Thursday: Application Structures

Application structure

When we began our project we created a basic application structure. This was good for when our application was still small but as it grows in size and complexity we will need to efficiently organize our project.

We will create a new application structure that we will use from now on in the end it will look something like this

```
|-Watchlist
    |-app/
        |-main/
            |-__init__.py
            |-errors.py
            |-forms.py
            |-views.py
        |-static/
        |-templates/
        |-__init__.py
        |-models.py
        |-requests.py
    |-tests/
        |-test_movie.py
        |-test_review.py
    |-virtual/
    |-config.py
    |-.gitignore
    |-manage.py
    |-start.sh
```

Please pay careful attention to the instructions to minimize chances of errors while following along.

Application configurations

We will move *config.py* from the `app` directory to the root directory.

config.py

```
 import os

class Config:

    MOVIE_API_BASE_URL ='https://api.themoviedb.org/3/movie/{}?api_key={}'
    MOVIE_API_KEY = os.environ.get('MOVIE_API_KEY')
    SECRET_KEY = os.environ.get('SECRET_KEY')


class ProdConfig(Config):
    pass


class DevConfig(Config):
    DEBUG = True

config_options = {
'development':DevConfig,
'production':ProdConfig
}
```

We update our *config.py* file. We first import the `os` module that will allow our application to interact with the operating system dependent functionality. We use the `os.environ.get()` function to get `MOVIE_API_KEY` and `SECRET_KEY` which we will set as environment variables.

We then create a dictionary `config_options` to help us access different configuration option classes.

### Application factory

Our application package is where most of the apps functionality is stored including static files and templates. How the application instance is created makes it difficult to apply configuration changes dynamically. This is useful when it comes to testing the application under different configurations. To solve this we create a function that can be launched from the script

app/__init__.py

```
from flask import Flask
from flask_bootstrap import Bootstrap
from config import config_options

bootstrap = Bootstrap()

def create_app(config_name):

    app = Flask(__name__)

    # Creating the app configurations
    app.config.from_object(config_options[config_name])

    # Initializing flask extensions
    bootstrap.init_app(app)

    # Will add the views and forms

    return app
```

We update our *app/__init__.py* to create our application factory function. We import import the `config_options` from the the *config.py* file that we updated. We then create the `bootstrap` instance.

We create a `create_app()` function that takes the configuration setting key as an argument. This is because we might want to create the application instance under different configurations. We then create the `Flask` app instance. We then import the configuration settings directly to the application using the `from_object()`method. We remove the `app.config.from_pyfile("config.py")` statement because all our secret configuration settings will be stored as environment variables.

We call the `init_app()` on an extension to complete on their initialization.Finally we return `app`.

Creating Blueprints

We now run into another problem. In our previous application structure the `app`instance was available in the global scope now it is created at runtime.Functionality like `@app.route()` cannot work until the `create_app()` is executed. This is where **Blueprints** come in handy.

A **Blueprint** is similar to an application in that it can also define routes.The main difference is that blueprints are dormant until they are registered by an application.

Inside the app instance create a new folder *main* this is where we will define our blueprint. We move our *errors.py forms.py* and *views.py* files from our app folder into our *app/main* subfolder. We will also create an *__init__.py* file inside our *app/main* subfolder to define the Blueprint.

app/main/__init__.py

```
from flask import Blueprint
main = Blueprint('main',__name__)
from . import views,errors
```

We import the `Blueprint` class from flask. We then initialize the `Blueprint` class by creating a variable `main`. The `Blueprint` class takes in 2 arguments. The name of the blueprint and the `__name__` variable to find the location of the blueprint.

To avoid circular dependencies we import the *views* and *errors* modules.

Registering a Blueprint

```
def create_app(config_name):
    app = Flask(__name__)

    # Creating the app configurations
    app.config.from_object(config_options[config_name])

    # Initializing flask extensions
    bootstrap.init_app(app)

    # Registering the blueprint
    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

We register a Blueprint inside our application factory function. We first import the instance we just created then we call the `register_blueprint()` method on the application instance and pass in the blueprint.

Blueprint Error handlers

The main difference with writing error handlers inside a blueprint is that it is invoked only inside the blueprint. To use application-wide error handler we must use the `app_errorhandler()` decorator.

app/main/errors.py

```
from flask import render_template
from . import main

@main.app_errorhandler(404)
def four_Ow_four(error):
    '''
    Function to render the 404 error page
    '''
    return render_template('fourOwfour.html'),404
```

Since we are defining our error handler inside our blueprint we import the blueprint instance `main` and use it to define our decorator.

Models

We create a *models.py* file inside our app folder that will contain our `Movie` and `Review` models.

app/models.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



class Review:

    all_reviews = []

    def __init__(self,movie_id,title,imageurl,review):
        self.movie_id = movie_id
        self.title = title
        self.imageurl = imageurl
        self.review = review


    def save_review(self):
        Review.all_reviews.append(self)


    @classmethod
    def clear_reviews(cls):
        Review.all_reviews.clear()

    @classmethod
    def get_reviews(cls,id):

        response = []

        for review in cls.all_reviews:
            if review.movie_id == id:
                response.append(review)

        return response
```

We create place both our `Review` and `Model` classes inside the _models.py_ file. We can now delete the models folder from our app directory.

Requests

app/requests.py

```
import urllib.request,json
from .models import Movie

#....
```

We first update our imports by importing the `Movie` class from the new _app/models.py_ file we have just created.

app/requests.py

```
import urllib.request,json
from .models import Movie

# Getting api key
api_key = None
# Getting the movie base url
base_url = None

def configure_request(app):
    global api_key,base_url
    api_key = app.config['MOVIE_API_KEY']
    base_url = app.config['MOVIE_API_BASE_URL']

#....
```

We then create two variables `api_key` and `base_url` and set them to `None`. We then create a function `configure_request()` that takes in the application instance and replaces the values of the None variables to application configuration objects.

Because we cannot access the application instance globally we need to call this function when we create our application instance. This will give us access to the application configuration objects.

app/__init__.py

```
def create_app(config_name):
    #....
    # Registering the blueprint
    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    # setting config
    from .requests import configure_request
    configure_request(app)

    return app
```

Inside our *app/__init__.py* file we import the `configure_request()` function from the *request.py* file. We call the function and pass the `app` instance.

Blueprint views

Now we can update our *main/views.py* file.

main/views.py

```
from flask import render_template,request,redirect,url_for
from . import main
from ..requests import get_movies,get_movie,search_movie
from .forms import ReviewForm
from ..models import Review
```

```
#...
```

We first update our files imports. We use the python relative import system in order to import a module according to where it is located in our project. We use one dot `.modulename` to import modules that are located within the same package, and two dots `..modulename` for modules located in a package higher up in the project hierarchy.

We then define our route decorators using the `main` blueprint instance instead of the app instance

main/views.py

```
# ...
# Views
@main.route('/')
def index():
    #....
```

Task: Define all the remaining routes with the appropriate decorators.

url_for() functions in Blueprints

Flask adds a **namespace** to all endpoints from a blueprint. The namespace is generally the name of the blueprint. For example the `index()` view function is registerd as `url_for('main.index')`. `url_for` also supports a shorthand format using `.`. `url_for('main.index')` can be shortened to `url_for('.index')`

Task: Update the `url_for` function in the `index` and `new_review` view functions with the proper namespace.

Unittests

We now move our `test_movie.py` and `test_review.py` files from our app folder into our tests folder.

tests/test_movie.py

```
import unittest
from app.models import 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','/
khsjha27hbs',8.5,129993)

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

We update our imports where we import our `Movie` class from the the *modules.py*file inside our app directory.

We then remove the `unittest.run()` command from the bottom of our file. We will consider how to run unittests later on. For now leave your file like this.

Task: Update the *test_review.py* file with the correct imports and remove the `unittest.run()` command from the bottom of the file.

Running our Application

Now that we have properly configured our application structure we can now launch our application. We will use a flask extension called Flask-Script that is a command line parser that will allow us to create startup configurations.

First install flask-script:

```
(virtual)$ pip install flask-script
```

Rename *run.py* file to *manage.py* and then setup flask-script.

manage.py

```
from app import create_app
from flask_script import Manager,Server

#...
```

We first import the `create_app` function from our app folder. We also import the `Manager` class from flask-script that will initialize our extension and the `Server` class that help us launch our server.

manage.py

```
# ...
# Creating app instance
app = create_app('development')

manager = Manager(app)
manager.add_command('server',Server)
```

We then call the `create_app` function and pass in the configuration_options key so as to create the application instance. We instantiate the `Manager` class by passing in the app instance. We then create a command line argument to tell us how to run our application. We use the `add_command` method to create a new command `'server'` which will launch our application server.

manage.py

```
# ...
manager.add_command('server',Server)
if __name__ == '__main__':
    manager.run()
```

We end the file by calling the `run` method on the Manager instance this will run the application.

Now we can run our application

```
(virtual)$ export MOVIE_API_KEY=<Your api key>
(virtual)$ export SECRET_KEY=<Your secret key>
(virtual)$ python3.6  manage.py server
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
```

Running your application this way everytime can get tedious we can create a start file that will setup our environment variables and run our application for us.

Inside our root directory create a new file *start.sh*

start.sh

```
 export MOVIE_API_KEY=<Your api key>
export SECRET_KEY=<Your secret key>

python3.6 manage.py server
```

We then can make that file executable by running `chmod a+x`.

```
 (virtual)$ chmod a+x start.sh
```

Now we can run our application very easily

```
 (virtual)$ ./start.sh
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
```

Running unittests

We can use the `manager` instance to create a command to run unittests in our application.

manage.py

```
# ...
manager.add_command('server',Server)
@manager.command
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)

if __name__ == '__main__':
    manager.run()
```

We use the `@manager.command` decorator to create a new command.We create the test function that loads all the test files and runs them all

```
python3.6 manager.py test
test_instance (test_movie.MovieTest) ... ok
test_check_instance_variables (test_review.TestReview) ... ok
test_get_review_by_id (test_review.TestReview) ... ok
```

```
test_instance (test_review.TestReview) ... ok
test_save_review (test_review.TestReview) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

We can now run unittests using the `python3.6 manager.py test` command from the terminal.