Flask best practices

Patterns for building testable, scalable, and maintainable APIs

Posted on May 20, 2019

I love Flask. It is simple and unopinionated. A consequence of this is that as you look to scale your Flask applications, there are an infinite number of ways you could choose to structure your application. Although there will always be a subset of developers who want to tinker with new design patterns outside of what is considered "standard", by and large my experience is that developers want to focus more on *what* they are building and less on *how* they need to build it. The lack of a well-formed application structure simply adds technical debt to the developer's brain, and this housekeeping need takes away from his/her ability to create features. Therefore, I have become increasingly in favor of being opinionated about application design.

The so-called entities discussed in this post may be generated easily using flaskerize (https://github.com/apryor6/flaskerize/), specifically see the schematic section on entities (https://github.com/apryor6/flaskerize/tree/master/flaskerize/schematics#entity)

I have created a full sample project with this pattern that you can find here (https://github.com/apryor6/flask_api_example). This includes an API with 1) multiple, modular namespaces within the same RESTplus API, 2) a standalone RESTplus API attached to a blueprint with its own Swagger documentation, and 3) A third-party/installable blueprint-based API demonstrating how you might share a modular API across apps, such as if your organization has a core API that you want to reuse.

After much trial-and-error, I have come up with a set of patterns that work really well, allowing you to build highly modular and scalable Flask APIs. This pattern has been battle-tested (double emphasis on the word "test"!) and works well for a big project with a large team and can easily scale to a project of any size. Although the design is tech-stack agnostic, throughout this project I use the following technologies:

- `Flask` (d'oh)
- `pytest`, for testing
- Marshmallow for data serialization/deserialization and input validation
- SQLAlchemy as an ORM
- Flask-RESTplus for Swagger documentation
- flask_accepts (https://github.com/apryor6/flask_accepts), a library I wrote that marries Marshmallow with Flask-RESTplus, giving you the control of marshmallow with the awesome Swagger documentation from flask-RESTplus.

In a nutshell, Flask requests are routed using RESTplus Resources, input data is validated with a Marshmallow schema, some data processing occurs via a service interacting with a model, and then the output is serialized back to JSON on the way out, again using Marshmallow. All of this is documented via interactive Swagger docs, and `flask_accepts` (https://github.com/apryor6/flask_accepts) serves as glue that under-the-hood maps each Marshmallow schema into an equivalent Flask-RESTplus API model so that these two amazing-but-somewhat-incompatible technologies can happily be used together.

For the rest of this post I will walk through the system and will explain my opinions as if they are facts. Keep in mind that my opinions are, well, opinions. However, also keep in mind that this is a topic that I have spent a *lot* of time thinking about and even more time implementing in the real world; therefore, I think there is a lot of truth here. I am certainly open to debate/feedback.

High-level overview

Application code should be grouped such that files related to the same topic are localized, and *not* such that code that is functionally similar is localized. This means you should have a folder for `widgets/` that contains the services, type-definitions, etc for `widgets/`, and you should NOT have a `services/` folder where you keep all of your services. Always ask yourself:

> *If my boss asked me to delete all of the code for feature `foo`, how hard would that be?*

If you are more bullish on your features and cannot imagine a world where your boss would ever want you to delete anything, feel free to change the question to:

> *When <insert Fortune 500 company> inevitably acquires my startup, how easy will it be for me to copy/paste modules into their monorepo?*

If the answer to either of these questions is "Not very long, because I just need to grab the `foo/` folder and update one or two configurations", then you are probably doing thing correctly.

The problem with having a `services/`, `tests/`, `controllers/` folder is that when your project scales it becomes cumbersome to sift through each of these large folders to find the code that you are working on. As the project continues to grow, this problem gets worse. Conversely, if I have a `widgets/` folder where I can find a `service.py`, `controller.py`, `model.py`, etc and all of the associated tests, everything is there in one place.

Terminology

The basic unit of an API is called an entity. An entity is a thing that you want to be able to get, create, update, or edit (REST). An entity might have an underlying database table, but it does not necessarily. It could also be a derived join of several tables, data fetched from a third-party API, predictions from a machine learning model, etc.

An entity consists of (at least) the following pieces:

- Model: Python representation of entities
- Interface: Defines types that make an entity
- Controller: Orchestrates routes, services, schemas for entities
- Schema: Serialization/deserialization of entities
- Service: Performs CRUD and manipulation of entities

Note how each of these pieces is focused on only one thing. If you cannot describe a class in one sentence, it probably should be broken up.

Test files for the entity live in the same folder, and are named identically to the file they test with the addition of `_test` appended to the name (prior to the `.py`)

This means a basic entity contains the following files.

```
├── __init__.py
├── controller.py
├── controller_test.py
├── interface.py
├── interface_test.py
├── model.py
├── model_test.py
├── schema.py
├── schema_test.py
├── service.py
└── service_test.py
```

General testing comments

Tests are incredibly important because they allow you to refactor aggressively. A nice side-effect is that you verify that your code works. I am being completely serious – the verification that your code works is *just a side effect* of why you test. The test is there so that if we decide we want to change every single route in the application to drop the plural "s", I can do so in 10 minutes and be sure everything is good to go, as opposed to booting up the app and poking around for 45 minutes to confirm that I am comfortable with pushing changes.

Often, testing gets cast aside as being an additional time investment on top of writing production code, and I argue strongly against this based on two points. The first is that testing is a time investment for the future: the longer the project runs and/or the more people that touch the code, the greater the benefit is from testing due to reduction of technical debt. The second point I argue is that the additional time taken to write tests should be essentially negligible if you are familiar with writing tests. Writing tests requires an entirely different line of thinking, and often it requires learning new libraries, so, sure, it is no surprise when a group that does no testing is hesitant to pick it up because it will require some slogging through documentation and trial-and-error at first. However, once you are comfortable with writing tests it really becomes second nature. Furthermore, the time at which you are writing the production code for a feature is when the topic is freshest in your mind, and at no point in the future is it easier to write the test than right then.

I love testing. It makes me feel safe and warm and fuzzy. Have I made my case yet?

For Python testing, you should use `pytest`. Even if you use `unittest` style test cases with classes, you should still use `pytest` to run them as it understands a wide variety of testing structures and will run anything. It is also better than unittest.

Don't put your tests in a `tests/` folder. They should be alongside the production code. Although I advocate that there *not* be a top-level `tests/` folder containing all of the projects tests for the reasons of modularity, there can be a such-named folder that contains general testing utilities. For example, I create a `tests.fixtures` module that defines reusable test fixtures used throughout the app, such as a `db` fixture that creates a fresh database instance for every test. Here's what might be in that file:

```
import pytest

from app import create_app


@pytest.fixture
def app():
    return create_app('test')


@pytest.fixture
def client(app):
    return app.test_client()


@pytest.fixture
def db(app):
    from app import db
    with app.app_context():
        db.create_all()
        yield db
        db.drop_all()
        db.session.commit()
```

This takes advantage of the dependency injection capabilities of pytest and makes working with database testing very clean. These are just examples, and you will likely find yourself adding additional fixtures to this file depending on your application.

Model

The model is where the entity itself is defined in a Python representation. If you are using SQLAlchemy, this will be a class that inherits from `db.Model`. However, the model does not necessarily need to be an object from an ORM with an underlying database table; it just needs to be a python representation of a thing regardless of how you create a thing. An example:

```
class Widget(db.Model):  # type: ignore
    '''A snazzy Widget'''

    __tablename__ = 'widget'

    widget_id = Column(Integer(), primary_key=True)
    name = Column(String(255))
    purpose = Column(String(255))
```

Testing models

Minimal testing of a model consists of ensuring that it can be successfully created. This can be done by making a new `widget` as a pytest fixture and asserting that it works. Plenty of other ways to make a similar test, but I like this approach by default because if the model has additional functionality that needs to be tested then you will want a fixture so that you get a clean object for each test.

```
from pytest import fixture
from .model import Widget

@fixture
def widget() -> Widget:
    return Widget(
        widget_id=1, name='Test widget', purpose='Test purpose'
    )


def test_Widget_create(widget: Widget):
    assert widget
```

Interface

The interface is a typed definition of what is needed to create an entity. You might ask why there is a separate need for an interface if you have already defined a model, and the reason is twofold. First, the model may be built upon constructs from a third-party library that are not directly typeable. SQLAlchemy is an immediate example: you declare fields in the model that map to underlying database tables, but when I want to construct an entity itself I just need to pass in the types of data that correspond to those underlying columns. Therefore, it makes sense to separately define *what is it* (model) vs *what makes it* (interface). In addition, if you want to create an entity using packing/unpacking with `Widget(**params)` syntax, you need to provide a type definition for `params` so that mypy knows what is compatible for the constructor.

Here's a basic example. I have recently moved to defining these with `TypedDict` from `mypy_extensions`. I imagine this will eventually make its way into the Python core library, and there is currently an open pep (https://www.python.org/dev/peps/pep-0589/) for this. The advantage of `TypedDict` is that it allows you to explicitly declare what the names of the keys are. The inclusion of `total=False` makes all of the parameters optional. You can omit this if you would prefer to always provide all parameters. If you are a TypeScript user, using `total=False` is essentially the same as declaring a `Partial<Widget>`, and it allows you to create a `Widget(**params)` without explicitly providing every parameter, such as when you want to rely on default values. Note, you will still get type errors if you put an invalid type for a key or mistype a key, so there is a lot of benefit all around.

```
from mypy_extensions import TypedDict


class WidgetInterface(TypedDict, total=False):
    widget_id: int
    name: str
    purpose: str
```

Testing interfaces

Testing an interface is trivial: verify that it can be constructed and that it can be used to construct it’s corresponding model. Nothing interesting here, and I use tools like flaskerize (https://github.com/apryor6/flaskerize) to generate this kind of code.

```
from pytest import fixture
from .model import Widget
from .interface import WidgetInterface


@fixture
def interface() -> WidgetInterface:
    return WidgetInterface(
        widget_id=1, name='Test widget', purpose='Test purpose'
    )


def test_WidgetInterface_create(interface: WidgetInterface):
    assert interface


def test_WidgetInterface_works(interface: WidgetInterface):
    widget = Widget(**interface)
    assert widget
```

Schema

Marshmallow schemas are responsible for handling input/output operations for the API and are where renaming conventions are handled. JavaScript is the language of the web, and in JavaScript the preferred style for variables is lowerCamelCase, whereas in python snake_case is preferred. The schema is the perfect place to handle this translation as it is the layer between your API and the outside world. Marshmallow is an amazing technology that you can learn more about here (https://marshmallow.readthedocs.io/en/3.0/), and it can do an enormous number of tasks that are much more complicated than the example I give below. For example, you can use the `post_load` hook from Marshmallow to declare what object(s) to create when the `Schema.load` method is invoked, which will automatically synergize with `flask_accepts` to provide a fully-instantiated object in `request.parsed_obj`, for example.

```
from marshmallow import fields, Schema


class WidgetSchema(Schema):
    '''Widget schema'''

    widgetId = fields.Number(attribute='widget_id')
    name = fields.String(attribute='name')
    purpose = fields.String(attribute='purpose')
```

Here we are saying that the incoming POST request is expected to have (up to) 3 fields with the names `widgetId`, `name`, `purpose`, and that these map to the attributes `widget_id`, `name`, and `purpose` of the serialized result. Note the case-mapping of `widgetId -> widget_id`. In this case the `attribute` parameter is redundant for `name` and `purpose` because they are the same in both styles, but I prefer to keep this for consistency (and, again, I am generating this code with flaskerize (https://github.com/apryor6/flaskerize))

The schema will be associated with a `Resource` in the controller through use of the `@accepts` (https://github.com/apryor6/flask_accepts/blob/master/flask_accepts/decorators/decorators.py) and `@responds` (https://github.com/apryor6/flask_accepts/blob/master/flask_accepts/decorators/decorators.py) decorators from flask_accepts (https://github.com/apryor6/flask_accepts), which declare what format the data should be for input and what the endpoint returns.

Testing schemas

Schema testing is very similar to interface testing with the wrinkle that we are also verifying that the names are mapped across case-styles correctly. It's important here that you don't write tests that are verifying that Marshmallow works correctly, instead the tests should be checking that *you* declared the schema properly so that the input payload that you want to provide actually builds an object.

If I'm being honest, the value of the tests I have here for schemas and interfaces is small, so if you think they are unnecessary then I have no problem with you skipping them. The tests further below for controllers, services, etc, however, are absolutely necessary.

```
from pytest import fixture

from .model import Widget
from .schema import WidgetSchema
from .interface import WidgetInterface


@fixture
def schema() -> WidgetSchema:
    return WidgetSchema()


def test_WidgetSchema_create(schema: WidgetSchema):
    assert schema


def test_WidgetSchema_works(schema: WidgetSchema):
    params: WidgetInterface = schema.load({
        'widgetId': '123',
        'name': 'Test widget',
        'purpose': 'Test purpose'
    }).data
    widget = Widget(**params)

    assert widget.widget_id == 123
    assert widget.name == 'Test widget'
    assert widget.purpose == 'Test purpose'
```

Service

The service is responsible for interacting with the entity. This includes managing CRUD (Create, Read, Update, Delete) operations, fetching data via a query, performing some pandas DataFrame manipulation, getting predictions from a machine learning model, hitting an external API, etc. The service should be the meat-and-potatoes of data processing inside of a route. Your services should be kept modular. Services can (and often, should) depend on other services. When you begin to have interservice dependencies you must use dependency injection (DI) or your system will not be maintainable or testable. I will not go into the details of DI in this post, but I will refer you to Flask-Injector (https://github.com/alecthomas/flask_injector). You could also roll your own simple system by attaching services to the app object at configuration time (here (https://speakerdeck.com/mitsuhiko/advanced-flask-patterns) is a very nice presentation on this topic).

```
from app import db
from typing import List
from .model import Widget
from .interface import WidgetInterface


class WidgetService():
    @staticmethod
    def get_all() -> List[Widget]:
        return Widget.query.all()

    @staticmethod
    def get_by_id(widget_id: int) -> Widget:
        return Widget.query.get(widget_id)

    @staticmethod
    def update(widget: Widget, Widget_change_updates: WidgetInterface) -> Widget
        widget.update(Widget_change_updates)
        db.session.commit()
        return widget

    @staticmethod
    def delete_by_id(widget_id: int) -> List[int]:
        widget = Widget.query.filter(Widget.widget_id == widget_id).first()
        if not widget:
            return []
        db.session.delete(widget)
        db.session.commit()
        return [widget_id]

    @staticmethod
    def create(new_attrs: WidgetInterface) -> Widget:
        new_widget = Widget(
            name=new_attrs['name'],
            purpose=new_attrs['purpose']
        )

        db.session.add(new_widget)
        db.session.commit()
```

```
        return new_widget
```

The exact methods for services will differ depending upon the situation; however, for the common case of a CRUD application these operations are pretty typical. In this example the underlying model is based in SQLAlchemy, so, naturally, the service is using the SQLAlchemy ORM patterns to manipulate objects. The service could also be using pandas to manipulate dataframes, apply some computer vision library, etc.

Testing services

Testing services is still relatively simple but important. We want to verify that the service is perform CRUD operations correctly, so for most of the tests we will want to use our `db` fixture so that we have a clean database. We then do some arranging to setup any objects we need, make a service call, and then assert that the result is what we expect. For example, in `test_delete_by_id` below we first add a pair of objects to the database (arrange) then delete one using it's ID (act) and finally assert that only one object remains and that it is the one we did not delete.

```
from flask_sqlalchemy import SQLAlchemy
from typing import List
from app.test.fixtures import app, db  # noqa
from .model import Widget
from .service import WidgetService  # noqa
from .interface import WidgetInterface


def test_get_all(db: SQLAlchemy):  # noqa
    yin: Widget = Widget(widget_id=1, name='Yin', purpose='thing 1')
    yang: Widget = Widget(widget_id=2, name='Yang', purpose='thing 2')
    db.session.add(yin)
    db.session.add(yang)
    db.session.commit()

    results: List[Widget] = WidgetService.get_all()

    assert len(results) == 2
    assert yin in results and yang in results


def test_update(db: SQLAlchemy):  # noqa
    yin: Widget = Widget(widget_id=1, name='Yin', purpose='thing 1')

    db.session.add(yin)
    db.session.commit()
    updates: WidgetInterface = dict(name='New Widget name')

    WidgetService.update(yin, updates)

    result: Widget = Widget.query.get(yin.widget_id)
    assert result.name == 'New Widget name'


def test_delete_by_id(db: SQLAlchemy):  # noqa
    yin: Widget = Widget(widget_id=1, name='Yin', purpose='thing 1')
    yang: Widget = Widget(widget_id=2, name='Yang', purpose='thing 2')
    db.session.add(yin)
    db.session.add(yang)
```

```
    db.session.commit()

    WidgetService.delete_by_id(1)
    db.session.commit()

    results: List[Widget] = Widget.query.all()

    assert len(results) == 1
    assert yin not in results and yang in results


def test_create(db: SQLAlchemy):  # noqa

    yin: WidgetInterface = dict(name='Fancy new widget', purpose='Fancy new purp
    WidgetService.create(yin)
    results: List[Widget] = Widget.query.all()

    assert len(results) == 1

    for k in yin.keys():
        assert getattr(results[0], k) == yin[k]
```

There's a lot of different ways to do this assertions for object equality. You can manually add assertions for each, iterate over key-value pairs in dicts, implement a `__eq__` method in the underlying model and directly use `==`, etc. I think all of these are fine as long as verbosity is limited and you are only testing one concept per test.

Controller

The controller is responsible for coordinating a Flask route(s), services, schemas, and other components that are needed to produce a response. Because we are using Flask-RESTplus for Swagger, this means the controller is also responsible for providing documentation. In order to prevent controllers from becoming cluttered with a high number of routes (which is hard to maintain), it is important that each entity be scoped such that it has its own Flask-RESTplus namespace. This namespace is used to attach routes to each of the resources. Although it is not necessary for the routes to be Flask-RESTplus Resources (meaning it is a class that inherits from Resource and implements get/post/put/delete methods), I would strongly recommend it for several

reasons. Firstly, Swagger is amazing and you miss out on it if you use `@app.route` directly. Second, if you only provide a single function per route with Flask, but want to support multiple HTTP methods, you have to implement some if-else logic to check which method has been invoked. Flask-RESTplus abstracts this away, and, sure, you could also do that yourself, but why bother reinventing the wheel?

```
from flask import request
from flask_accepts import accepts, responds
from flask_restplus import Namespace, Resource
from flask.wrappers import Response
from typing import List

from .schema import WidgetSchema
from .service import WidgetService
from .model import Widget
from .interface import WidgetInterface

api = Namespace('Widget', description='Single namespace, single entity')  # noqa


@api.route('/')
class WidgetResource(Resource):
    '''Widgets'''

    @responds(schema=WidgetSchema, many=True)
    def get(self) -> List[Widget]:
        '''Get all Widgets'''

        return WidgetService.get_all()

    @accepts(schema=WidgetSchema, api=api)
    @responds(schema=WidgetSchema)
    def post(self) -> Widget:
        '''Create a Single Widget'''

        return WidgetService.create(request.parsed_obj)


@api.route('/<int:widgetId>')
@api.param('widgetId', 'Widget database ID')
class WidgetIdResource(Resource):
    @responds(schema=WidgetSchema)
    def get(self, widgetId: int) -> Widget:
        '''Get Single Widget'''

        return WidgetService.get_by_id(widgetId)
```

```
    def delete(self, widgetId: int) -> Response:
        '''Delete Single Widget'''
        from flask import jsonify

        id = WidgetService.delete_by_id(widgetId)
        return jsonify(dict(status='Success', id=id))

    @accepts(schema=WidgetSchema, api=api)
    @responds(schema=WidgetSchema)
    def put(self, widgetId: int) -> Widget:
        '''Update Single Widget'''

        changes: WidgetInterface = request.parsed_obj
        Widget = WidgetService.get_by_id(widgetId)
        return WidgetService.update(Widget, changes)
```

One noteworthy difference here from the Flask-RESTplus documentation is that I am using `flask_accepts`, which allows me to get the same documentation from Swagger that I would get with `api.expect` *without* having to write two models (one for Marshmallow and one for RESTplus). The `@accepts` decorator declares what is to be expected in the body of a POST/PUT request, and the parsed object will be attached to the `flask.request` object as `request.parsed_obj` and is available inside the body of the decorated function. Likewise, the `@responds` decorator declares what is to be used to serialize the output object. The function you are decorating with `@responds` should return the actual object you want to return, and then `@responds` will handle the serialization and ultimately return the object in a `jsonify` call with a `200` status if everything works properly. For those used to `Flask-RESTplus`, the `@accepts` decorator is like `@api.expect` and `@responds` is like `@marshal_with`.

The one inconvenience about `flask_accepts` is that you have to pass the api object into the `@accepts` decorator so that the documentation can be registered correctly (`flask_accepts` also works with non-RESTplus routes and thus needs a way to know where to attach Swagger); however, I think this is a minor detail.

Testing controllers

Testing controllers is trickier than the previous testing tasks as it requires mocking a service. Why do we need to mock a service? So that we can keep the test focused on a single concept. The controller is responsible for telling the service what to do, what to do it with, and what to do with the response, but it is *not* responsible for the functionality of the service itself – that should be tested in service_test.py. Therefore, we want to mock the values provided by the service so that we can control them and then verify that the controller is doing what it is supposed to given that the service works as expected.

I'll walkthrough a single test here in detail, but you can find a full set of examples for each CRUD operation in the full example project (https://github.com/apryor6/flask_api_example).

```
from unittest.mock import patch
from flask.testing import FlaskClient

from app.test.fixtures import client, app  # noqa
from .service import WidgetService
from .schema import WidgetSchema
from .model import Widget
from .interface import WidgetInterface
from . import BASE_ROUTE


def make_widget(id: int = 123, name: str = 'Test widget',
                purpose: str = 'Test purpose') -> Widget:
    return Widget(
        widget_id=id, name=name, purpose=purpose
    )


class TestWidgetResource:
    @patch.object(WidgetService, 'get_all',
                  lambda: [make_widget(123, name='Test Widget 1'),
                           make_widget(456, name='Test Widget 2')])
    def test_get(self, client: FlaskClient):  # noqa
        with client:
            results = client.get(f'/api/{BASE_ROUTE}',
                                 follow_redirects=True).get_json()
            expected = WidgetSchema(many=True).dump(
                [make_widget(123, name='Test Widget 1'),
                 make_widget(456, name='Test Widget 2')]
            ).data
            for r in results:
                assert r in expected
```

Let's start with simulating an API request, which starts with the request. Flask provides a testing client so that we can properly simulate all of the necessary context and globals that are provided in a real Flask app. This object is available as `app.test_client` and, in my case, is provided through the `app.test` module as the `client` fixture, which can be imported and injected into

any pytest test. You then wrap the test in `with client:`, which establishes the necessary request context. Inside of the test we make our `get` request (in this example to the root route of our `widget` API, which will be attached at `/api/widget/`).

If you glance back at the previous section, you'll see that this route simply calls `WidgetService.get_all()` and returns the result as a `List[Widget]` (you could also infer this return type because of the response decorator contains `many=True` with `WidgetSchema`). For the test, we want to mock out this service call so that we can control the return value. This is done with `patch` from `unittest.mock`. In this project I am using static classes for services (you don't have to), so I am using the `patch.object` decorator, which will take its first argument (a class) and override the method provided in the second parameter (as a string) with the implementation in the third argument. I like to use a lambda for this third argument so that everything is kept close together. This patch will only last for the current test, so subsequent tests will see the Service revert to it's normal state.

For the expected result, I manually create the relevant schema and dump out the same result as what is returned from my mock. Note that this test is dependent upon the fact that `@responds` is working as expected, which is totally fine. Remember, don't test other people's code. So our final test here is asserting that if a user makes a get request that they will in fact receive the properly serialized version of `WidgetService.get_all()`, which is a single concept despite the complexity of the controller.

Apis, Namespaces, Blueprints.. oh my

Comments on app configuration, route registration, and Swagger

Route declaration in large Flask apps is tricky if you do not do it correctly. For one, it is very easy to generate circular import situations. For example, this occurs if you declare an Api or Namespace at the top of a module and then try to import that api into submodules that implement functionality that is to be barreled-up and re-exported at the top-level. A terrible workaround for this is to define the Api/namespace *prior* to importing the dependent functionality, resulting in fragile and confusing declaration of variables sandwiched in between sets of imports. This is actually what is suggested in one example here (https://flask-restplus.readthedocs.io/en/stable/scaling.html#multiple-apis-with-reusable-namespaces), but I would not recommend it. There is a better way!

To allow for clean route registration, you should use the factory method for creating your Flask apps (which you are already doing because you are testing everything, right!?), and then following the same line of thinking we want to delay route registration until the app is created. After all, that's the point of the lazy factory pattern – we want to allow for our application to be configurable by telling it how to behave at the last second, *after* all of the other code has been defined. Same idea here. To do this, I add a `register_routes` method in the `__init__.py` file for each of the entities. This function imports the various Namespaces from each controller and attaches them to the parent application, which is passed in the parameters.

```
# app/widget/__init__.py

from .model import Widget  # noqa
from .schema import WidgetSchema  # noqa
BASE_ROUTE = 'widget'


def register_routes(api, app, root='api'):
    from .controller import api as widget_api

    api.add_namespace(widget_api, path=f'/{root}/{BASE_ROUTE}')
```

My preferred `register_routes` function takes in a Flask-RESTplus api, the Flask app itself, and a root url to attach the routes at. I chose these arguments because depending upon the type of attachment (blueprint vs namespace-only), you need different pieces, and I prefer to have the same API for all registration even if, for example, I am registering a blueprint with its own Api and need the `app` instead of the `api` . Feel free to change it if you feel strongly, but I prefer consistency as it lowers mental baggage.

Then, within the top-level `create_app` function the `register_routes` functions for all of the various entites/modules I want to attach are imported (with helpful aliases) and applied sequentially. To prevent these calls from cluttering up the root `__init__.py` , I move them into a separate `routes.py` file, which becomes the master routing configuration of the entire application.

```
# routes.py -- where the main app is configured for routing
def register_routes(api, app, root='api'):
    from app.widget import register_routes as attach_widget
    from app.fizz import register_routes as attach_fizz
    from app.other_api import register_routes as attach_other_api
    from app.third_party.app import create_bp

    # Add routes
    attach_widget(api, app)
    attach_fizz(api, app)
    attach_other_api(api, app)
    app.register_blueprint(create_bp(), url_prefix='/third_party')
```

Multiple namespaces can either be combined into a single API or attached to separate ones using Flask blueprints followed by registration of each of the blueprints at the app level. In the above, `widget` is a single entity with a single namespace, `fizz` is a module with two namespaces that are both attached to the same Flask-RESTplus Api (which means they share a Swagger instance), the `other_api` is registered as a blueprint with it's own Swagger documentation, and finally a simulated third-party library is attached with `app.register_blueprint` assuming that the third-party library exports some way of accessing the fully-formed blueprint. If it does not, you could write your own wrapper module and accomplish the same thing, but I would point out if you are writing your own module you could just follow the `attach_other_api` function and perform the `app.register_blueprint` call within an `__init__.py`, hiding the implementation details from the route declaration unless it is unavoidable.

How do you know whether to use one Api, one blueprint, many blueprints, many Apis? The choice for how to break up your app depends upon the context of the project. If you use a single API, then all of the Swagger documentation is on one page, which is both the primary advantage and disadvantage depending on the size of the project, and thus the decision is a judgement call. A good rule of thumb is that as long as the number of entities you have is limited enough such that the Swagger collapsible tabs can fit on a page without scrolling, then you are fine to load everything into one API. Beyond that, it is time to start using blueprints.

Conclusion

And that's it. Rinse and repeat building an interface, model, schema, service, controller over and over again and you will have yourself a highly scalable, modular, and, most importantly, testable API. If you think this pattern sounds boring, that is the beauty of it. I can work on a massive project all day long and barely exercise any brain power on *how* I am developing, and instead spend my time on what matters – *what* I am developing.

Tags: flask, python, backend development, api, data science, software engineering

(ht… (ht… (ht…

← PREVIOUS POST (/2019-05-15-RXJS-FLATTENING-OPERATORS/)

NEXT POST → (/2019-08-23-FLASKERIZE-UPDATE-PROPOSAL/)

(https://github.com/apryor6) (https://twitter.com/pryor_aj) (mailto:apryor6@gmail.com)
(https://linkedin.com/in/alan-pryor-02a52b57)