Sending Data to an API with POST

Estimated Time: 2-3 hours

In the previous two assignments you created an API to retrieve posts. While that is looking great, it will be much more useful when you can add posts using your API. In this assignment you'll add posts by sending POST requests to end endpoint.

Test first

To get started, write a quick test for adding a post. To add the post you are going to make a POST request to /api/posts. The request will contains the title and body of your post as JSON. Notice how the endpoint URL is the same as the one for getting a list of posts; the only difference is that you are sending a POST rather than a GET request.

Your test needs to check that:

- The request to your endpoint is successful
- The post is getting added to the database as expected
- The endpoint is returning the correct JSON
- The endpoint is setting the correct Location header (more on this later)

> Add the following test to the `TestAPI` class in *tests/api_tests.py*.

```
    def test_post_post(self):
        """ Posting a new post """
        data = {
            "title": "Example Post",
            "body": "Just a test"
        }

        response = self.client.post("/api/posts",
            data=json.dumps(data),
            content_type="application/json",
            headers=[("Accept", "application/json")]
        )

        self.assertEqual(response.status_code, 201)
        self.assertEqual(response.mimetype, "application/json")
        self.assertEqual(urlparse(response.headers.get("Location")).pa
                         "/api/posts/1")

        data = json.loads(response.data.decode("ascii"))
        self.assertEqual(data["id"], 1)
        self.assertEqual(data["title"], "Example Post")
        self.assertEqual(data["body"], "Just a test")

        posts = session.query(models.Post).all()
        self.assertEqual(len(posts), 1)

        post = posts[0]
        self.assertEqual(post.title, "Example Post")
        self.assertEqual(post.body, "Just a test")
```

While quite a lot of this is similar to the tests for the GET endpoints, there are a few new things to notice.

When you make the post request you provide two new pieces of information:

- data: a JSON serialized representation of the post.
- content_type: a string describing the content type of your request.

The response that you expect from the server is also more complex. Your first check on the response makes sure that you get a `201 Created` status. This status is used to tell the client that the post has been created successfully.

Your next check makes sure that a Location header has been set, and points to the endpoint which you can use to retrieve the new post. The Location header is used to point the client towards the location of the new resource. In order to access the header you use the `response.headers.get` method. You then use Python's urlparse module to remove the server address from the Location header. This allows you to compare the returned location with your expected endpoint address.

The final checks on the response make sure that it contains JSON holding the correct representation of the post.

To make sure that the item was added to the database correctly, you perform a simple query on the database. Then check that the details stored match up with what was in your input.

Try running the test using `nosetests tests` . It should fail at the first assertion, as you haven't added the endpoint yet.

Making it work

With your test in place it's time to write the endpoint in *posts/api.py*.

```
@app.route("/api/posts", methods=["POST"])
@decorators.accept("application/json")
def posts_post():
    """ Add a new post """
    data = request.json

    # Add the post to the database
    post = models.Post(title=data["title"], body=data["body"])
    session.add(post)
    session.commit()
    # Return a 201 Created, containing the post as JSON and with the
    # Location header set to the location of the post
    data = json.dumps(post.as_dictionary())
    headers = {"Location": url_for("post_get", id=post.id)}
    return Response(data, 201, headers=headers,
                    mimetype="application/json")
```

Use the `request.json` dictionary to access the data which you passed in to the endpoint. This data is then used to create a new `Post` instance, which is added to the database. You then create and return the response as usual, this time adding in a headers dictionary containing our Location header. In order to find the correct location you use Flask's `url_for` function, which determines the URL for a specific endpoint.

Try running the tests again. If everything is working correctly you should now be able to both add and retrieve a post using your API.

Good behavior Pt 1. - Mimetypes

Just like with your GET endpoints there are a couple of additional steps you can take to make your POST endpoint more robust and helpful to clients.

Currently you use the `accept` decorator to ensure that the client can deal with the JSON that you return. But what if a client tries to send you data that you don't understand? At the moment your code will fail, so let's try to rectify that.

First write a test which tries to send data of an incorrect type to your endpoint:

```
def test_unsupported_mimetype(self):
    data = "<xml></xml>"
    response = self.client.post("/api/posts",
        data=json.dumps(data),
        content_type="application/xml",
        headers=[("Accept", "application/json")]
    )
```

```
        self.assertEqual(response.status_code, 415)
        self.assertEqual(response.mimetype, "application/json")

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

When you send the XML data you expect to see a `415 Unsupported Media Type` error, which tells the client that the data was in the wrong format. You also check that you return a JSON error message in the usual format.

Try running the test. It should cause an error when the endpoint tries to access the non-existent JSON data. You can write a decorator to fix that. It will check whether the mimetype is set to `application/json`, and if not it will return an error.

```
def require(mimetype):
    def decorator(func):
        """
        Decorator which returns a 415 Unsupported Media Type if the cl
        something other than a certain mimetype
        """
        @wraps(func)
        def wrapper(*args, **kwargs):
            if (request.mimetype ==  mimetype):
                return func(*args, **kwargs)
            message = "Request must contain {} data".format(mimetype)
            data = json.dumps({"message": message})
            return Response(data, 415, mimetype="application/json")
        return wrapper
    return decorator
```

This should look pretty familiar; it is almost exactly the same as your `accept` decorator, with a slightly different condition.

Try using this to decorate the `post_post` endpoint and then run the tests again. You should now be dealing with incorrect input formats in a well-behaved way.

Good behavior Pt 2. - Validation

You've seen how to handle clients sending the wrong type of data, but what if the data they are sending is valid JSON, but is in the wrong format. For example, what if rather than a string for the title the client is giving us a list?

In order to handle these conditions gracefully you'll need to validate the data that the client sends. Fortunately there is an excellent module for validating JSON data called *jsonschema*.

First, write a couple of tests checking that your endpoint returns an error if the client sends invalid data or data that's missing important information.

```
    def test_invalid_data(self):
        """ Posting a post with an invalid body """
        data = {
            "title": "Example Post",
            "body": 32
        }

        response = self.client.post("/api/posts",
            data=json.dumps(data),
            content_type="application/json",
            headers=[("Accept", "application/json")]
        )

        self.assertEqual(response.status_code, 422)

        data = json.loads(response.data.decode("ascii"))
        self.assertEqual(data["message"], "32 is not of type 'string'"

    def test_missing_data(self):
        """ Posting a post with a missing body """
        data = {
            "title": "Example Post",
        }

        response = self.client.post("/api/posts",
            data=json.dumps(data),
            content_type="application/json",
            headers=[("Accept", "application/json")]
        )

        self.assertEqual(response.status_code, 422)

        data = json.loads(response.data.decode("ascii"))
        self.assertEqual(data["message"], "'body' is a required proper
```

The tests check that the endpoint will return a helpful error message and a `422 Unprocessable Entity` status. This tells the client that you couldn't correctly process the data provided.

Now, you will use jsonschema to check the data that you're given, and make sure it complies with the format you're looking for. The jsonschema library uses a complete description of your expected object to check whether the data is valid. The description (known as the schema) is described using a Python dictionary. Try adding the following schema describing your post format to the top of the *posts/api.py* file:

```
# JSON Schema describing the structure of a post
post_schema = {
    "properties": {
        "title" : {"type" : "string"},
        "body": {"type": "string"}
    },
    "required": ["title", "body"]
}
```

In your schema you say that you expect an object containing two properties: a string containing the title, and a string containing the body. You specify that both items are required for the JSON to be valid.

Next you can change your `post_post` endpoint to validate the input data against your schema:

```
@app.route("/api/posts", methods=["POST"])
@decorators.accept("application/json")
@decorators.require("application/json")
def posts_post():
    """ Add a new post """
    data = request.json

    # Check that the JSON supplied is valid
    # If not you return a 422 Unprocessable Entity
    try:
        validate(data, post_schema)
    except ValidationError as error:
        data = {"message": error.message}
        return Response(json.dumps(data), 422, mimetype="application/j

    # Add the post to the database
    post = models.Post(title=data["title"], body=data["body"])
    session.add(post)
    session.commit()

    # Return a 201 Created, containing the post as JSON and with the
    # Location header set to the location of the post
    data = json.dumps(post.as_dictionary())
    headers = {"Location": url_for("post_get", id=post.id)}
    return Response(data, 201, headers=headers,
                    mimetype="application/json")
```

You're trying to validate the data against your post schema. If this throws a `ValidationError` then you create an error message from the exception and return it with a `422` status. Otherwise carry on as usual and add the post to the database.

Try to run your tests again. You should see that you now give descriptive error messages if the wrong types of data are sent to your API.

With that in place you now have a complete, well-behaved API allowing you to add and retrieve posts. Next you will create some real single-page applications by building APIs.

Try It!

Try adding a new endpoint which receives a PUT request at `/api/post/<id>`. This endpoint should accept JSON in the same format as our POST method, and use it to edit the title and body of the specified post.

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

✓ Mark as completed

Previous

Next