New Generation of APIs

José Antonio Perdiguero López

http://www.perdy.io

https://github.com/PeRDy

https://www.linkedin.com/in/perdy

perdy@perdy.io

October 6, 2018

Head of Data Science @ Whalar

Index

Introduction

Introduction

What's wrong with current API frameworks? Nothing at all, except they seems a bit **old** and **unexpressive**.

Let's improve it using new Python functionalities like:

- Type annotation.
- Module `typing`.
- New `async`/`await` semantic.
- Module `asyncio`.

Introduction

The codebase can be very **expressive** in terms of describing the API. Make it the first information source of the API.

Goals

Speep up API building and maintenance.

Create an API with a codebase **expressive**.

An **interactive documentation** kept **in sync** with the API.

A way to infer and generate API **schema**.

Possibility to work with **ASGI** and **websockets**.

Example API

Puppy API 🐾

- **Register** a new puppy.
- **List** all puppies, filtered by name.

Example API

```
def puppy(request):
    if request.method == "POST":
        data = JSONParser().parse(request)
        serializer = PuppySerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data, status=201)
        return JsonResponse(serializer.errors, status=400)

    if request.method == "GET":
        puppies = Puppy.objects.all()
        name = request.query_params.get('name', None)
        if name is not None:
            puppies = puppies.filter(name=name)
        serializer = PuppySerializer(puppies, many=True)
        return JsonResponse(serializer.data, safe=False)
```

Routes

A single route for a single view

```
urls = [
    ("/puppy/", puppy),
]
```

Splitting the view

```
def register_puppy(request):
    """
    Register a new puppy !
    """
    data = JSONParser().parse(request)
    serializer = PuppySerializer(data=data)
    if serializer.is_valid():
        serializer.save()
        return JsonResponse(serializer.data, status=201)
    return JsonResponse(serializer.errors, status=400)

def list_puppy(request):
    """
    List all puppies !
    """
    puppies = Puppy.objects.all()
    name = request.query_params.get('name', None)
    if name is not None:
        puppies = puppies.filter(name=name)
    serializer = PuppySerializer(puppies, many=True)
    return JsonResponse(serializer.data, safe=False)
```

Multiple routes for multiple views

```
urls = [
    ("/puppy/", "POST", register_puppy),
    ("/puppy/", "GET", list_puppy),
]
```

Views

Async view

```python
async def register_puppy(request):
    """
    Register a new puppy !
    """
    data = JSONParser().parse(request)
    serializer = PuppySerializer(data=data)
    if serializer.is_valid():
        serializer.save()
        # Do your async stuff...
        return JsonResponse(serializer.data, status=201)
    return JsonResponse(serializer.errors, status=400)

def list_puppy(request):
    """
    List all puppies !
    """
    puppies = Puppy.objects.all()
    name = request.query_params.get('name', None)
    if name is not None:
        puppies = puppies.filter(name=name)
    serializer = PuppySerializer(puppies, many=True)
    return JsonResponse(serializer.data, safe=False)
```

Components

Validation as part of the view

```
def list_puppy(request: Request) -> Response:
    """
    List all puppies !
    """
    puppies = Puppy.objects.all()
    name = request.query_params.get('name', None)
    if name is not None:
        if name[0].islower():
            raise ValidationError("Puppy name must start with uppercase")
        else:
            puppies = puppies.filter(name=name)
    serializer = PuppySerializer(puppies, many=True)
    return JsonResponse(serializer.data, safe=False)
```

Validation as part of component instance

```
class PuppyName:
    def __init__(self, name: QueryParam):
        if name[0].islower():
            raise ValidationError("Puppy name must start with uppercase")

        self.value = name
```

View with injected components

```
def list_puppy(name: PuppyName) -> Response:
    """
    List all puppies !
    """
    puppies = Puppy.objects.all()
    name = request.query_params.get('name', None)
    if name is not None:
        puppies = puppies.filter(name=name.value)
    serializer = PuppySerializer(puppies, many=True)
    return JsonResponse(serializer.data, safe=False)
```

Types

Types to define Schemas

```
class PuppyType(Type):
    name = validators.String(
        title="name",
        description="Word to pay attention"
    )
    age = validators.Integer(
        title="age",
        description="I'm a puppy yet?"
    )
```

Views using Types

```
def register_puppy(puppy: PuppyType) -> PuppyType:
    """
    Register a new puppy !
    """
    Puppy.objects.create(puppy)
    return JsonResponse(puppy, status=201)
```

Benefits

Docs in sync with code

GET /puppy/

List all puppies !

Query params: **name**

Response body: **List[PuppyType]**

POST /puppy/

Register a new puppy !

Request body: **PuppyType**

Response body: **PuppyType**

Mock the API

Views defined plain input parameters and output schema so that can be completely mocked.

```
def list_puppy(name: PuppyName) -> typing.List[PuppyType]:
    """
    List all puppies !
    """
    pass
```

Generate API Schema

All these changes made that API to expose all the information needed to automatically generate the schema.

Types has a direct relation to **JSON Schema**, so each one can generate his own schema.

The whole **API can be inspected** to build the schema based on standards like **OpenAPI** (former Swagger).

Generic clients

Based on standard schemas such as **OpenAPI** it's quite easy to create generic clients for our API.

Open source your code !

https://discuss.apistar.org/

https://github.com/encode/apistar/tree/version-0.5.x

https://github.com/perdy/apistar-crud

https://github.com/encode/starlette

https://github.com/perdy/starlette-api