Previous Next

Writing your first endpoints

Estimated Time: 2-3 hours

Over the next three assignments you'll create a simple publishing API which lets you store and retrieve posts. It could be used as the basis for a blog, CRM, or eBook.

You'll start by writing endpoints (URLs designed so you can access and modify data) to retrieve either a list of all of the posts, or a single post.

In the Blogful project we spent quite a lot of time setting up boilerplate code which you'll need for every Flask project. Because you know this code fairly well already you can go ahead and clone it from this repository that already contains the boilerplate. You'll use this as a basis for the API. Be sure to take a look through the code to double check that you are familiar with how it is works. To get started on this project:

1. Clone the starter repository (`git clone https://github.com/oampo/pip-posts-project-template posts`)
2. Move into the project's directory (`cd posts`)
3. Set up a virtualenv for the project (`python3 -m venv env`)
4. Activate the virtualenv (`source env/bin/activate`)
5. Install the project's dependencies (`pip install -r requirements.txt`)

6. Set up a database (`createdb posts`)
7. Set up a test database (`createdb posts-test`)

Writing API Endpoints Test first

The first endpoint you're going to work on will return a list of posts in the JSON format. JSON is a serialization format that allows you to encode and decode data easily in pretty much any programming language. It has become a kind of lingua-franca of the web, and you will find that most APIs work with JSON encoded data.

For this project, you'll use the test first approach. In the *tests/api_tests.py* file there is a test fixture already set up. Add a test to this that ensures:

- A request to your endpoint is successful
- The endpoint is returning JSON as expected
- The endpoint returns an empty list, as you have no posts

The test should look something like this:

```
def test_get_empty_posts(self):
    """ Getting posts from an empty database """
    response = self.client.get("/api/posts")

    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.mimetype, "application/json")

    data = json.loads(response.data.decode("ascii"))
    self.assertEqual(data, [])
```

Note

In this project, you'll be using Nose which is a framework that wraps Python's built in UnitTest framework. One of Nose's key features is

auto-discovery of your test suites, meaning you only need to point it to a single parent directory, and it will find and run all of the tests within that directory and any children. You'll use the `nosetests` command to run your tests, but Nose won't change how you write the actual tests. Nose is listed as one of the requirements for this app in requirements.txt.

Add this test to the TestAPI class, and try running it using `nosetests tests`. Notice how you don't need to set the `PYTHONPATH` environment variable – Nose takes care of setting this for you. The test should fail at the first assertion; this is what we would expect as you haven't written the endpoint yet.

There are a few things to notice here:

1. You are using the test client to make a GET request to `/api/posts`. This should give you a clue about what the endpoint route will look like.
2. You're checking that the status returned by our endpoint is `200 OK`. This means that your request has worked correctly.
3. You're checking that the endpoint has returned JSON by looking at the response mimetype. This should give you another hint about something you might need in your endpoint.
4. You're decoding the response data, first converting the bytes into text according to the ASCII standard, then using `json.loads`. Internet protocols transmit and receive bytes, but JSON is a textual standard; even though it may seem obvious to a human, the text string "Hello" is very different from the byte string b"\x48\x65\x6c\x6c\x6f".
5. You're checking that the JSON contains an empty list.

Making it pass

Now that you have your test in place, try to make it pass. The first thing to do is to set up a route in the *posts/api.py* file. This should look something like:

```
@app.route("/api/posts", methods=["GET"])
def posts_get():
    """ Get a list of posts """
    pass
```

Notice how you're using the `methods` parameter to specify that this endpoint will only handle GET requests.

Now add a body to your function which will return the empty list as JSON:

```
@app.route("/api/posts", methods=["GET"])
def posts_get():
    """ Get a list of posts """
    data = json.dumps([])
    return Response(data, 200, mimetype="application/json")
```

Here you manually construct a Response object which returns:

- An empty list encoded as JSON using `json.dumps`.
- A status of 200 OK.
- The correct mimetype for JSON.

Try running the test again (`nosetests tests`). Hopefully it will pass this time. If so, congratulations – you've just written your first API endpoint!

Adding a model

Now you know the basics of constructing an endpoint, so let's expand the code to make it work with some actual data. The first step is to create a SQLAlchemy model to represent your posts. To keep things simple you'll limit the model to three fields: an id, the post title, and some body text.

The *posts/database.py* already contains the setup code for a database connection, so all you need to do is add the following to your *posts/models.py* file:

```
class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True)
    title = Column(String(128))
    body = Column(String(1024))
```

This should all look pretty familiar to you. If not have a quick look back over the material on SQLAlchemy from earlier in the course.

At this stage you can add one final piece of code which will come in handy shortly. To allow you to easily convert your SQLAlchemy model into JSON, you'll add an `as_dictionary` method to the `Post` class, which will return the post's data as a Python dictionary. This should look something like:

```
    def as_dictionary(self):
        post = {
            "id": self.id,
            "title": self.title,
            "body": self.body
        }
        return post
```

A trickier test

Now that your model is in place, make a new test which adds a couple of posts and makes sure that your endpoint returns them correctly.

First, create the test in *tests/api_tests.py* and add code to create the posts:

```
    def test_get_posts(self):
        """ Getting posts from a populated database """
```

```
postA = models.Post(title="Example Post A", body="Just a test"
postB = models.Post(title="Example Post B", body="Still a test

session.add_all([postA, postB])
session.commit()
```

Then you can copy the request and assertions from `test_empty_posts` into your new test, changing the code so that it checks that the posts are returned correctly:

```
response = self.client.get("/api/posts")

self.assertEqual(response.status_code, 200)
self.assertEqual(response.mimetype, "application/json")

data = json.loads(response.data.decode("ascii"))
self.assertEqual(len(data), 2)

postA = data[0]
self.assertEqual(postA["title"], "Example Post A")
self.assertEqual(postA["body"], "Just a test")

postB = data[1]
self.assertEqual(postB["title"], "Example Post B")
self.assertEqual(postB["body"], "Still a test")
```

Try running the tests again. This new test should make it past where you check the status and mimetype of the response. It should only fail when you start to check that the correct data was returned.

Returning data

Next, try to update your endpoint so that it passes this more complex test. In order to make the test pass your endpoint needs to:

- Query the database for all of the posts
- Convert the list of posts to JSON

- Return this list in your response

The modified endpoint should look something like this:

```
@app.route("/api/posts", methods=["GET"])
def posts_get():
    """ Get a list of posts """

    # Get the posts from the database
    posts = session.query(models.Post).order_by(models.Post.id)

    # Convert the posts to JSON and return a response
    data = json.dumps([post.as_dictionary() for post in posts])
    return Response(data, 200, mimetype="application/json")
```

Here you use a list comprehension to create a list of dictionaries containing the data for the posts. List comprehensions provide a simple way to create populated lists from another iterable object. So this code:

```
post_dictionaries = [post.as_dictionary() for post in posts]
```

Is equivalent to this:

```
post_dictionaries = []
for post in posts:
    post_dictionaries.append(post.as_dictionary())
```

You may be wondering why you don't just call `json.dumps(posts)` to encode your posts as JSON. Unfortunately this will not work because the json module does not know how to serialize SQLAlchemy models. The solution is to convert your data to Python dictionaries, which the json module is happy to serialize.

Try running the tests again. Hopefully the second test will now pass, showing that you can retrieve a list of posts from your API.

GETting a single post

Now let's look at the endpoint for retrieving a single post. First of all you can add a couple of tests: one for when you request a post that exists, and a second for requesting a non-existent post.

```
    def test_get_post(self):
        """ Getting a single post from a populated database """
        postA = models.Post(title="Example Post A", body="Just a test"
        postB = models.Post(title="Example Post B", body="Still a test

        session.add_all([postA, postB])
        session.commit()

        response = self.client.get("/api/posts/{}".format(postB.id))

        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.mimetype, "application/json")

        post = json.loads(response.data.decode("ascii"))
        self.assertEqual(post["title"], "Example Post B")
        self.assertEqual(post["body"], "Still a test")

    def test_get_non_existent_post(self):
        """ Getting a single post which doesn't exist """
        response = self.client.get("/api/posts/1")

        self.assertEqual(response.status_code, 404)
        self.assertEqual(response.mimetype, "application/json")

        data = json.loads(response.data.decode("ascii"))
        self.assertEqual(data["message"], "Could not find post with id
```

The `test_get_post` method should look very familiar as it is almost identical to the `test_get_posts` method you wrote earlier. The one thing to note here is the format of the route: you're accessing the post at `/api/posts/<id>`.

The test for a non-existent post is more interesting. When a post doesn't exist, check that a `404 Not Found` status is returned. Also look for a JSON encoded message in the response body with the following format:

```
{
    "message": "A description of the error message"
}
```

This is a common pattern for returning errors in an API. The message is often accompanied by a more detailed error code and a link to documentation describing how the problem can be resolved.

With your tests written you can try to create the endpoint:

```
@app.route("/api/posts/<int:id>", methods=["GET"])
def post_get(id):
    """ Single post endpoint """
    # Get the post from the database
    post = session.query(models.Post).get(id)

    # Check whether the post exists
    # If not return a 404 with a helpful message
    if not post:
        message = "Could not find post with id {}".format(id)
        data = json.dumps({"message": message})
        return Response(data, 404, mimetype="application/json")

    # Return the post as JSON
    data = json.dumps(post.as_dictionary())
    return Response(data, 200, mimetype="application/json")
```

Here you're running a query against your database to find the post with the correct ID. If the post doesn't exist then you construct your error message and return it with a `404` status. If the post exists, you convert it to JSON and return it with a `200` status.

Try running the tests one more time to make sure your new endpoint is working correctly.

Good behavior

By this point you have a great API which allows you to retrieve both individual posts and a complete list of all of the posts. But before moving on, you'll need to do one more quick thing to make your API behave nicely.

When a client (such as a browser) makes a request to a website it provides an Accept header. This is a list telling the server which types of content the client understands, and which it would prefer to use. As your API only returns JSON you'll want to make sure that the client is happy to accept that JSON before you start sending it back. If the client doesn't like JSON you should return an error message.

The Flask test client doesn't send an Accept header as default. As a first step you need to update the requests in your tests to send the correct Accept header. For example in `test_get_empty_posts` you need to change your request to look something like this:

```
response = self.client.get("/api/posts",
    headers=[("Accept", "application/json")]
)
```

Next you can write a test which sends an unsupported Accept header. Rather than getting a `200 OK` response you should expect to see a `406 Not Acceptable`, which tells the client that you cannot send it data in a format which it will accept.

```
def test_unsupported_accept_header(self):
    response = self.client.get("/api/posts",
        headers=[("Accept", "application/xml")]
    )

    self.assertEqual(response.status_code, 406)
    self.assertEqual(response.mimetype, "application/json")

    data = json.loads(response.data.decode("ascii"))
    self.assertEqual(data["message"],
                     "Request must accept application/json data")
```

Notice how you check for an error message in the same format as you use in the `post_get` method.

With your test written you can try to find a way to return the error when an unsupported Accept header is detected. One option would be to add an `if` statement to each endpoint checking the Accept header. This would work well, but you would have to write the same code for every endpoint, which would not adhere to the principle of DRY (Don't Repeat Yourself). A nicer alternative is to write a decorator which performs the same role.

Try adding the following decorator to the *posts/decorators.py* file:

```
def accept(mimetype):
    def decorator(func):
        """
        Decorator which returns a 406 Not Acceptable if the client won
        a certain mimetype
        """
        @wraps(func)
        def wrapper(*args, **kwargs):
            if mimetype in request.accept_mimetypes:
                return func(*args, **kwargs)
            message = "Request must accept {} data".format(mimetype)
            data = json.dumps({"message": message})
            return Response(data, 406, mimetype="application/json")
        return wrapper
    return decorator
```

While there is quite a lot of syntax here the important part is pretty simple. Inside the `wrapper` function you check whether the supplied mimetype is in the Accept header. If it is, you continue with the route as usual (which is what `return func(*args, **kwargs)` does). If the mimetype is not in the header then you send a response with your JSON message and a `406` status code.

Then all you have to do is add the decorator to your endpoints, for example:

```
@app.route("/api/posts", methods=["GET"])
@decorators.accept("application/json")
def posts_get():
    """ Get a list of posts """

    ...
```

Try running the tests one more time to make sure that you are correctly handling the Accept headers. If that all works then we've successfully written our first endpoints, tested that they work correctly, and made them behave nicely.

In the next assignment you're going to introduce query strings to our API, allowing you to filter the lists of posts which you retrieve.

Try It!

Write tests and an endpoint for deleting a single post. The endpoint should be at `/api/posts/<id>`, and should only accept a `DELETE` request. Remember to test for and handle any errors which you think could arise with your endpoint.

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Mark as completed

Previous Next