Flask Hintful

Release 0.0.4

May 26, 2020

Contents

Flask-Hintful helps you write Restful APIs using Flask by taking advantage of Python's type hints.

It deserializes HTTP query/path parameters according to type hints and automatically generates OpenApi documentation for your registered routes.

For serializing/deserializing complex types it supports using Marshmallow and Dataclasses.

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Installation

Install using pip

```
pip install flask-hintful
```

1.1.2 Creating the FlaskHintful object

Import Flask and FlaskHintful, then pass an instance of your Flask application to the constructor of FlaskHintful.

```
from flask import Flask
from flask_hintful import FlaskHintful

app = Flask('My API')
api = FlaskHintful(app)
```

1.1.3 Creating your API routes

Use type hints to describe your parameters/return types and Flask Hintful will take care of serializing/deserializing them, as well as generating the OpenApi documentation automatically!

```
@api.route('/')
def get_dataclasses(foo: int, bar: str = None) -> List[DataclassModel]:
    '''Returns list of DataclassModel using query args'''
    pass

@api.route('/<id>')
def get_dataclass(id: str) -> DataclassModel:
```

(continues on next page)

(continued from previous page)

```
    '''Returns DataclassModel using path arg'''
    pass

@api.route('/', methods=['POST'])
def create_dataclass(model: DataclassModel) -> DataclassModel:
    '''Creates a DataclassModel using POST and request body'''
    return model
```

For Marshmallow Schema see *Using Marshmallow Schemas*

1.1.4 Registering routes and Blueprints

Use the FlaskHintful object to register routes using the `@route` decorator

```
@api.route('/api/test')
def view_func():
    pass
```

Or register a Flask Blueprint that contains your view funcs.

```
flask_bp = Blueprint('flask_bp', __name__)

@flask_bp.route('/api/test')
def view_func():
    pass

api.register_blueprint(flask_bp)
```

1.1.5 Using Marshmallow Schemas

When using an object that has a Marshmallow Schema Flask Hintful needs a reference to that schema.

By default Flask Hintful search for schemas in an attribute `__marshmallow__`, if you wish to change that behaviour look at *Subclassing*.

Your Marshmallow Schema must also return an instance of your model when using `load` or `loads`, so you'll have to to use the `post_load` decorator to instantiate your object.

Example Marshmallow Schema:

```
from marshmallow import Schema, fields, post_load

class MarshmallowModel():
    def __init__(self,
                 str_field,
                 int_field
                 ):
        self.str_field = str_field
        self.int_field = int_field

class MarshmallowModelSchema(Schema):
    str_field = fields.Str()
    int_field = fields.Int()
```

(continues on next page)

(continued from previous page)

```
    @post_load
    def make_some_model(self, data, **kwargs):
        return MarshmallowModel(**data)

# Sets MarshmallowModelSchema as a schema for MarshmallowModel
setattr(MarshmallowModel, '__marshmallow__', MarshmallowModelSchema)

# Flask Hintful will pick  MarshmallowModel from type hints and use the Schema's dump/
↪load to serialize/deserialize it
@api.route('/', methods=['POST'])
def create_model(model: MarshmallowModel) -> MarshmallowModel:
    '''Creates a MarshmallowModel'''
    return model
```

1.2 Custom Serializers/Deserializers

You can add or replace serializers/deserializers for any "basic" types. For customizing Marshmallow/Dataclass behaviour you'll need to do *Subclassing*.

Do note that adding a (de)serializer to an existing type will override the *default*.

1.2.1 Adding Serializers

On your FlaskHintful instance use `.serializer.add_serializer`

You will need to provide a callable that can receive an instance of the type and return a `str`.

Example, custom serializer that serializes bools to TRUE and FALSE:

```
def my_bool_serializer(data: bool) -> str:
    return str(data).upper()

api.serializer.add_serializer(bool, my_bool_serializer)
```

1.2.2 Adding Deserializers

On your FlaskHintful instance use `.deserializer.add_deserializer`

You will need to provide a callable that can receive a `str` and return an instance of that type.

Example, custom deserializer that deserializes `'yay'` to `True`:

```
def my_bool_deserializer(data: str) -> bool:
    if data.lower() == 'yay':
        return True
    return False

api.deserializer.add_deserializer(bool, my_bool_deserializer)
```

1.2.3 Default Serializers

For "basic" types

```
self.serializers: Dict[Type, Callable] = {
    dict: lambda d: json.dumps(d, default=isodate_json_encoder),
    str: str,
    int: str,
    float: str,
    bool: str,
    date: lambda d: d.isoformat(),
    datetime: lambda d: d.isoformat(),
}
```

1.2.4 Default Deserializers

For "basic" types

```
self.deserializers: Dict[Type, Callable] = {
    dict: json.loads,
    str: str,
    int: int,
    float: float,
    bool: str_to_bool,
    datetime: date_parser,
    date: lambda d: date_parser(d).date()
}
```

1.3 Subclassing

You can subclass `Serializer` or `Deserializer` to change how we detect and serialize/deserialize `Dataclasses` or `Marshmallow Schemas`.

1.3.1 Subclassing Marshmallow

Your subclass will need to implement these methods on Serializer:

- `is_marshmallow_model`
- `serialize_marshmallow_model`
- `serialize_marshmallow_model_to_dict`

And these methods on Deserializer:

- `is_marshmallow_model`
- `deserialize_marshmallow_model`

Example:

```
from typing import T, Type, Union

from flask import Blueprint, Flask
from flask_hintful import Deserializer, FlaskHintful, Serializer

class MySerializer(Serializer):
```

(continues on next page)

(continued from previous page)

```
    def is_marshmallow_model(data: T) -> bool:
        '''Returns True if Data is a Marshmallow object'''
        pass

    def serialize_marshmallow_model(data: T) -> str:
        '''Serializes Marshmallow object data into a JSON str'''
        pass

    def serialize_marshmallow_model_to_dict(data: T) -> dict:
        '''Serializes Marshmallow object data into a dict'''
        pass


class MyDeserializer(Deserializer):

    @staticmethod
    def is_marshmallow_model(type_: Type) -> bool:
        '''Returns True if type_ is a marshmallow'''
        pass

    @staticmethod
    def deserialize_marshmallow_model(data: Union[str, dict], type_: Type) -> T:
        '''Returns deserialized data into instance of type_'''
        pass


app = Flask('My API')
api = FlaskHintful(app,
                   serializer=MySerializer(),
                   deserializer=MyDeserializer())
```

1.3.2 Subclassing Dataclass

Your subclass will need to implement these methods on Serializer:

- `is_dataclass`
- `serialize_dataclass`
- `serialize_dataclass_to_dict`

And these methods on Deserializer:

- `is_dataclass`
- `deserialize_dataclass`

Example:

```
from typing import T, Type, Union

from flask import Blueprint, Flask
from flask_hintful import Deserializer, FlaskHintful, Serializer

class MySerializer(Serializer):

    def is_dataclass(data: T) -> bool:
```

(continues on next page)

(continued from previous page)

```
        '''Returns True if Data is a Dataclass'''
        pass

    def serialize_dataclass(data: T) -> str:
        '''Serializes Dataclass data into a JSON str'''
        pass

    def serialize_dataclass_to_dict(data: T) -> dict:
        '''Serializes Dataclass data into a dict'''
        pass


class MyDeserializer(Deserializer):

    @staticmethod
    def is_dataclass(type_: Type) -> bool:
        '''Returns True if type_ is a Dataclass'''
        pass

    @staticmethod
    def deserialize_dataclass(data: Union[str, dict], type_: Type) -> T:
        '''Returns deserialized data into instance of type_'''
        pass


app = Flask('My API')
api = FlaskHintful(app,
                   serializer=MySerializer(),
                   deserializer=MyDeserializer())
```

1.4 OpenApi Documentation

OpenApi documentation is automatically generated by Flask Hintful.

To access the SwaggerUI the default route is `/swagger`. The default route for the OpenApi JSON specfication is `/openapi.json`

1.4.1 Security

You can tell OpenApi which kind of security your application is expecting.

Currently, only these authentication types are supported: Bearer, ApiKey, Basic

```
app = Flask(__name__)
api = FlaskHintful(app, openapi_security=['bearer', 'apikey', 'basic'])
```

Note: This is just for OpenApi documentation, Flask-Hintful does not handle authentication/authorization.

1.4.2 Custom Routes

You can change the routes that we are serving the SwaggerUI and OpenApi JSON using these two configurations in your Flask App.

```
app = Flask(__name__)
app.config['FLASK_HINTFUL_OPENAPI_JSON_URL'] = '/custom_openapi.json'
app.config['FLASK_HINTFUL_OPENAPI_UI_URL'] = '/custom_swagger'

api = FlaskHintful(app)
```