Monday: Movie APIs; Macros

Movie API

We are going to get our movie data through **The Movie Database (https://www.themoviedb.org/)** API. Head on to https://www.themoviedb.org/ and sign up for an account.

Creating an API key

1. First go to your accounts settings page
2. Then click on the API menu on the left
3. Click on create an API Key to generate a new API key

Storing API keys

We will create another configuration file that will store our API key. Inside our root folder create a new directory *instance* and inside it create a new *config.py* file. This is where we will store secret objects that we do not want to display to anyone.

instance/config.py

```
MOVIE_API_KEY = '<Your Api Key>'
```

We then include this folder inside our `.gitignore` file

.gitignore

```
virtual/
*.pyc
instance/
```

Now we have to connect our application with this new configuration file.

`__init__.py`

```
from flask import Flask
from .config import DevConfig

# Initializing application
app = Flask(__name__,instance_relative_config = True)

# Setting up configuration
app.config.from_object(DevConfig)
app.config.from_pyfile('config.py')

from app import views
```

We update our `__init__.py` file.

We pass in `instance_relative_config` which allow us to connect to the `instance/` folder when the app instance is created.

The `app.config.from_pyfile('config.py')` connects to the `config.py` file and all its contents are appended to the `app.config`.

We then store the movie base URL inside the `app/config` file.

```
class Config:
    '''
    General configuration parent class
    '''
    MOVIE_API_BASE_URL ='https://api.themoviedb.org/3/movie/{}?api_key={}'
```

We use the `{}` to represent sections in the URL that will be replaced with actual values

API Requests

We will then create *request.py* file inside our app folder. This is where we will write code to make requests to our API.

request.py

```
from app import app

# Getting api key
api_key = app.config['MOVIE_API_KEY']
```

We import the app instance and from it we get the API key from the config object.

You can find the application at this point here

https://github.com/mbuthiya/watchlist/tree/03-movie-list-api

Movie Class

We now need to create a Movie class that will allow us to create movie instances from the response from the API.

Create a new folder inside the app and name it *models* inside it create a new file *movie.py*. We also create an empty `__init__.py` inside the folder.

This will make the folder a python package that can be imported to other packages outside it

We then create test case file for the class inside our app folder we create a *movie_test.py* file.

movie_test.py

```
import unittest
from models import movie
Movie = movie.Movie

class MovieTest(unittest.TestCase):
    '''
    Test Class to test the behaviour of the Movie class
    '''
```

```
    def setUp(self):
        '''
        Set up method that will run before every Test
        '''
        self.new_movie = Movie(1234,'Python Must Be Crazy','A thrilling new Python Series','h
ttps://image.tmdb.org/t/p/w500/khsjha27hbs',8.5,129993)

    def test_instance(self):
        self.assertTrue(isinstance(self.new_movie,Movie))


if __name__ == '__main__':
    unittest.main()
```

We import the `Unittest` module and the movie module. We then get the `Movie` class which we will create.

We then create a test class and inside we define a `setUp()` method. This will instantiate our Movie class to make the `self.new_movie` object. We pass in six arguments.

We then define a test case `test_instance` that uses the `isinstance()` function that checks if the object `self.new_movie` is an instance of the `Movie` class.

Make sure the test is failing. We then can add code to make the code work

models/movie.py

```
class Movie:
    '''
    Movie class to define Movie Objects
    '''

    def __init__(self,id,title,overview,poster,vote_average,vote_count):
        self.id =id
        self.title = title
        self.overview = overview
        self.poster = 'https://image.tmdb.org/t/p/w500/'+ poster
        self.vote_average = vote_average
        self.vote_count = vote_count
```

Here we define a `Movie` class and then we create an `__init__` method and we pass in the six parameters we want inside our movie objects.

```
1. Title - The name of the movie
2. Overview - A short description on the movie
3. image- The poster image for the movie
4. vote_average - Average rating of the movie
5. vote_count - How many people have rated the movie
6. id - The movie id
```

Task: For practice write a test that confirms that the object is instantiated correctly.

You can find the application at this point here https://github.com/mbuthiya/watchlist/tree/04-create-movie-class

Creating First API call

We now need to get data from our API. We insert the following code in our *request.py* file

request.py

```
from app import app
import urllib.request,json
from .models import movie

Movie = movie.Movie
```

First, we handle the imports. We import the flask application instance. We then import the Python `urllib.request` module that will help us create a connection to our API URL and send a request and `json` modules that will format the JSON response to a Python dictionary.

request.py

```
....
# Getting api key
api_key = app.config['MOVIE_API_KEY']

# Getting the movie base url
base_url = app.config["MOVIE_API_BASE_URL"]
```

We then access our app configuration objects. We access the configuration objects by calling `app.config['name_of_object']`. We get the API key and the movie URL.

request.py

```
...

def get_movies(category):
    '''
    Function that gets the json response to our url request
    '''
    get_movies_url = base_url.format(category,api_key)

    with urllib.request.urlopen(get_movies_url) as url:
        get_movies_data = url.read()
        get_movies_response = json.loads(get_movies_data)

        movie_results = None

        if get_movies_response['results']:
            movie_results_list = get_movies_response['results']
            movie_results = process_results(movie_results_list)


    return movie_results
```

We then create a `get_movies()` function that takes in a movie category as an argument.

We use the `.format()` method on the `base_url` and pass in the movie category and the api_key. This will replace the `{}` curly brace placeholders in the base_url with the category and api_key respectively.

This creates `get_movies_url` as the final URL for our API request.

We then use `with` as our context manager to send a request using the`urllib.request.urlopen()` function that takes in the `get_movies_url` as an argument and sends a request as `url`

We use the `read()` function to read the response and store it in a `get_movies_data`variable.

We then convert the JSON response to a Python dictionary using json.loads function and pass in the `get_movies_data` variable.

We then check if the response contains any data. For us to better understand what is happening, we need first to see what the data looks like. We achieve this by running the base URL(https://api.themoviedb.org/3/movie/popular?api_key=<your_API_KEY>) in our browser to get json response as shown in the image below:

As you can see from the json response above, we have a property *result* which is a list that contains the movie objects. This property is what we use to check if the response contains any data.

If it does we call a `process_results()` function that takes in the list of dictionary objects and returns a list of movie objects .

We then return `movie_results` which is a list of movie objects.

Processing Results

We need to create a function that will process the results and create movie objects from the elements that we need.

request.py

```
...
def process_results(movie_list):
    '''
    Function  that processes the movie result and transform them to a list of Objects

    Args:
        movie_list: A list of dictionaries that contain movie details

    Returns :
        movie_results: A list of movie objects
    '''
    movie_results = []
    for movie_item in movie_list:
        id = movie_item.get('id')
        title = movie_item.get('original_title')
        overview = movie_item.get('overview')
        poster = movie_item.get('poster_path')
        vote_average = movie_item.get('vote_average')
        vote_count = movie_item.get('vote_count')

        if poster:
            movie_object = Movie(id,title,overview,poster,vote_average,vote_count)
            movie_results.append(movie_object)

    return movie_results
```

We create a function `process_results()` that takes in a list of dictionaries. We create an empty list `movie_results` this is where we will store our newly created movie objects.

We then loop through the list of dictionaries using the `get()` method and pass in the keys so that we can access the values.

Some `movie_item` 's might not have a poster. This will give an error when we are trying to create an object. So we check if the `movie_item` has a poster then we create the movie object. We use the values we get to create a new movie object then we append it to our empty list

We then return the list with movie objects.

To get a better sense on how to use The Movie DB api visit the documentation page https://developers.themoviedb.org/3/getting-started/

Making the API call

We can now make the API call to get a particular category of movies

views.py

```
...
from .request import get_movies

@app.route('/')
def index():

    '''
    View root page function that returns the index page and its data
    '''

    # Getting popular movie
    popular_movies = get_movies('popular')
    print(popular_movies)
    title = 'Home - Welcome to The best Movie Review Website Online'
    return render_template('index.html', title = title,popular = popular_movies)
```

We first import the `get_movies()` function from the request module. We want to get the popular movie from our API.

We create a variable `popular_movies` where we call our `get_movies()` function and pass in `"popular"` as an argument. we then pass the result from that function call to our template

For loops

Now we need to display our popular movies to our template.

index.html

```
....
<body>
        <!-- Popular movie section  -->
        <h3> Popular Movies</h3>
         <ul>
            {% for popular_movie in popular %}
                <li>{{ popular_movie.title}}</li>
            {% endfor %}
        </ul>
 </body>
 ....
```

We create a `for` loop inside a control block and loop through each movie. We create a list with the movie title.

You can now run the application to confirm that the API is working and you should see a list of movies displayed.

Adding More Categories

views.py

```
@app.route('/')
def index():

    '''
    View root page function that returns the index page and its data
    '''

    # Getting popular movie
    popular_movies = get_movies('popular')
    upcoming_movie = get_movies('upcoming')
    now_showing_movie = get_movies('now_playing')
    title = 'Home - Welcome to The best Movie Review Website Online'
    return render_template('index.html', title = title, popular = popular_movies, upcoming =
upcoming_movie, now_showing = now_showing_movie )
```

We use our `get_movies()` function to get two more movie categories *upcoming movies* and *now showing movies* and then we pass them into our template.

index.html

```
....
<body>
        <!-- Popular movie -->
        <h3> Popular Movies</h3>
         <ul>
            {% for popular_movie in popular %}
                <li>{{ popular_movie.title}}</li>
            {% endfor %}
        </ul>

        <!-- Upcoming movie  -->
        <h3> Upcoming Movies </h3>
         <ul>
            {% for upcoming_movie in upcoming %}
                <li>{{ upcoming_movie.title}}</li>
            {% endfor %}
        </ul>
        <!-- Now showing movie -->
        <h3>Now Showing</h3>
         <ul>
            {% for now_showing_movie in now_showing %}
                <li>{{ now_showing_movie.title}}</li>
            {% endfor %}
        </ul>
</body>
```

Notice as we keep on adding more categories we start repeating our `for` loop control blocks. We can use **macros** to make our code DRY.

Macros

Macros are the Jinja equivalent to python functions. It allows us to group blocks of our code's logic that will be reused multiple times.

Create a new template file *macros.html* this is where we will store all our macros.

macros.html

```
{% macro displayMovieList(movie_list) %}

    {% for movie in movie_list %}
        <li>{{ movie.title}}</li>
    {% endfor %}

{% endmacro %}
```

We define our first macro `displayMovieList()` that takes one argument which is a list of movie objects. We then loop through the movie objects creating `<li>` tags to display movie titles.

We can now update our *index.html* file to use the macro.

index.html

```
{% import 'macros.html' as macro%}
```

At the top of our file we will import our macros html file so that we can use in our document.

index.html

```
{% import 'macros.html' as macro%}
...
    <body>
        <!-- Popular movie -->
        <h3> Popular Movies</h3>
        <ul>
            {{ macro.displayMovieList(popular)}}
        </ul>
        <!-- Upcoming movie  -->
        <h3> Upcoming Movies </h3>
        <ul>
            {{ macro.displayMovieList(upcoming)}}
        </ul>
        <!-- Now showing movie -->
        <h3>Now Showing</h3>
        <ul>
            {{ macro.displayMovieList(now_showing)}}
        </ul>
    </body>
```

We then use the variable blocks to call our `displayMovieList` function and pass in the different movie category lists.

Now we can run our application code to see if the movie titles are being displayed.

You can find the application at this point here https://github.com/mbuthiya/watchlist/tree/06-macros