based on their defined `Meta.model` attributes.

`drf_magic.serializers.loader.`**`register_main_serializer`**(*serializer_class*)

Registers a ModelSerializer as the main serializer for a model type when attached as a decorator on the serializer class declaration. It uses the Meta.model attribute, and each model can only have a single main serializer associated with it.

We can use this in a project in the following manner. Let's say we have a `Person` model defined, such as what is below.

```
from django.db import models


class Person(models.Model):
```

(continues on next page)

(continued from previous page)

```
    first_name = models.CharField(max_length=64, default='')

    last_name = models.CharField(max_length=64, default='')

    occupation = models.CharField(max_length=128, default='programmer')
```

We can then define a `PersonalSerializer` that is registered as the main serializer for the above-defined `Person` model.

```
# my_app/serializers.py
from drf_magic.serializers.loader import register_main_serializer
from rest_framework import serializers

from .models import Person

@register_main_serializer  # <-- registers the main serializer
class PersonSerializer(serializers.ModelSerializer):
    """Main serializer class for converting Person instances to JSON and back
    """

    class Meta:
        model = Person
        fields = ['id', 'first_name', 'last_name', 'occupation']
```

Notice how we include the `@register_main_serializer` around the serializer class definition. That's the key feature that registers this serializer to the `Person` class.

We can then pull out this serializer in a declarative, but dynamic way using the `load_model_serializer` function.

`drf_magic.serializers.loader.`**`load_model_serializer`**(*model_class*)

Used to safely load a serializer from the main serializer-to-model map

```
# my_app/views.py
from drf_magic.serializers.loader import load_model_serializer

# Somewhere in our code...
person_serializer_class = load_model_serializer(Person)
assert person_serializer_class == PersonSerializer
```

Automatically Load Main Serializers

Due to the nature of Django apps and how they are loaded, we must load them when the app configs are `ready()`. This can be done automatically with the `AutoSerializerAppConfig` class.

Note: This does not have to be included specifically if your app configs already subclass from `drf_magic.MagicAppConfig`, as this subclass is already included in that case.

---

class `drf_magic.serializers.loader.`**`AutoSerializerAppConfig`**(*app_name*, *app_module*)

Will attempt to load and register the app's serializers automatically on app load

In a real example, it may look like the following, continuing with our ongoing example.

```python
# my_app/apps.py

# Preferred method if also using auto model accessors
from drf_magic import MagicAppConfig


class MyAppConfig(MagicAppConfig):
    pass

# --------------------------------

# If needed only for serializers
from drf_magic.serializers.loader import AutoSerializerAppConfig


class MyAppConfig(AutoSerializerAppConfig):
    pass
```

1.2.2 Serializers in Viewsets

Once we have main model serializers set up for our major models, we can have *viewsets* automatically use them when we define the viewset's `model` attribute. By default, DRF Magic-based viewsets will load the main serializer for its associated views. This functionality comes from the `AutoSerializerMixin` viewset mixin and its overriding of `get_serializer_class`.

With all of the following performed, we can then have model instances and data be automatically set and converted based on the mapping of the `Person` model to the `PersonSerializer` class.

```python
# my_app/views.py
from rest_framework.responses import Response

from .models import Person

# TODO: Fix this import once its finalized
class PersonViewset(AutoSerializerMixin or MagicViewset):
    """API routes to handle Person model interaction
    """
    model = Person

    # get_serializer_class() --> PersonSerializer

    # builtin actions (list, retrieve, create, delete...) will
    # expect a PersonSerializer-compatible JSON structure for
    # incoming data requests and will convert model instances
    # to the same structure on responses automatically

    @action(detail=True)
    def custom_retrieve(self, **kwargs):
        person = self.get_object()

        # This is where our main serializer setup comes into play
        serializer_class = self.get_serializer_class()

        serializer = serializer_class(instance=person)
        return Response(serializer.data)
```

Overridding the Main Serializer

In some cases though, it may be deemed necessary to override what serializer to use for a specific viewset action. This may be useful if you have custom actions on the viewset that require different JSON data structures, or just want to have different serializers for a specific action type.

Tip: A scenario that I find myself overriding the core serializer is when implementing a generic main serializer for **list**-ing, and a detailed serializer when **retrieve**-ing, or vis versa.

These overrides can be implemented by placing a `<action>_serializer_class` attribute on the associated viewset where `<action>` is the action type that the serializer class should be overridden for.

```
from .serializers import PersonCreationSerializer


class PersonViewset(AutoSerializerMixin or MagicViewset):

    # used during 'create' actions
    create_class_serializer = PersonCreationSerializer

    # used during 'fire' actions
    fire_serializer = FirePersonSerializer

    @action(detail=True)
    def fire(self, **kwargs):
        serializer_class = self.get_serializer_class()
        # ... do something and return a response
```

1.3 Route & Viewset Building

1.4 Contributing & Interaction Guidelines

Welcome! Thank you for your interest in participating in the `drf-magic` community! We are excited to see new individuals wanting to get involved in the project, and we want to support you as best we can

Before contributing, we kindly ask that you read out *Code of Conduct* below to get a sense of the kind, caring, and inclusive community we are trying to foster around these projects. We then kindly ask that you also read our *Contributing Overview* to learn how to give back to the project.

From the Glyph team, thank you so much for your time and support!

1.4.1 Code of Conduct

Like the technical community as a whole, the Glyph team and community is made up of a mixture of professionals and volunteers from all over the world, working on every aspect of the mission - including mentorship, teaching, and connecting people.

Diversity is one of our huge strengths, but it can also lead to communication issues and unhappiness. To that end, we have a few ground rules that we ask people to adhere to. This code applies equally to founders, mentors and those seeking help and guidance.

This isn't an exhaustive list of things that you can't do. Rather, take it in the spirit in which it's intended - a guide to make it easier to enrich all of us and the technical communities in which we participate.

This code of conduct applies to all spaces managed by the Glyph project. This includes IRC, the mailing lists, the issue tracker, and any other forums created by the project team which the community uses for communication. In addition, violations of this code outside these spaces may affect a person's ability to participate within them.

If you believe someone is violating the code of conduct, we ask that you report it by emailing bbpuzzle-hunt@gmail.com.

- **Be friendly and patient.**
- **Be welcoming.** We strive to be a community that welcomes and supports people of all backgrounds and identities. This includes, but is not limited to members of any race, ethnicity, culture, national origin, colour, immigration status, social and economic class, educational level, sex, sexual orientation, gender identity and expression, age, size, family status, political belief, religion, and mental and physical ability.
- **Be considerate.** Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions. Remember that we're a world-wide community, so you might not be communicating in someone else's primary language.
- **Be respectful.** Not all of us will agree all the time, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one. Members of the Glyph community should be respectful when dealing with other members as well as with people outside the Glyph community.
- **Be careful in the words that you choose.** We are a community of professionals, and we conduct ourselves professionally. Be kind to others. Do not insult or put down other participants. Harassment and other exclusionary behavior aren't acceptable. This includes, but is not limited to:
  - Violent threats or language directed against another person.
  - Discriminatory jokes and language.
  - Posting sexually explicit or violent material.
  - Posting (or threatening to post) other people's personally identifying information ("doxing").
  - Personal insults, especially those using racist or sexist terms.
  - Unwelcome sexual attention.
  - Advocating for, or encouraging, any of the above behavior.
  - Repeated harassment of others. In general, if someone asks you to stop, then stop.
- **When we disagree, try to understand why.** Disagreements, both social and technical, happen all the time and Glyph is no exception. It is important that we resolve disagreements and differing views constructively. Remember that we're different. The strength of Glyph comes from its varied community, people from a wide range of backgrounds. Different people have different perspectives on issues. Being unable to understand why someone holds a viewpoint doesn't mean that they're wrong. Don't forget that it is human to err and blaming each other doesn't get us anywhere. Instead, focus on helping to resolve issues and learning from mistakes.

Original text courtesy of the Speak Up! project. This version was adopted from the Django Code of Conduct.

1.4.2 Contributing Overview

Woah, we are super stoked that you want to work on the project and contribute feedback, code, or ideas! We love and always appreciate positive involvement of any kind. Before you begin though, please read the following sections to increase efficiency, reduce the waiting time for responses to your comments & coded, and to help us better manage the project.

Thank you for your contributions and cooperation.

Issues

You can use github issues to request features and file bug reports. An issue is also a good place to ask questions. We are happy to help out if you have reached a dead end, but please try to solve the problem yourself first.

When creating an issue there are couple of things you need to remember:

1. **Update to the latest version if possible and see if the problem remains.**

   If updating is not an option you can still request critical bug fixes for older versions.

2. **Describe your problem.**

   Please fill out an issue using one of the provided templates. Answer the associated questions to the best of your abilities & knowledge, as it will allow us to better help you in a more timely manner. People often leave out details or use made up examples because they think they are only leaving out irrelevant stuff. If you do that, you have already made an assumption about what the problem is and it's usually something else. Also provide all possible stack traces and error messages.

Please bear in mind that we aim to have high test coverage against both the frontend and backend of the application to reduce issues before they occur. If you run into an issue, this doesn't mean the larger picture issue is broken, but potential one small use-case or edge-case that you found may be. That's why enough context is necessary. It's not enough to say, "Feature *X* fails". You need to provide the code and background that fails and usually the models that are used too. And let's say this again: **don't provide made up examples!** When you do, you only write the parts you think are relevant and usually leave out the useful information. Use the actual code that you have tested to fail.

Pull requests

If you have found a bug or want to add a feature, pull requests are always welcome! It's better to create an issue first to open a discussion if the feature is something that should be added to Glyph. In case of bugfixes it's also a good idea to open an issue indicating that you are working on a fix to avoid duplicated work.

For a pull request to get merged it needs to have the following things:

1. **A good description of what the PR fixes or adds. You can just add a link to the corresponding issue.**

   Please use one of our existing Pull Request templates and fill in the information to the best of your ability so that we may streamline the review and discussion process.

2. **Tests that verify the fix/feature.**

   It's possible to create a PR without tests and ask for someone else to write them but in that case it may take a long time or forever until someone finds time to do it. *Untested code will never get merged!*

3. **For features you also need to write documentation.**

   As the project matures and becomes more popular, good documentation is key for allowing those in the Glyph community to work with the project in an independent, autonomous manner.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX