Wednesday: WTF Forms

WTF Forms

To handle forms in our application, we are going us Flask-WTF extension which is a flexible form rendering and validation library.

WTF forms make working with web forms much easier. It provides a lot of out of the box functionality Like security against **CSRF (https://en.wikipedia.org/wiki/Cross-site_request_forgery)** Cross site Request Forgery. And also offers validation.

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

To enable CSRF protection we need to create a Secret Key that will verify authenticity of requests with form data.

instance/config.py

```
MOVIE_API_KEY = '<Movie API Key>'
SECRET_KEY = '<Flask WTF Secret Key>'
```

Now we can start creating our web Form to create Reviews for the movies. Inside the app directory, create a forms.py file

forms.py

```
from flask_wtf import FlaskForm
from wtforms import StringField,TextAreaField,SubmitField
from wtforms.validators import Required

class ReviewForm(FlaskForm):

    title = StringField('Review title',validators=[Required()])
    review = TextAreaField('Movie review', validators=[Required()])
    submit = SubmitField('Submit')
```

We first Import `FlaskForm` class from the `flask_wtf` module this will help us create a Form class.

We then import `StringField`,`TextAreaField` and `SubmitField` field classes. These will help us create a text field, a text Area field and a submit button. **List of all the available fields (https://wtforms.readthedocs.io/en/latest/fields.html#basic-fields)** lastly, we import the `Required` class validator that will prevent the user from submitting the form without Inputting a value. **List of all the available validators (https://wtforms.readthedocs.io/en/latest/validators.html#built-in-validators)** We then create the `ReviewForm` class that inherits from the `FlaskForm`class. We Initialize the Field types by passing in two parameters. The first is the label and the second is a list of Validators where we initialize the `Required` validator.

Let us first create the template file that will display our form. Create a new file *new_review.html* inside the templates folder.

new_review.html

```
{% extends 'base.html'%}
{% import "bootstrap/wtf.html" as wtf %}
```

Flask-Bootstrap provides a high level function that allows us to render forms with bootstrap styling. We import _bootstrap/wtf.html_ from Flask-Bootstrap that defines helper functions to render web forms

new_review.html

```
{% extends 'base.html'%}
{% import "bootstrap/wtf.html" as wtf %}

{% block content %}
    <div class="container">

        <div class="col-md-8">

            <img src="{{movie.poster}}" alt="{{movie.title}}" class="img-responsive">
        </div>

        <div class="col-md-4">

            {{ wtf.quick_form(review_form) }}
        </div>

    </div>

{% endblock %}
```

The `wtf.quick_form()` function takes in a Flask-WTF form object and renders it with bootstrap styling.

Let us create the view function for that will handle this template.

views.py

```
from .models import review
from .forms import ReviewForm
Review = review.Review
```

We first import the `Review` class from our models folder. We also import the `ReviewForm` class from our forms file.

views.py

```
@app.route('/movie/review/new/<int:id>', methods = ['GET','POST'])
def new_review(id):
    form = ReviewForm()
    movie = get_movie(id)

    if form.validate_on_submit():
        title = form.title.data
        review = form.review.data
        new_review = Review(movie.id,title,movie.poster,review)
        new_review.save_review()
        return redirect(url_for('movie',id = movie.id ))
```

```
    title = f'{movie.title} review'
    return render_template('new_review.html',title = title, review_form=form, movie=movie)
```

We create a new dynamic route for our new_review function and pass in the movie id. We also add the `methods` argument to our decorator which tells flask to register the view function as a handler for both `GET` and `POST` requests. When `methods` argument is not given the view function is registered to handle `GET` requests only.

We then create an instance of the `ReviewForm` class and name it `form` . We also call the `get_movie` and pass in the ID to get the movie object for the movie with that ID.

The `validate_on_submit()` method returns `True` when the form is submitted and all the data has been verified by the validators. If `True` we gather the data from the form input fields and create a new review object and save it. We then redirect the response to the `movie` view function and pass in the dynamic movie ID.

If the `validate_on_submit()` method returns `False` we will render our *new_review.html* template file and pass in the `title`, the `form` object and the `movie`object.

movie.html

```
<div class="col-xs-6 col-sm-6 col-md-6 col-lg-6 movie-details">
            <h3>{{ movie.title }}</h3>

            <p class="overview"> {{ movie.overview }}</p>

            <p class="ratings"> <b> {{ movie.vote_average }}</b> - <i>{{ movie.vote_count}} v
otes </i> </p>
            <a class="ratings" href="/movie/review/new/{{movie.id}}"> Write a new review</a>

</div>
```

We update our `movie-details` column in the *movie.html* file to add a link to a review page where we will display the form.

When we run the application and click on the link we should see our reviews form displayed for us and we can add new reviews to our movie.

Displaying Reviews

We need to create a method that will display all the reviews for a particular movie.

models/review.py

```
class Review:

    all_reviews = []
...... # Some code is here
    @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 a new class method `get_reviews` that takes in an ID. It loops through all the reviews in the `all_reviews` list and checks for reviews that have the same movie ID as the id passed. We then append those reviews to a new `response` list and return that response list.

views.py

```
@app.route('/movie/<int:id>')
def movie(id):

    '''
    View movie page function that returns the movie details page and its data
    '''
    movie = get_movie(id)
    title = f'{movie.title}'
    reviews = Review.get_reviews(movie.id)

    return render_template('movie.html',title = title,movie = movie,reviews = reviews)
```

We update our `movie` view function and call the `get_reviews` class method that takes in a movie ID and will return a list of reviews for that movie. We then pass in the reviews list to our template

macros.html

```
<!-- Displaying reviews macro -->
{% macro displayReviews(review_list) %}
    {% for review in review_list %}
    <div class="col-xs-12 col-sm-4 col-md-4 col-lg-4 review-card">
        <h2> {{ review.title}} </h2>
        <p> {{ review.review | truncate(30) }}</p>
    </div>
    {% endfor %}
{% endmacro %}
```

In our *macros.html* template we create a new macro `displayReviews` that takes in a list of reviews and loops through them to display each review by title and short snippet of the review.

movie.html

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

We first import *macros.html* template file so that we can use our newly created macro.

movie.html

```
...
    <div class="row">

        <h1>Reviews</h1>
        <hr>

        {% if reviews %}
            {{macro.displayReviews(reviews)}}

        {% else%}
            <h3 class="text-center">This movie has no reviews </h3>

            <a class="btn btn-success btn-lg text-center" href="/movie/review/new/{{movie.i
```

```
d}}"> Write a new review</a>
        {% endif %}

    </div>
...
```

We then create a new row where first we check if any review exists for that particular movie and if True we call our macro and pass in the reviews list. If not we put in a helpful statement and a link to write a new review.

Now when we run our application and create a new review it will be displayed on the movie page.

You can find the application at this point from here https://github.com/mbuthiya/watchlist/tree/14-wtf-forms