Tuneful project

Create a Song Management API

The database

In the file *tuneful/models.py* we are going to need two SQLAlchemy models.

1. The Song model: This should have an integer id column, and a column specifying a one-to-one relationship with a File.

2. The File model: This should have an integer id column, a string column for the filename, and the backref from the one-to-one relationship with the Song.

Each model should have an `as_dictionary` method.
The `Song.as_dictionary` method should produce a dictionary which looks something like this:

```
{

    "id": 1,
    "file": {
        "id": 7,
        "name": "Shady_Grove.mp3"
    }
}
```

The `File.as_dictionary` method should just return the `file` element of the song dictionary (i.e. `{"id": 7, "name": "Shady_Grove.mp3"}`).
Try to create those two models, and use Python's interactive interpreter to experiment with adding Song and Files and creating relationships between them.

The GET endpoint

In the *tuneful/api.py* file we need a GET endpoint for `/api/songs` which returns a list of all of the songs as JSON. Try to follow the practice laid out in the previous lesson as you create this, adding tests as you go and trying to make your endpoint well-behaved for clients.

The POST endpoint

In the *tuneful/api.py* file add a POST endpoint for `/api/songs` which allows you to add a new song to the database. The endpoint should look for JSON POST data in the following format:

```
{

    "file": {
    "id": 7
  }
}
```

You should use the file ID to make sure that the file exists in the database, and make the relationship with your new song. Again you should try to follow the good practice for working with POST data, and write tests to make sure that your endpoint is working as you expect.

Testing it out

Now that you have the beginning of an API try adding a couple of example `File` and `Song` object to the database from the interactive Python interpreter and then visit the front-end. You should be able to see the songs listed in the left column of the site.
While you are in the front-end, try adding a song using the `Add Song` button. You should see a failed request to the `/api/files` endpoint in the console as it tries to upload the file. In the next assignment you'll be looking at how file upload works on a

single-page web app and writing the endpoints to get this up and running on your site.

Extension task

Add PUT and DELETE endpoints to your API allowing you to edit and delete songs from the app.

Uploading Files Using an API

You've already seen how simple it is to send data to an API using JSON. So surely it should be equally simple to send a file in to your application? Sadly a programmer's life is never that easy; JSON does not have a built-in way to encode binary data such as files. This leaves you with two options, each with advantages and disadvantages:

1. You can encode the file using Base-64
    1. Only one HTTP request required
    2. Encoding and decoding the data takes time
    3. The data you transfer will be larger
2. You can send the file as multi-part form data
    0. Requires two HTTP requests – one for the file and another for the related information
    1. No encoding and decoding required
    2. Less data needs transferring

In this assignment you are going to choose the second option, sending the data as multi-part form data. This is the default method used by `<form>`elements for sending data to a server.

Accessing files

First you are going to add a route which will allow you to access the files which you upload. Create a test where you add a file to an upload folder then try to access it through an HTTP request.

```
def test_get_uploaded_file(self):
    path =  upload_path("test.txt")
    with open(path, "wb") as f:
        f.write(b"File contents")

    response = self.client.get("/uploads/test.txt")

    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.mimetype, "text/plain")
    self.assertEqual(response.data, b"File contents")
```

Here you're using the `upload_path` function, which is defined in the *tuneful/utils.py* file, to get the location where the file should live. Create the file, then send a GET request to `/uploads/<filename>`. You check that the response contains the correct file with the right mimetype.
Now add your route to try to make the test pass.

Test first

Next you'll put together a test which tries to upload a simple text file to the server. In the `TestAPI` class in the *tests/api_tests.py* file try adding the following test:

```
def test_file_upload(self):
    data = {
        "file": (BytesIO(b"File contents"), "test.txt")
    }

    response = self.client.post("/api/files",
        data=data,
        content_type="multipart/form-data",
        headers=[("Accept", "application/json")]
    )

    self.assertEqual(response.status_code, 201)
    self.assertEqual(response.mimetype, "application/json")

    data = json.loads(response.data.decode("ascii"))
    self.assertEqual(urlparse(data["path"]).path, "/uploads/test.txt")

    path = upload_path("test.txt")
    self.assertTrue(os.path.isfile(path))
    with open(path, "rb") as f:
        contents = f.read()
    self.assertEqual(contents, b"File contents")
```

There are a few things to notice here. You're constructing the form data as a dictionary, using an instance of the Python `BytesIO` class to simulate a file object (you can import this class from the `io` module). You then send this dictionary to the `/api/files` endpoint with a content type of `multipart/form-data`. Finally you carry out a series of checks, making sure that the response data points you to a URL where you can access the file, and that the file has been saved correctly in an upload folder specified in the *tuneful/config.py* file.
Try running the test. You should see that the request fails with a `404` error.

Writing the endpoint

Next you can write the endpoint to handle the uploads. In the *tuneful/api.py* file try adding the following endpoint:

```
@app.route("/api/files", methods=["POST"])
@decorators.require("multipart/form-data")
@decorators.accept("application/json")
def file_post():
    # Put your code here
```

You're trying to access the uploaded file from Flask's `request.files`dictionary. If you don't find the file you return an error in the usual way.

Next you use the Werkzeug `secure_filename` function to create a safe version of the filename which the client supplies. This prevents malicious users from creating files anywhere on the server by passing in specially crafted filenames. So for example the path *../../../etc/passwd* (which could point to the file containing the authorized users for the server) is replaced by *etc_passwd*.
Use your secure filename to create the `File` object and add it to the database. (Be sure to check this code against your actual model, to make sure you have the same column names.) Then you can use the `upload_path`function which is defined in the *tuneful/utils.py* file to save the file to an upload folder.
Lastly you use the `File.as_dictionary` method to return the file information. At the moment this does not return the path of the uploaded file. Update this method to return the location where you can access the file.

Try running the test again. You should now have a working method of uploading files to the application.

Try firing up the server using `python run.py` and visit the site, then upload a song using the Add Songs button. When the song has uploaded you should be able to select it and listen to it using the player at the bottom of the interface.