Web Studio: Week 3 Turn-in #2

Name: Ivy Chen
UNI: ic2389
Due: Friday 9/21 at 2PM
COMS6998 Adv. Web Studio

Original Goals

High Level Goal

1. User can look up an actor and see all the other movies an actor was in.

Low Level Goals

7-10 low level goals that will help you accomplish your high level goal

1. Create database models for movies and actors
2. Create a table for the relationship between movies and actors
3. Set up a SQLite database + populate with at least 5 movies and 2 actors per movie
4. Set up app configuration
5. Learn to make a SQLalchemy database query
6. Modify add movie to add movie to database
7. Perform at least one join operation
8. Creating actor profiles with actor information

Lessons Learned Through Iteration

1. My plan was to create models for all the tables I wanted in my database eg. movies, cast members, and "stars in" relationship and my initial implementation consisted of three models, but then I read the Flask documentation and it provided a different suggestion for how to model many-to-many relationships (http://flask-sqlalchemy.pocoo.org/2.3/models/). Thus, I had to change my `models.py` in order to fit the "right" way to represent the relationship between actors and cast members.

   In addition, I had to change some of my SQL queries to work with the new implementation. Since I've never used SQLalchemy and wasn't familiar with SQLite databases (at first, I was just trying to figure out how to even set one up 😅 and turns out I just needed to create a `.db` file).

```python
from app import db

stars = db.Table(
                'stars',
                db.Column('movieId', db.Integer, db.ForeignKey('movie.id'),
                primary_key=True),
                db.Column('talentId', db.Integer,
                db.ForeignKey('talent.tid'), primary_key=True),
                db.Column('role', db.String(140), nullable=False)
                )

class Movie(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(140), nullable=False)
    year = db.Column(db.Integer, nullable=False)
    genre = db.Column(db.String(140), nullable=False)
    runtime = db.Column(db.Integer)
    overview = db.Column(db.Text)
    talent = db.relationship('Talent', secondary=stars,
    backref=db.backref('movie', lazy=True), lazy='subquery')

    def __repr__(self):
        return '<Movie {}, {}, {}>'.format(self.title, self.year,
        self.overview)

class Talent(db.Model):
    tid = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(140), nullable=False)
    # movie_id = db.Column(db.Integer, db.ForeignKey('movies.id'),
    nullable=False)

    def __repr__(self):
        return '<Talent {}>'.format(self.name)
```

2. Conceptually, it seemed like updating movies and their actors weren't super difficult, but I ran into a lot of edge cases when I set out to implement adding and updating movie and movie cast information:
    a. Not wanting to duplicate/create the same actors when movie information is updated.
    b. Modifying relationships so when a movie's cast list is updated, only the actors in the updated list are part of the movie-actor relationship.
    c. What if an actor is in a movie, but plays two different roles?
    d. Dealing with duplicate movies

    I ended up solving the problem by making some simplifying assumptions:
    - Actor names must be unique
    - An actor can only play one role in a movie

- My implementation is inefficient, but it accomplishes what I wanted: only the actors in the latest list are included in the movie-actor relationship.
- In the forms to add and update movies, the formatting of cast and roles must be in the following form: `name,role`
- Duplicate movies are okay for now

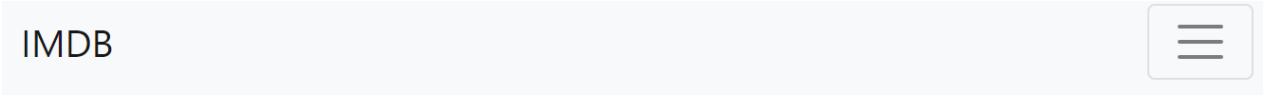
```python
@app.route('/addMovie', methods=['POST', 'GET'])
def addMovie():
    if request.method == 'POST':
        title = request.form['title']
        year = int(request.form['year'])
        genre = request.form['genre']
        runtime = int(request.form['runtime'])
        overview = request.form['overview']
        castList = list(request.form['cast'].split('\n'))
        cast = [tuple(c.split(',')) for c in castList]

        # Add movie to db
        m = models.Movie(title=title, year=year, genre=genre, runtime=runtime, overview=overview)
        db.session.add(m)
        db.session.commit()
        db.session.refresh(m)

        # Add any new cast members with their roles only if new name
        for name,role in cast:
            # Check if the actor already exists
            exists = models.Talent.query.filter_by(name=name).first()

            if not exists:
                t = models.Talent(name=name)
                db.session.add(t)
                db.session.commit()
                db.session.refresh(t)
                statement = models.stars.insert().values(movieId=m.id, talentId=t.tid, role=role)
            else:
                statement = models.stars.insert().values(movieId=m.id, talentId=exists.tid,
                role=role)

            db.session.execute(statement)
            db.session.commit()

        return redirect('/movie/' + str(m.id))

    return render_template('addMovie.html')
```

Adding movie and new actors (only if uniquely new name, otherwise, create a relationship with existing actor with the name).

```python
@app.route('/saveMovie', methods=['POST', 'GET'])
def save():
    if request.method == 'POST':
        # data = request.args
        # data = request.get_json(force=True)
        id = request.form['id']

        # Remove all stars in relationships that aren't in the updated cast list
        delCast = models.stars.delete(models.stars.c.movieId == id)
        db.session.execute(delCast)
        db.session.commit()

        m = db.session.query(models.Movie).filter_by(id=id).first()
        m.title = request.form['title']
        m.year = int(request.form['year'])
        m.genre = request.form['genre']
        m.runtime = int(request.form['runtime'])
        m.overview = request.form['overview']
        db.session.commit()

        castList = list(request.form['cast'].split('\n'))
        cast = [tuple(c.split(',')) for c in castList]

        for name,role in cast:
            # Check if the actor already exists
            exists = models.Talent.query.filter_by(name=name).first()
            if not exists:
                t = models.Talent(name=name)
                db.session.add(t)
                db.session.commit()
                db.session.refresh(t)
                statement = models.stars.insert().values(movieId=id, talentId=t.tid, role=role)
            else:
                statement = models.stars.insert().values(movieId=id, talentId=exists.tid,
                role=role)

            db.session.execute(statement)
            db.session.commit()

        return redirect('/movie/' + str(id))

    return render_template('/')
```

Updating movie information.

3. My plan was to create actor profiles similar to how IMDB has actor pages with a brief biography, birthdate films & movies an actor has starred in, and some other fun facts about an actor. I ran into the problem of figuring out what information is valuable to add (eg. if I added an actor's age, then I'd need to update it on their birthday each year). I solved the problem by reducing the scope for the actor pages subgoal this week. For now, I sought to focus on querying the database for actor/role information and only include an actor's name and the movies they starred in/role they played.



Keira Knightley

Filmography:

- Gretta in Begin Again
- Joan clark in The Imitation Game

Basic actor page include name and filmography

Goal Progress

Status Summary

✓ = DONE
SOON = PARTIAL/DEFERRED
✗ = NOT DONE

High Level Goal	**Status**
User can look up an actor and see all the other movies an actor was in.	✓
Low Level Goals	**Status**
Create database models for movies and actors	✓
Set up a SQLite database + populate with at least 5 movies and 2 actors per movie	✓
Learn to make a SQLalchemy database query	✓
Modify add movie to add movie to database	✓
Allow for movie information to be updated	✓
Perform at least one join operation	✓
Creating actor profiles with actor information	SOON

High Level Goal

1. User can look up an actor and see all the other movies an actor was in.
   Accomplished! Met the following requirements:
   - Store data in SQLite database
   - Query with SQLAlchemy
   - Enables CRUD for movies
   - Uses at least one “join” operation

Low Level Goals

1. Create database models for movies and actors — DONE
   I followed Flask documentation for creating a many-to-many relationship here: http://flask-sqlalchemy.pocoo.org/2.3/models/

```python
from app import db

stars = db.Table(
                'stars',
                db.Column('movieId', db.Integer, db.ForeignKey('movie.id'),
                primary_key=True),
                db.Column('talentId', db.Integer,
                db.ForeignKey('talent.tid'), primary_key=True),
                db.Column('role', db.String(140), nullable=False)
                )

class Movie(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(140), nullable=False)
    year = db.Column(db.Integer, nullable=False)
    genre = db.Column(db.String(140), nullable=False)
    runtime = db.Column(db.Integer)
    overview = db.Column(db.Text)
    talent = db.relationship('Talent', secondary=stars,
    backref=db.backref('movie', lazy=True), lazy='subquery')

    def __repr__(self):
        return '<Movie {}, {}, {}>'.format(self.title, self.year,
        self.overview)

class Talent(db.Model):
    tid = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(140), nullable=False)
    # movie_id = db.Column(db.Integer, db.ForeignKey('movies.id'),
    nullable=False)

    def __repr__(self):
        return '<Talent {}>'.format(self.name)
```

DB models for movie, talent and table for "stars in" relationship

2. Create a table for the relationship between movies and actors — DONE
   By following the Flask documentation, I created the table `stars` to represent that a `movie` stars various actors (`talent`).

3. Set up a SQLite database — DONE
   I initially couldn't figure out how to setup a SQLite database, ie. if I needed to install something, spin up a database instance, etc. I found this tutorial (https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-iv-database/page/2) which was super helpful in helping me setup my SQLAlchemy configuration and walking through the basic concepts of models and querying my database using SQLAlchemy.

```python
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    SQLALCHEMY_DATABASE_URI = os.environ.get('SQLALCHEMY_DATABASE_URI') or 'sqlite:///' +
    os.path.join(basedir, 'imdb.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SQLALCHEMY_ECHO = True

```

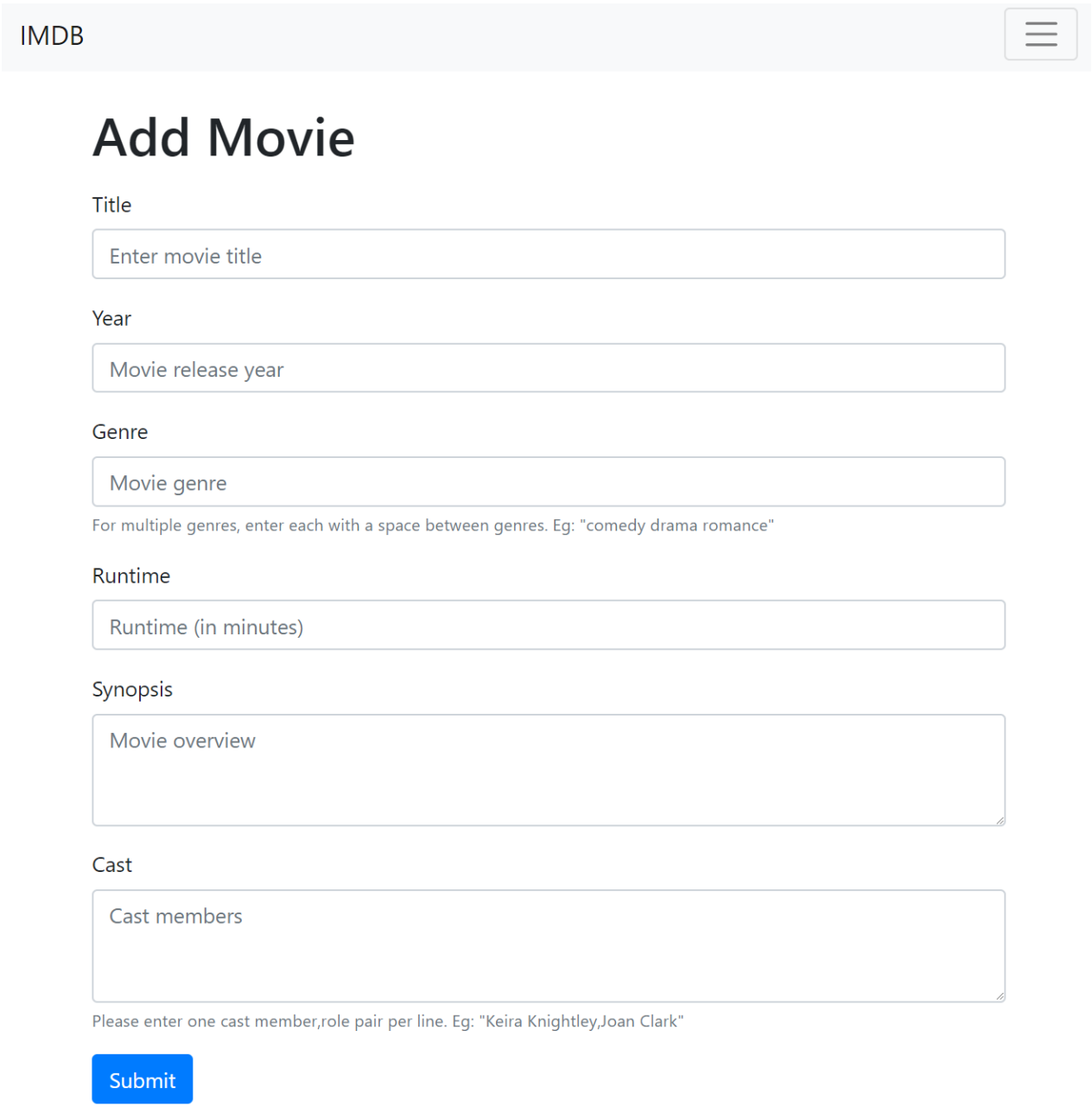
Configuration for SQLAlchemy and database

4. Learn to make a SQLalchemy database query — DONE
   Tutorials and Google searching were great to figure out how to accomplish things such as joining tables in a many-to-many relationship.

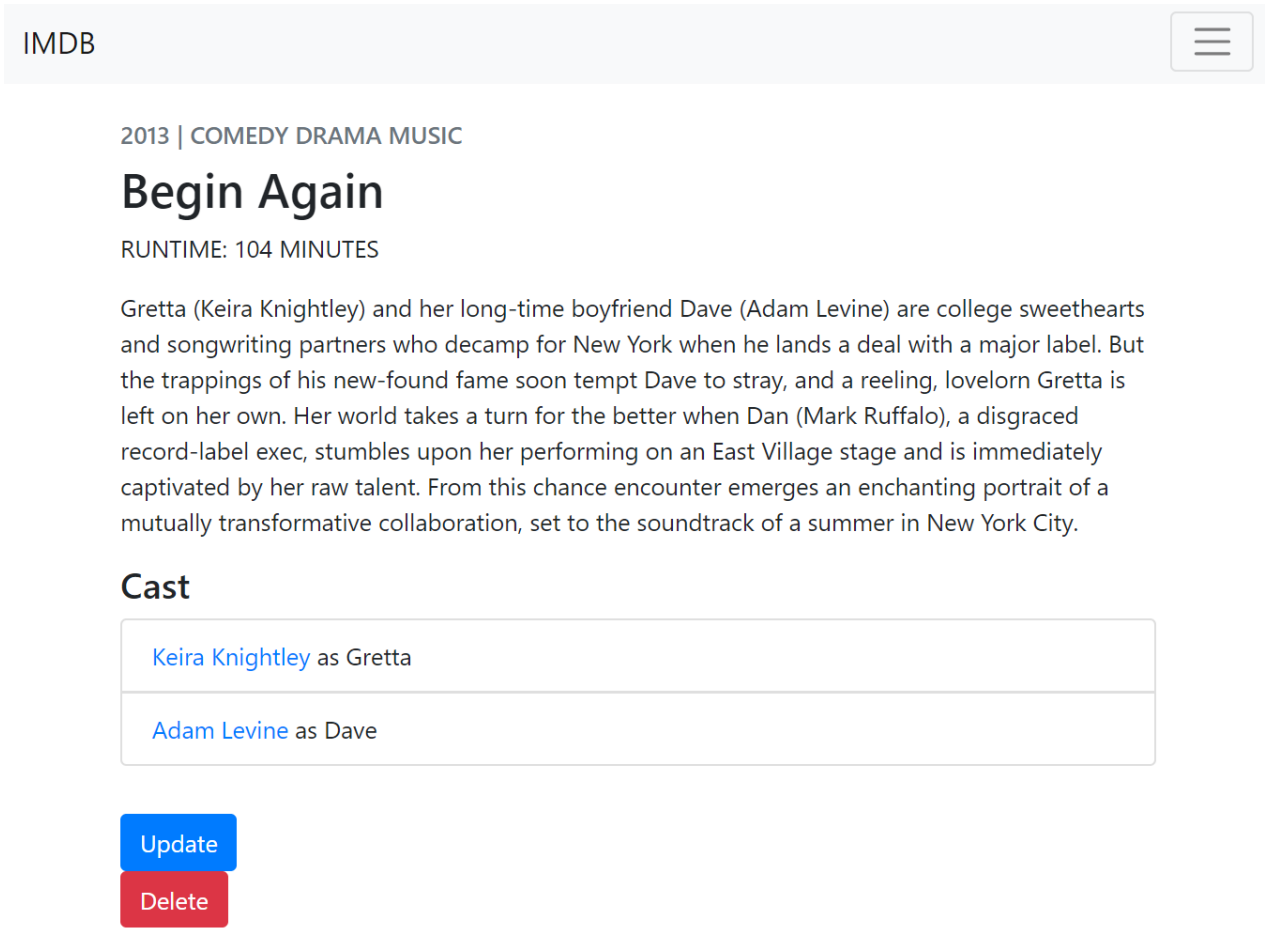
```python
@app.route('/')
def main():
    m = models.Movie.query.all()
    return render_template('main.html', movies=m)

@app.route('/movie/<id>', methods=['GET'])
def movie(id):
    movie_data = models.Movie.query.get(id)
    cast = db.session.query(models.Talent.name, models.stars)\
        .join(models.stars)\
        .filter_by(movieId=movie_data.id).all()
    db.session.commit()

    return render_template('movie.html', movie_data=movie_data, cast=cast)
```

Went through Flask-SQLAlchemy documentation and Google searching to figure out how to query the database and return relevant columns.

5. Modify add movie to add movie to database — DONE
   Since I updated the movie schema, I also needed to update the “Add Movie” form to reflect that as well as the backend database queries to allow new movies and actors to be added. I decided that new actors would be created if they had a unique name that wasn’t already in the database. The backend also handles creating the relationships between actors and movies.

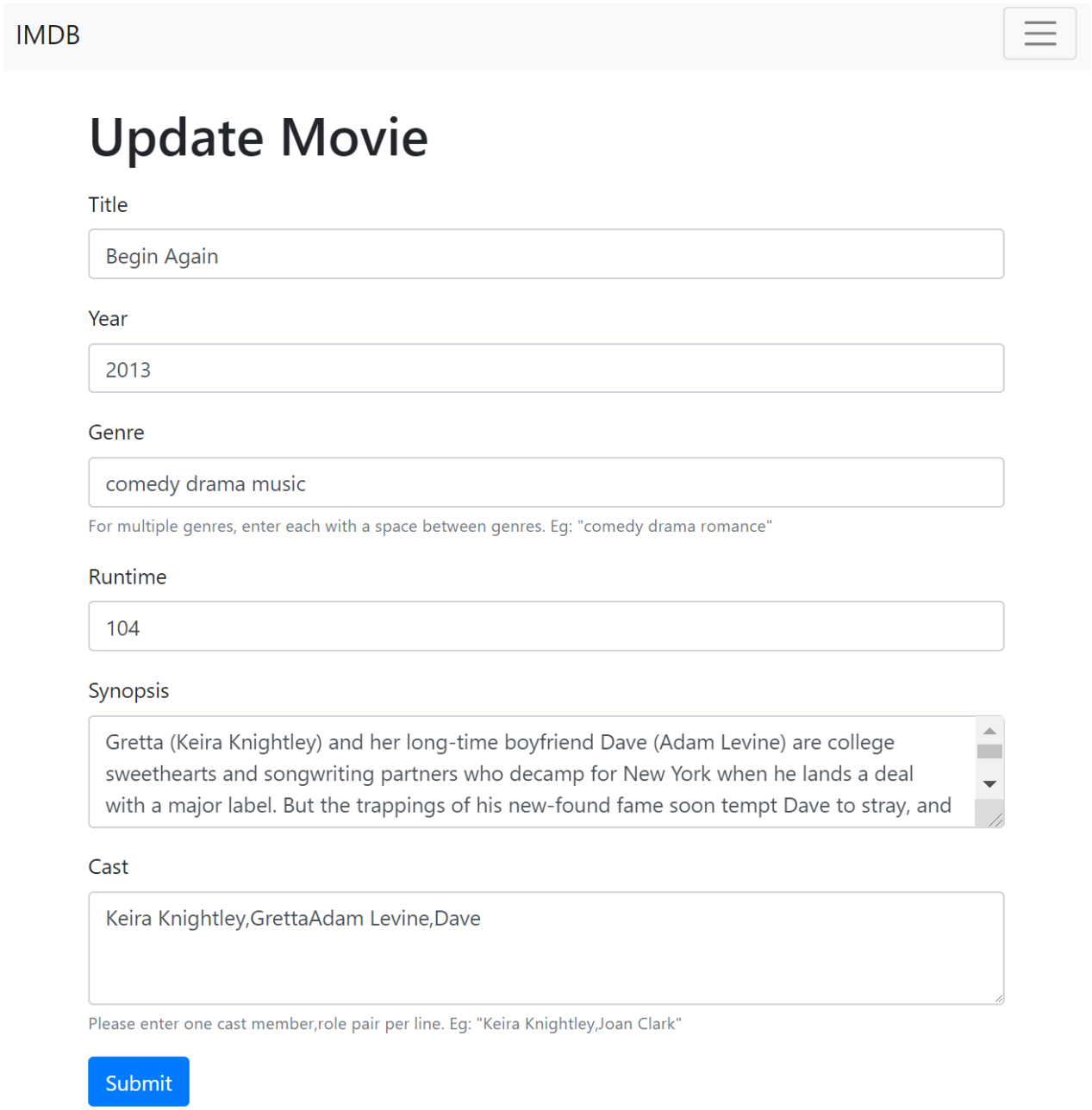


6. Allow for movie information to be updated — DONE
   I created a new form for updating movie information. The form uses the previous movie entry as a default in case the user doesn't enter new information in some fields. The form can be accessed from individual movie pages.



Movie page has a link to "Update" page

When a user submits the movie update form, they are redirected to the movie's page so they can see the updated information.



Movie update page has default data pre-populated

7. Perform at least one join operation — DONE
   I performed two join operations:
   a. On each movie page to find the actors in each movie
   b. On each actor page to find the movies the actor was in

```python
@app.route('/movie/<id>', methods=['GET'])
def movie(id):
    movie_data = models.Movie.query.get(id)
    cast = db.session.query(models.Talent.name, models.stars)\
        .join(models.stars)\
        .filter_by(movieId=movie_data.id).all()
    db.session.commit()

    return render_template('movie.html', movie_data=movie_data, cast=cast)

@app.route('/cast/<id>', methods=['GET'])
def cast(id):
    cast_data = models.Talent.query.get(id)
    m = db.session.query(models.Movie.title, models.stars)\
        .join(models.stars)\
        .filter_by(talentId=cast_data.tid).all()

    return render_template('cast.html', cast=cast_data, movies=m)
```

Join operations for movie and cast pages