Netflix Library

DSCI 551 Final Project - Team #79

Team Details

Student #1
Name: Pooridon Rattanapairote
Email: prattana@usc.edu
USC ID: 1469709999

Student #2
Name: Pannawat Chauychoo
Email: pchauych@usc.edu
USC ID: 7282127237

Github: DSCI-551-project-79 Repository

Table of Content

Planned Implementation

Project Scope

Project:

- Develop a scalable movie database with user interaction capabilities

Objectives:

- Design and implement a MongoDB database to store movie, user, and user interaction
- Utilize MongoDB sharding on EC2 Ubuntu instances for horizontal scalability and fault tolerance.
- Build two web applications:
    - Admin application: Provides CRUD functionality for managing movie data.
    - User application: Enables users to search and explore movie information, with functionalities such as adding favorites. These interactions will be stored in the database for further analysis.

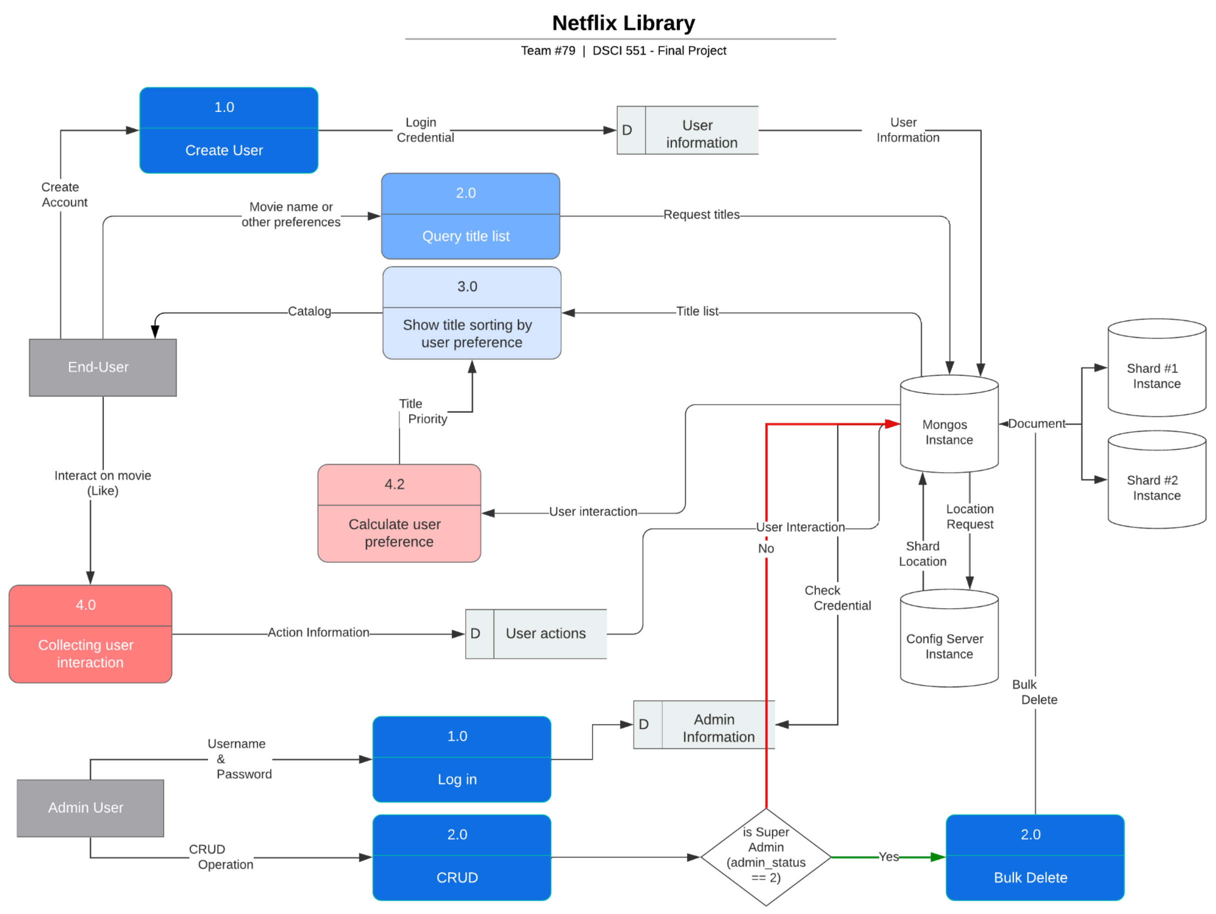
Database Specifications

- Database: MongoDB
- Hosting platform: EC2 Ubuntu instances
- MongoDB sharding:
    - 2 shard servers (3 replicas each)
    - 1 config server (3 replicas)
    - 1 mongos server for application access
- Database collections:
    - movies
    - users
    - user_interactions

Frontend Specification

- Libraries: HTML, CSS, Javascript, flask
- Host: Github
- 3 pages: Home (search), recommendation, catalog

Architecture Design



End-User

1. Login
    a. User can input their username and pass after creating their own account
2. Home page
    a. Query for movies
        i. Search bar: word/phrase
        ii. Preference search: input preferred type, genre, actor and director names
    b. Recommendation page
        i. Will show the top 3 movies by imdb_score that matches their query
3. Catalog
    a. Scrollable catalog showing 100 top movies sorted by imdb score
    b. Allows user to add their favorite movies to the system for future uses like recommendation engine
    c. Will update the catalog with new movies after users added their faves

Admin-User

1. Login (1.0)
    a. Admin Login: Admin users begin their interaction with the system by logging in through a secure authentication process. They enter their username and password, which the system validates.
    b. Credential Check: Upon entering the credentials, the system checks these against the hashed credentials stored in the database.
    c. Admin Information: If the credentials are verified, the system retrieves the admin's specific information to determine their level of access and privileges within the system.
2. CRUD Operations (2.0)
    a. Database Interaction: Once logged in, admin users have the ability to perform Create, Read, Update, and Delete (CRUD) operations on the data stored within the system.
    b. Accessing Shard Instances: The CRUD operations interact with data that may be distributed across multiple shard instances, which will be handled by the config server
3. Bulk Delete (2.0)
    a. Special Privileges: Admin users with higher privileges, such as Super Admins (admin status == 2), have the capability to perform bulk delete operations.
    b. Bulk Operations: This process involves sending delete commands that could affect multiple records simultaneously. These operations are managed through the Mongos instance, which coordinates with the shard instances and the config server to ensure data consistency and integrity across the distributed database system.
4. Integration with Config Servers and Shard Instances
    a. Database Interaction: Data operations from admin actions will be sent to mongos router to process location by config server, then stored or modified in shard cluster of a certain index

Implementation

Functionalities

Front-end application

- Authentication:
    - Direct users to the app and remind admin to login through another pipeline
- Query for movies:
    - Search bar: Uses regular expression to query for movies from the MongoDB database with text received from the search bar

    - Preference boxes: Gather all inputs from the dropdown boxes to query for movies featuring the actor or director
- Catalog
    - Query the top 100 movies by score to show in the catalog
    - When user select movies and submit, system will gather the movie_id from selected titles
    - Then, system will input movie_id and user_id combination as an interaction
    - Constraint: combination of movie_id and user_id must be unique

Code Implementation (src/user_app)

Python scripts
- User_run.py: Initialize Flask and blueprint
- Views.py: Bulk of implementation for all pages with Flask
- Config.py: App secret key
- Forms.py: User login form

Static files
- Style_catalog.css: Style for catalog page
- Style_home.css: Style for home page
- Style_rec.css: Style for recommendation page
- Style.css: Style for login page/Duplicate with admin login style

Template files
- Login.html: Login page/Duplicate with admin login
- Index.html: Query page
- Recommendations.html: Recommendation page
- Catalog.html: Catalog page

Db files
- Used the same database structure as discussed below for querying

Admin application

- User Authentication and Authorization
    - Ensures that only authenticated users can access the admin dashboard.
    - Supports different levels of access rights, allowing for distinctions between regular admin users and superadmins.
- CRUD Operations
    - Create: Admins can add new entries to the database, such as new movie listings, user accounts, and interaction records.
    - Read: Facilitates the retrieval of information from the database, enabling admins to view detailed lists and records of movies, user profiles, and interactions.
    - Update: Allows for the modification of existing database entries, which is crucial for keeping the database up to date with the latest movie information or user details.

    - Delete: Provides the capability to remove outdated or incorrect data from the database, helping maintain its cleanliness and relevance.
- Data Validation with MongoEngine
    - The admin application also incorporates the use of Mongoengine, a Document-Object Mapper (DOM) for working with MongoDB from Python. This integration plays a critical role in validating data before insertion or modification in the database, ensuring data integrity and adherence to the database schema.
    - Field Validation: This includes type checking, domain constraint validation
    - Reference Integrity: The use of ReferenceField in Mongoengine helps maintain the integrity of relationships between documents
    - Error Handling: When data validation fails, Mongoengine raises specific exceptions that can be caught and handled by the admin application
- Interactive Dashboards
    - Provides a rich interface with dashboards for quick access to data summaries in each collection with function to sort and filter in any condition.
    - Enables admins to quickly understand user behavior, popular movies, and system performance at a glance.
- Search and Filter Capabilities
    - Offers robust search functionality that allows admins to find specific movies or users within the database.
    - Includes advanced filtering options to streamline the management process, allowing admins to view data that meets specific criteria.
- Bulk Data Operations
    - Supports the import of data in bulk using a JSON-format file, which is essential for initializing the database or making mass insertion.
    - Supports the deletion of data in bulk using checkboxes, which is only available for superadmins.

Code Implementation (src/db)

- Python Scripts
    - __init__.py: Initializes the Flask application and sets up the blueprint that segregates admin functionality from other parts of the application.
    - views.py: Contains the route handlers that process requests and return responses. It includes routes for login, dashboard views, and operations like insert, edit, and delete records.
    - forms.py: Utilizes Flask-WTF to define forms for user input, such as login forms and data entry forms for managing movie data, ensuring validation and security.
    - config.py: Manages configuration settings for the admin app, which might include database connection parameters, secret keys, or other operational settings.
- Templates
    - base.html: Serves as the base template that includes common layout elements and scripts. All other templates extend this base.
    - login.html: Provides an HTML form for administrative user login, linking to authentication routes defined in views.py.

    - dashboard.html: Acts as the central hub for all administrative actions like viewing records in a tabular format, accessing forms to insert or edit data, and buttons for record management.
- Static Files
    - style.css: Defines the styling for the admin interface
    - scripts.js: Contains JavaScript code that adds interactive functionalities to the admin panel. This includes form submissions and all modal operations.

Database

- Database Collections
    - Movies: Stores detailed information about movies including title, description, genre, release date, imdb score, production country, etc.
    - Credits: Contains records linking movies to their cast and director. Each document details the role and character in a specific movie.
    - Users: Maintains user profiles, including username and hashed password
    - User Interactions: Records the actions that users take on the platform such as likes.
- Distribution
    - The database uses a sharded cluster setup to distribute data across multiple machines, enhancing both data management efficiency and query response times. The system uses two shards, each storing a portion of the dataset, facilitating horizontal scaling and load balancing.
    - A custom hash function is employed for sharding to ensure uniform data distribution across the shards
- Replication
    - Each shard operates with a replication factor of three, meaning each data piece in the shard is copied onto two other nodes in the same shard. This setup enhances data availability and fault tolerance, ensuring that the system can withstand the failure of a node without data loss.
    - The config server, critical for managing the cluster's metadata, also uses a three-node replica set to safeguard the cluster configuration data.
- Cloud Service
    - The database infrastructure is hosted on EC2 Ubuntu instances within the AWS cloud environment. These instances are equipped with Elastic IPs, ensuring that the addresses remain static even if the instances are stopped or restarted, thus maintaining reliable network accessibility.
- Security
    - Security groups in EC2 are configured to tightly control access to the instances. These security groups restrict inbound traffic to ensure that only authorized users can access the SSH for administration and the Mongos client for database operations.

Code Implementation (src/db)

- __init__.py
    - This file makes the folder a Python package and can initialize the database subsystem.
- config.py
    - Defines settings such as database URI, connection details, and other parameters critical for connecting to MongoDB.
- connection.py
    - Establishes and manages the connection to the database.
- models.py
    - Defines the data models or schema of the database using classes.
    - Uses MongoEngine to map these models to MongoDB documents.
- operations.py
    - Contains functions for performing CRUD operations on the database.
    - Also contains other operations such as: check user credentials, hash password, custom hash, check password
    - Key functions
        - insert_documents(collection_name, documents)
            - Objective: Inserts multiple documents into a specified MongoDB collection.
            - Usage: Batch data insertion tasks like adding bulk data from external sources.
        - find_document(collection_name, query)
            - Objective: Retrieves documents from a specified collection that match certain criteria.
            - Usage: Data retrieval operations, like fetching user details, searching for specific movies, or obtaining user interactions based on specific queries.
        - update_document(collection_name, query, update_data)
            - Objective: Updates documents in a MongoDB collection based on specified criteria
            - Usage: Used for modifying existing data
        - delete_document(collection_name, query)
            - Objective: Removes documents from a collection that match a given query.
            - Usage: Used for deletion

Technology Stack

Database

- MongoDB: Utilized as the primary database to store and manage movie, user, and interaction data.

Hosting Platform

- Amazon EC2 (Elastic Compute Cloud): to host the database ensuring scalability
- Ubuntu Instances: The operating system of choice for the EC2 servers.

Backend

- Flask: A micro web framework written in Python, used for building web applications.
- Python: Serves as the backend programming language, facilitating data manipulation and interaction with the database.

Frontend

- HTML5/CSS3: Used for structuring and designing the presentation of the web interface.
- JavaScript: Enhances interactivity on the client side, used in both admin and user applications for handling dynamic content and asynchronous requests.

Libraries/Frameworks

- Jinja2: A modern and designer-friendly templating language for Python, modeled after Django’s templates. Used in Flask applications to dynamically render HTML pages based on the backend data.
- MongoEngine: A Document-Object Mapper (DOM) for working with MongoDB from Python. It simplifies working with MongoDB documents through Python classes.

Deployment and Version Control

- Git: Used for version control, allowing multiple developers to work collaboratively and maintain the entire development history.

Deployment Architecture

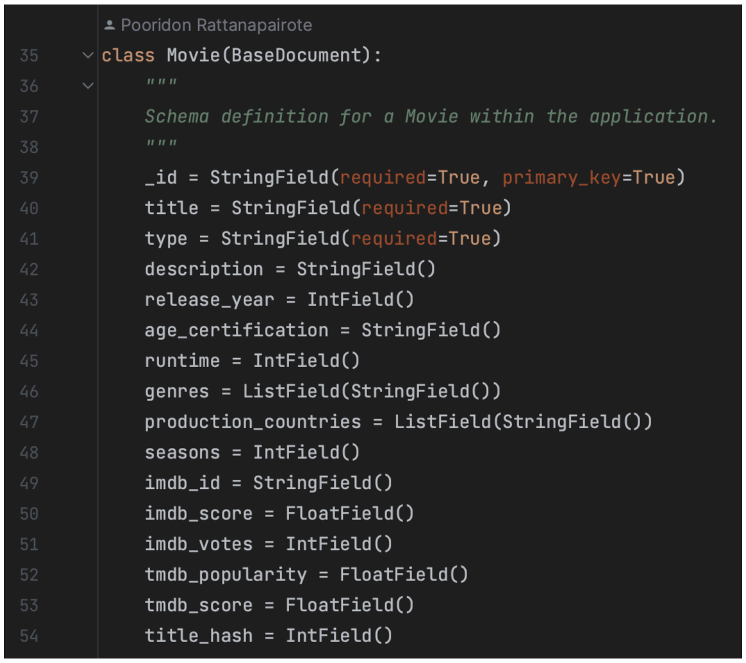
- Sharded Cluster: The MongoDB deployment uses sharding across multiple EC2 instances to distribute data and ensure fault tolerance.
- Shard Servers: Host the data shards with replication enabled.
- Config Servers: Maintain the metadata of the cluster, manage the topology, and facilitate the interaction between different shards.
- Mongos: Query routers that direct client requests to the appropriate shard.

Implementation Screenshots

src/db/

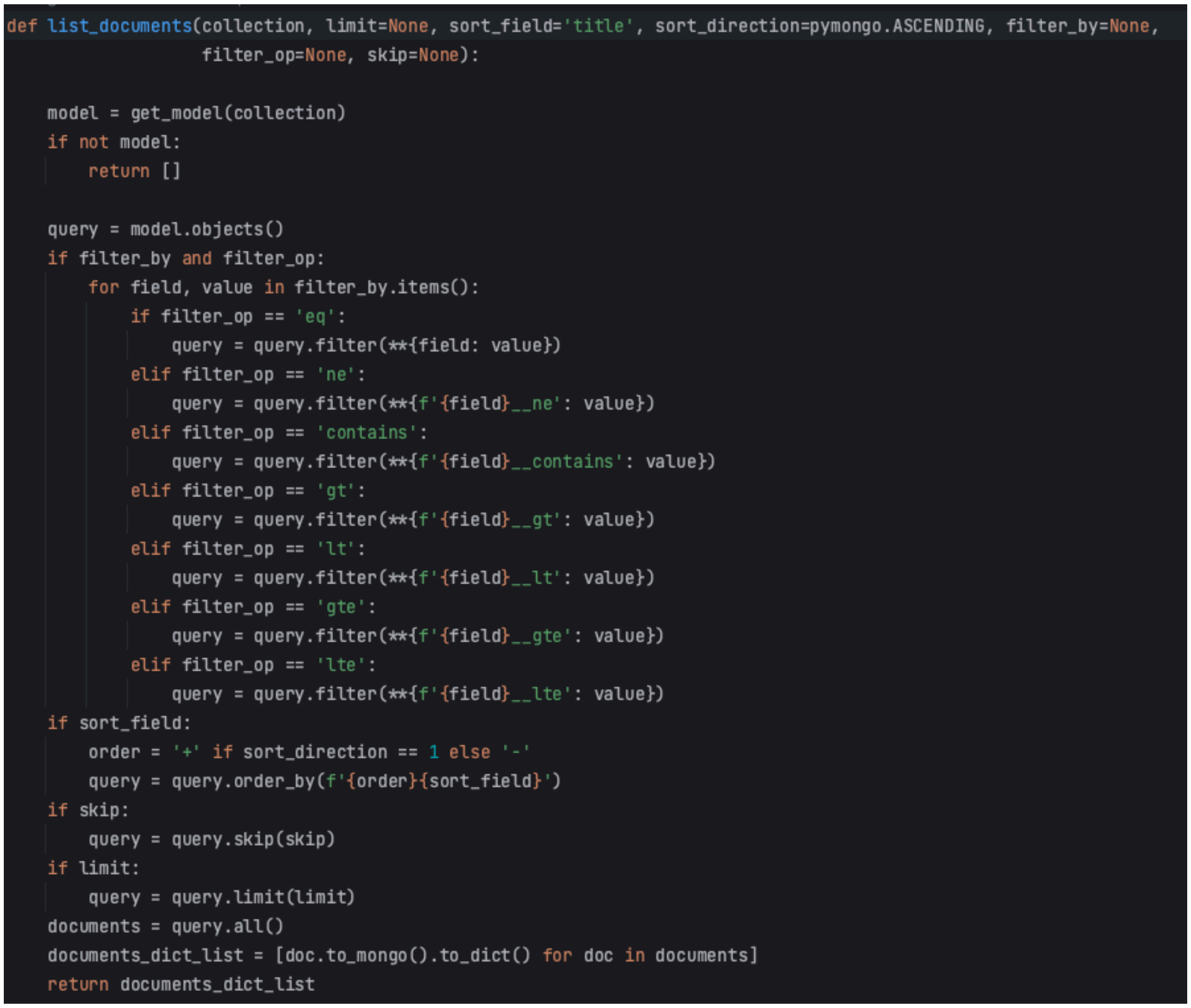
models.py

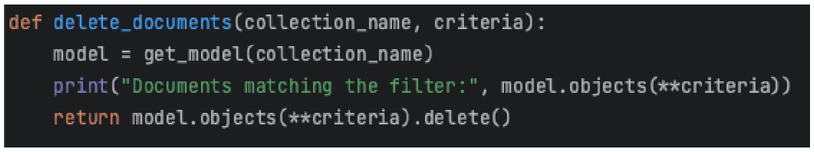
- Defines the data models or schema of the database using classes.

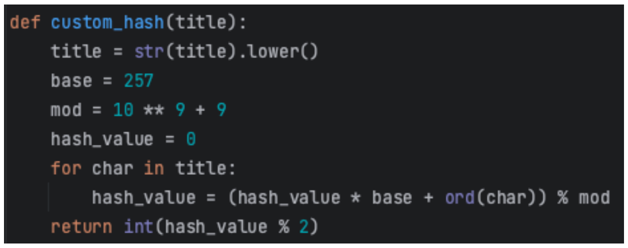


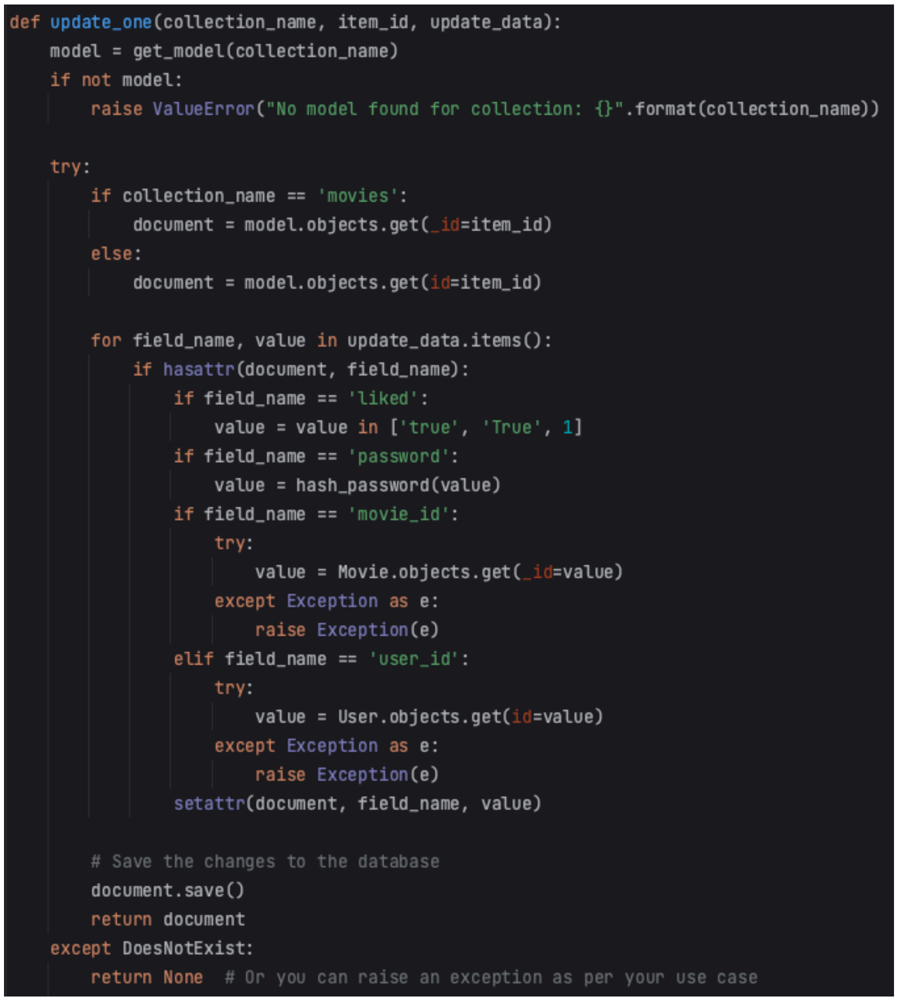
```python
class Movie(BaseDocument):
    """
    Schema definition for a Movie within the application.
    """
    _id = StringField(required=True, primary_key=True)
    title = StringField(required=True)
    type = StringField(required=True)
    description = StringField()
    release_year = IntField()
    age_certification = StringField()
    runtime = IntField()
    genres = ListField(StringField())
    production_countries = ListField(StringField())
    seasons = IntField()
    imdb_id = StringField()
    imdb_score = FloatField()
    imdb_votes = IntField()
    tmdb_popularity = FloatField()
    tmdb_score = FloatField()
    title_hash = IntField()
```

operations.py
- Contains functions for performing CRUD operations on the database.

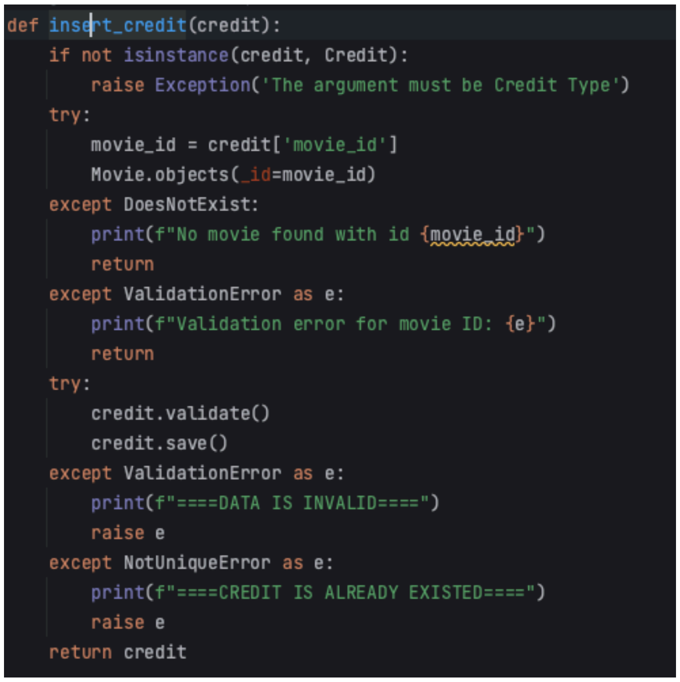

```python
def list_documents(collection, limit=None, sort_field='title', sort_direction=pymongo.ASCENDING, filter_by=None,
                   filter_op=None, skip=None):

    model = get_model(collection)
    if not model:
        return []

    query = model.objects()
    if filter_by and filter_op:
        for field, value in filter_by.items():
            if filter_op == 'eq':
                query = query.filter(**{field: value})
            elif filter_op == 'ne':
                query = query.filter(**{f'{field}__ne': value})
            elif filter_op == 'contains':
                query = query.filter(**{f'{field}__contains': value})
            elif filter_op == 'gt':
                query = query.filter(**{f'{field}__gt': value})
            elif filter_op == 'lt':
                query = query.filter(**{f'{field}__lt': value})
            elif filter_op == 'gte':
                query = query.filter(**{f'{field}__gte': value})
            elif filter_op == 'lte':
                query = query.filter(**{f'{field}__lte': value})
    if sort_field:
        order = '+' if sort_direction == 1 else '-'
        query = query.order_by(f'{order}{sort_field}')
    if skip:
        query = query.skip(skip)
    if limit:
        query = query.limit(limit)
    documents = query.all()
    documents_dict_list = [doc.to_mongo().to_dict() for doc in documents]
    return documents_dict_list
```


```python
def delete_documents(collection_name, criteria):
    model = get_model(collection_name)
    print("Documents matching the filter:", model.objects(**criteria))
    return model.objects(**criteria).delete()
```


```python
def custom_hash(title):
    title = str(title).lower()
    base = 257
    mod = 10 ** 9 + 9
    hash_value = 0
    for char in title:
        hash_value = (hash_value * base + ord(char)) % mod
    return int(hash_value % 2)
```



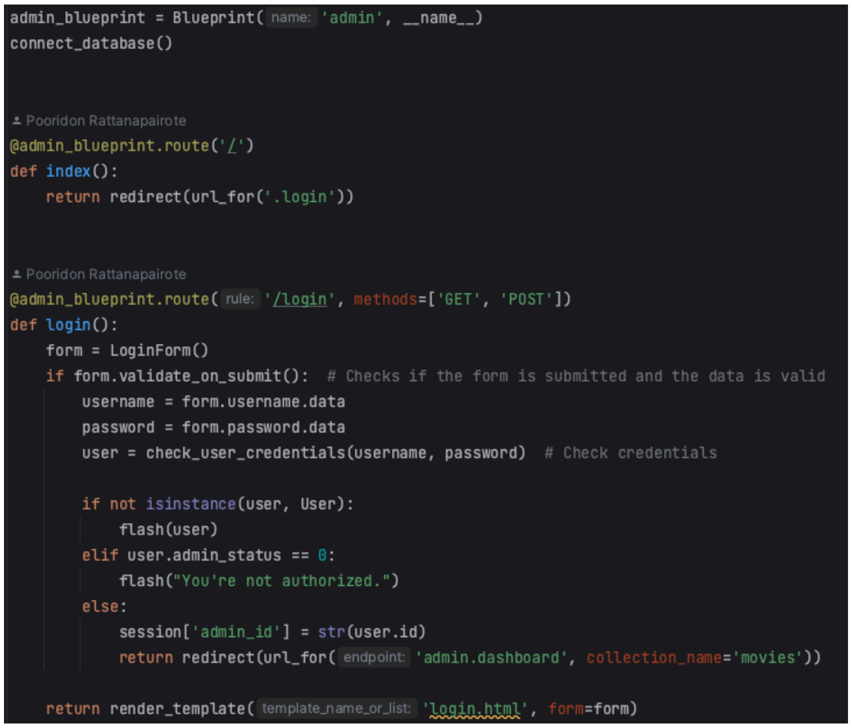
```python
def update_one(collection_name, item_id, update_data):
    model = get_model(collection_name)
    if not model:
        raise ValueError("No model found for collection: {}".format(collection_name))

    try:
        if collection_name == 'movies':
            document = model.objects.get(_id=item_id)
        else:
            document = model.objects.get(id=item_id)

        for field_name, value in update_data.items():
            if hasattr(document, field_name):
                if field_name == 'liked':
                    value = value in ['true', 'True', 1]
                if field_name == 'password':
                    value = hash_password(value)
                if field_name == 'movie_id':
                    try:
                        value = Movie.objects.get(_id=value)
                    except Exception as e:
                        raise Exception(e)
                elif field_name == 'user_id':
                    try:
                        value = User.objects.get(id=value)
                    except Exception as e:
                        raise Exception(e)
                setattr(document, field_name, value)

        # Save the changes to the database
        document.save()
        return document
    except DoesNotExist:
        return None  # Or you can raise an exception as per your use case
```



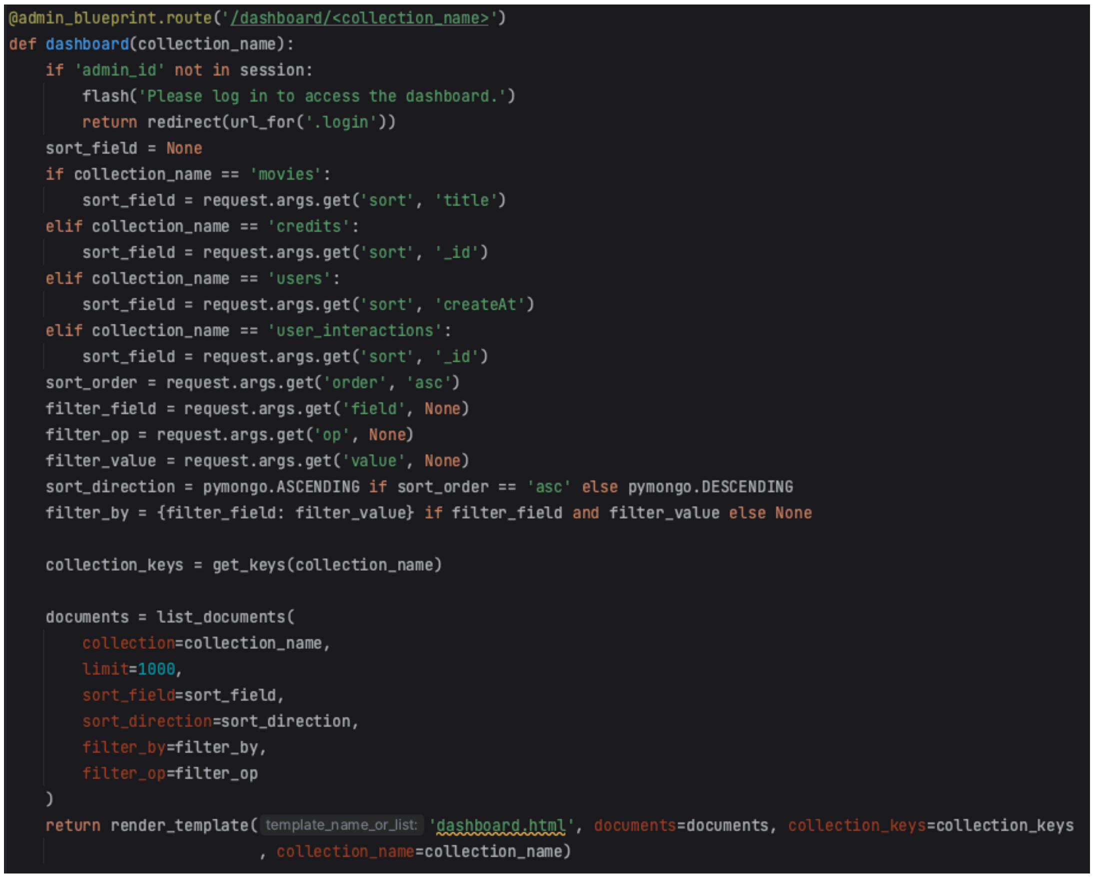
```python
def insert_credit(credit):
    if not isinstance(credit, Credit):
        raise Exception('The argument must be Credit Type')
    try:
        movie_id = credit['movie_id']
        Movie.objects(_id=movie_id)
    except DoesNotExist:
        print(f"No movie found with id {movie_id}")
        return
    except ValidationError as e:
        print(f"Validation error for movie ID: {e}")
        return
    try:
        credit.validate()
        credit.save()
    except ValidationError as e:
        print(f"====DATA IS INVALID====")
        raise e
    except NotUniqueError as e:
        print(f"====CREDIT IS ALREADY EXISTED====")
        raise e
    return credit
```

src/admin_app/

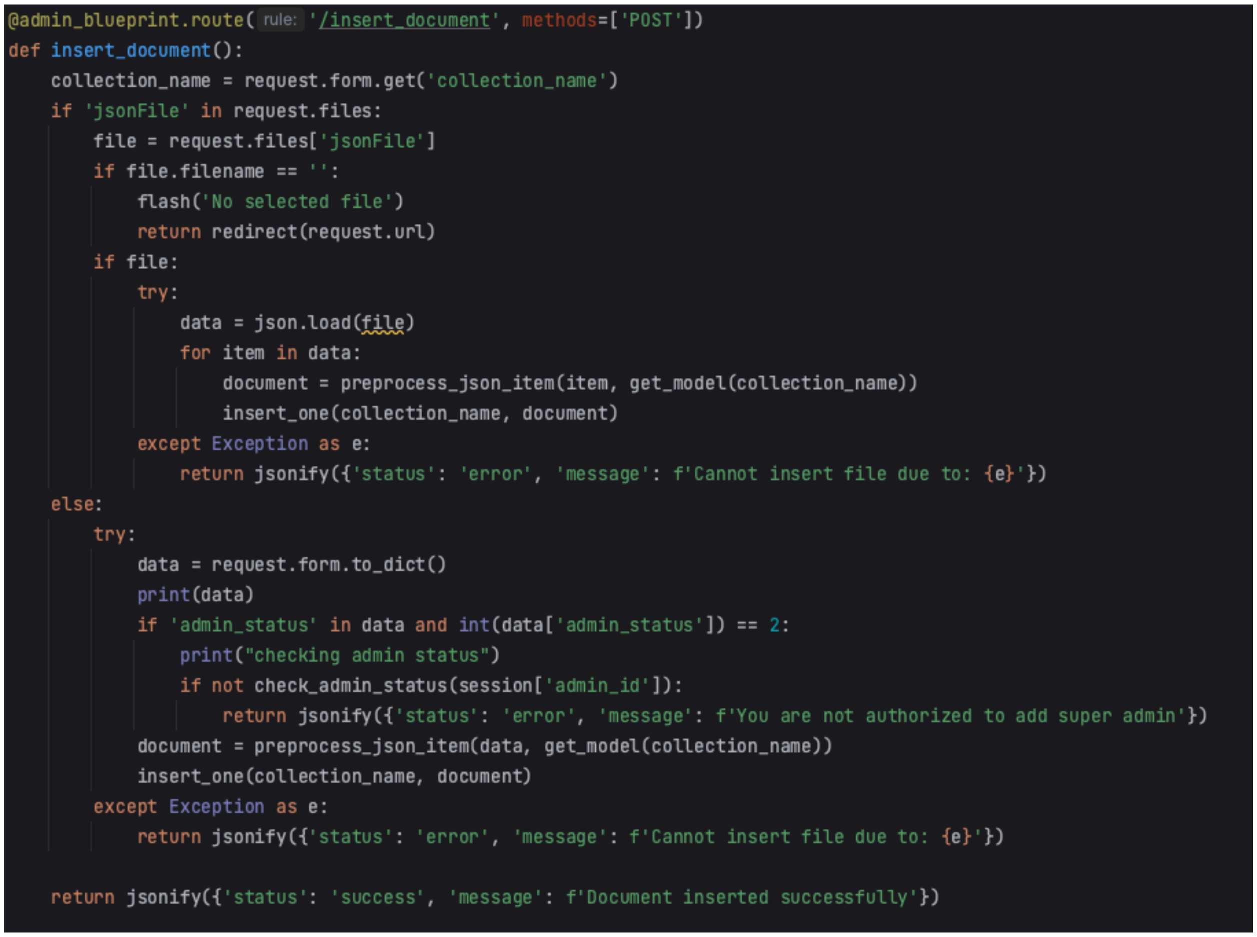
view.py



```python
admin_blueprint = Blueprint( name: 'admin', __name__)
connect_database()


 Pooridon Rattanapairote
@admin_blueprint.route('/')
def index():
    return redirect(url_for('.login'))


 Pooridon Rattanapairote
@admin_blueprint.route( rule: '/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():  # Checks if the form is submitted and the data is valid
        username = form.username.data
        password = form.password.data
        user = check_user_credentials(username, password)  # Check credentials

        if not isinstance(user, User):
            flash(user)
        elif user.admin_status == 0:
            flash("You're not authorized.")
        else:
            session['admin_id'] = str(user.id)
            return redirect(url_for( endpoint: 'admin.dashboard', collection_name='movies'))

    return render_template( template_name_or_list: 'login.html', form=form)
```



```python
@admin_blueprint.route('/dashboard/<collection_name>')
def dashboard(collection_name):
    if 'admin_id' not in session:
        flash('Please log in to access the dashboard.')
        return redirect(url_for('.login'))
    sort_field = None
    if collection_name == 'movies':
        sort_field = request.args.get('sort', 'title')
    elif collection_name == 'credits':
        sort_field = request.args.get('sort', '_id')
    elif collection_name == 'users':
        sort_field = request.args.get('sort', 'createAt')
    elif collection_name == 'user_interactions':
        sort_field = request.args.get('sort', '_id')
    sort_order = request.args.get('order', 'asc')
    filter_field = request.args.get('field', None)
    filter_op = request.args.get('op', None)
    filter_value = request.args.get('value', None)
    sort_direction = pymongo.ASCENDING if sort_order == 'asc' else pymongo.DESCENDING
    filter_by = {filter_field: filter_value} if filter_field and filter_value else None

    collection_keys = get_keys(collection_name)

    documents = list_documents(
        collection=collection_name,
        limit=1000,
        sort_field=sort_field,
        sort_direction=sort_direction,
        filter_by=filter_by,
        filter_op=filter_op
    )
    return render_template( template_name_or_list: 'dashboard.html', documents=documents, collection_keys=collection_keys
                           , collection_name=collection_name)
```

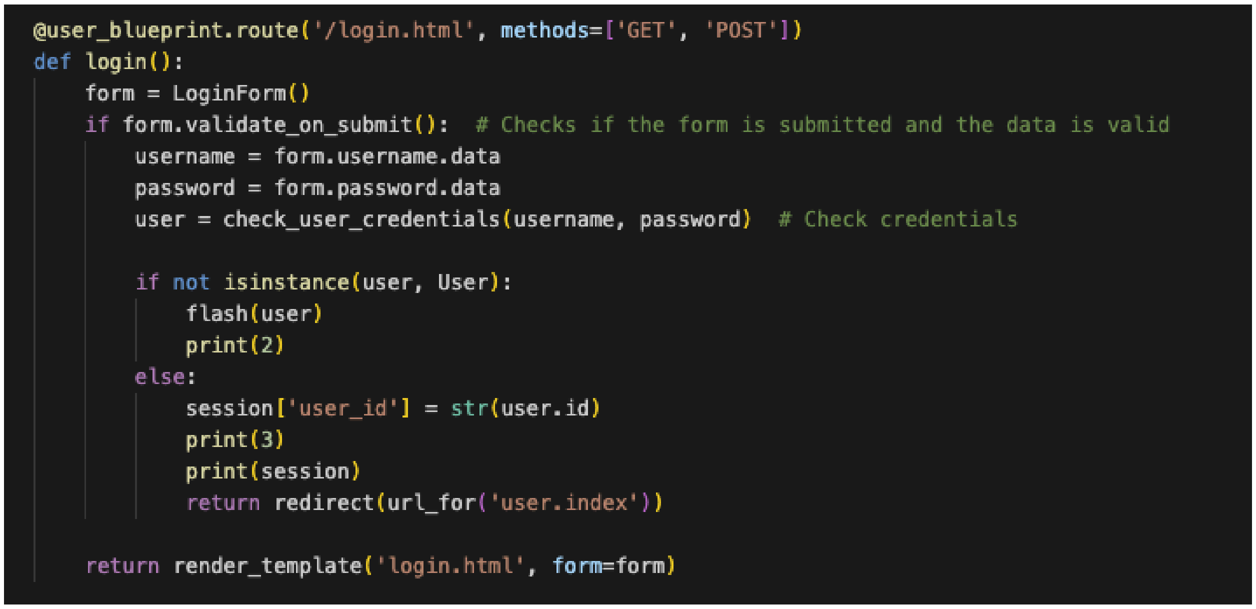


```python
@admin_blueprint.route( rule: '/insert_document', methods=['POST'])
def insert_document():
    collection_name = request.form.get('collection_name')
    if 'jsonFile' in request.files:
        file = request.files['jsonFile']
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if file:
            try:
                data = json.load(file)
                for item in data:
                    document = preprocess_json_item(item, get_model(collection_name))
                    insert_one(collection_name, document)
            except Exception as e:
                return jsonify({'status': 'error', 'message': f'Cannot insert file due to: {e}'})
    else:
        try:
            data = request.form.to_dict()
            print(data)
            if 'admin_status' in data and int(data['admin_status']) == 2:
                print("checking admin status")
                if not check_admin_status(session['admin_id']):
                    return jsonify({'status': 'error', 'message': f'You are not authorized to add super admin'})
            document = preprocess_json_item(data, get_model(collection_name))
            insert_one(collection_name, document)
        except Exception as e:
            return jsonify({'status': 'error', 'message': f'Cannot insert file due to: {e}'})

    return jsonify({'status': 'success', 'message': f'Document inserted successfully'})
```

src/users_app/

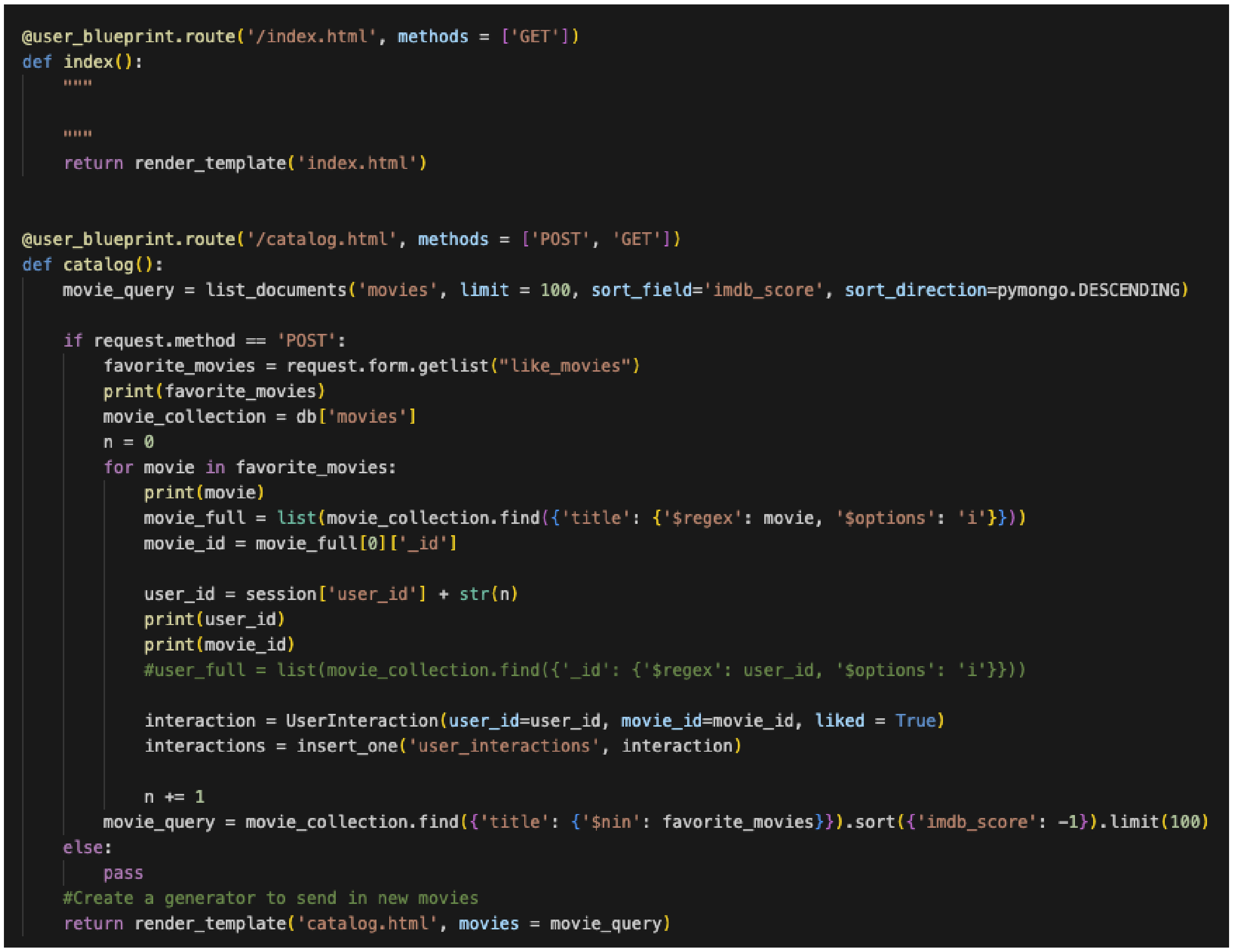
Screenshots

Views.py (contain the most important code as it connects the front-end with the back-end)
Login code:



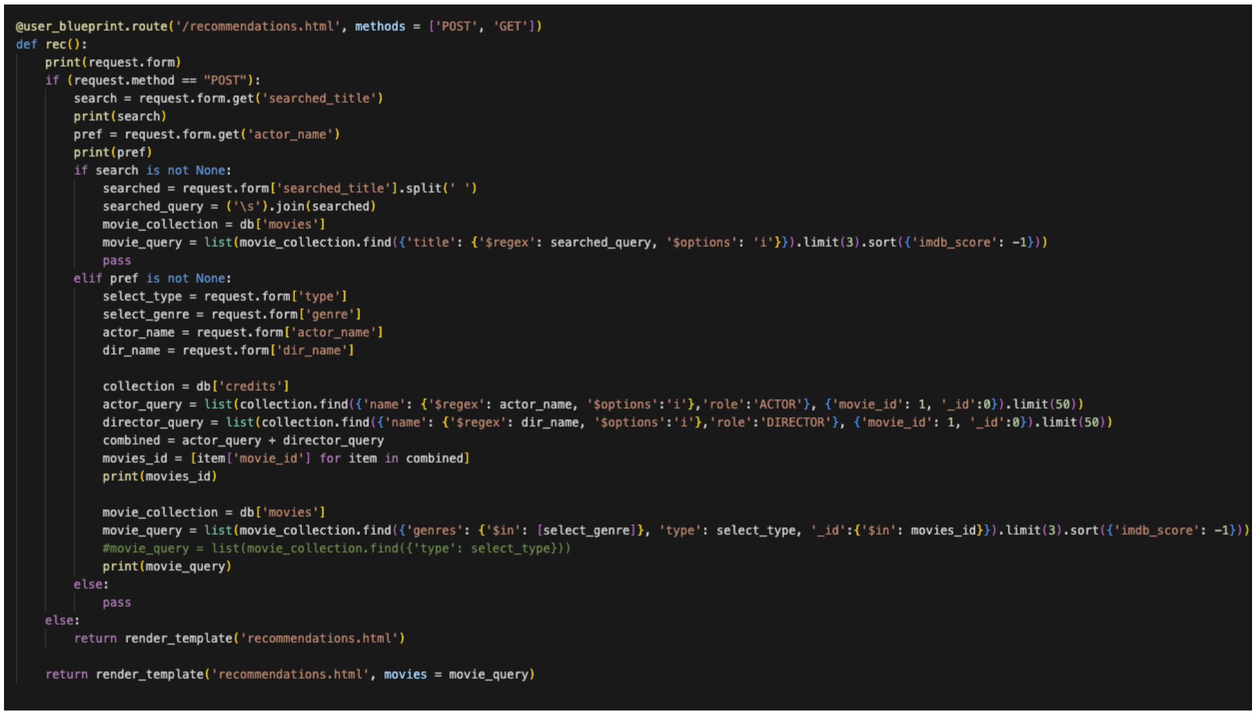
```python
@user_blueprint.route('/login.html', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():  # Checks if the form is submitted and the data is valid
        username = form.username.data
        password = form.password.data
        user = check_user_credentials(username, password)  # Check credentials

        if not isinstance(user, User):
            flash(user)
            print(2)
        else:
            session['user_id'] = str(user.id)
            print(3)
            print(session)
            return redirect(url_for('user.index'))

    return render_template('login.html', form=form)
```

Home & Catalog code:



```python
@user_blueprint.route('/index.html', methods = ['GET'])
def index():
    """

    """
    return render_template('index.html')


@user_blueprint.route('/catalog.html', methods = ['POST', 'GET'])
def catalog():
    movie_query = list_documents('movies', limit = 100, sort_field='imdb_score', sort_direction=pymongo.DESCENDING)

    if request.method == 'POST':
        favorite_movies = request.form.getlist("like_movies")
        print(favorite_movies)
        movie_collection = db['movies']
        n = 0
        for movie in favorite_movies:
            print(movie)
            movie_full = list(movie_collection.find({'title': {'$regex': movie, '$options': 'i'}}))
            movie_id = movie_full[0]['_id']

            user_id = session['user_id'] + str(n)
            print(user_id)
            print(movie_id)
            #user_full = list(movie_collection.find({'_id': {'$regex': user_id, '$options': 'i'}}))

            interaction = UserInteraction(user_id=user_id, movie_id=movie_id, liked = True)
            interactions = insert_one('user_interactions', interaction)

            n += 1
        movie_query = movie_collection.find({'title': {'$nin': favorite_movies}}).sort({'imdb_score': -1}).limit(100)
    else:
        pass
    #Create a generator to send in new movies
    return render_template('catalog.html', movies = movie_query)
```

Recommendation code:



```python
@user_blueprint.route('/recommendations.html', methods = ['POST', 'GET'])
def rec():
    print(request.form)
    if (request.method == "POST"):
        search = request.form.get('searched_title')
        print(search)
        pref = request.form.get('actor_name')
        print(pref)
        if search is not None:
            searched = request.form['searched_title'].split(' ')
            searched_query = ('\s').join(searched)
            movie_collection = db['movies']
            movie_query = list(movie_collection.find({'title': {'$regex': searched_query, '$options': 'i'}}).limit(3).sort({'imdb_score': -1}))
            pass
        elif pref is not None:
            select_type = request.form['type']
            select_genre = request.form['genre']
            actor_name = request.form['actor_name']
            dir_name = request.form['dir_name']

            collection = db['credits']
            actor_query = list(collection.find({'name': {'$regex': actor_name, '$options':'i'},'role':'ACTOR'}, {'movie_id': 1, '_id':0}).limit(50))
            director_query = list(collection.find({'name': {'$regex': dir_name, '$options':'i'},'role':'DIRECTOR'}, {'movie_id': 1, '_id':0}).limit(50))
            combined = actor_query + director_query
            movies_id = [item['movie_id'] for item in combined]
            print(movies_id)

            movie_collection = db['movies']
            movie_query = list(movie_collection.find({'genres': {'$in': [select_genre]}, 'type': select_type, '_id':{'$in': movies_id}}).limit(3).sort({'imdb_score': -1}))
            #movie_query = list(movie_collection.find({'type': select_type}))
            print(movie_query)
        else:
            pass
    else:
        return render_template('recommendations.html')

    return render_template('recommendations.html', movies = movie_query)
```

Learning Outcomes

Challenges Faced

- Database Switching
  - Initially, the project was set up using MongoDB Atlas, but due to new requirements from our instructor, we had to rebuild the database infrastructure from scratch. This task included reconfiguring servers, re-establishing sharding, and re-establishing client connections, all of which extended our timeline and added complexity to the project setup.
- Hosting MongoDB on EC2
  - Our team, not having access to high-specification local machines, opted to host our database on an EC2 instance. This decision allowed for better accessibility across various devices and user accounts.
  - However, setting up the MongoDB cluster with replication and sharding on Amazon EC2 was a significant challenge. We had to install mongodb on each of the four instances and configure them to communicate with one another.
- Security Breach by Hacker
  - An unexpected and severe challenge was a security breach. Initially, our database was accessible without IP restrictions, which led to unauthorized

access. A script was run against our database that deleted significant amounts of data and attempted to extort ransom by threatening further data loss.
    - Afterward, we swiftly implemented a security group within the EC2 management console to restrict access. We set up rules to allow only specific IP addresses to access our database via the Mongos client and SSH, significantly tightening security to prevent further incidents.

Individual Contribution (for multi-person team)

Pooridon

- MongoDB on EC2 implementation
- Admin Application

Pannawat

- User interface frontend and backend

Conclusion

The project for creating a distributed movie database system has been a comprehensive exercise in applying data management principles to a real-world application. This system is designed to handle massive amounts of data across multiple collections including movies, users, credits, and user interactions, utilizing MongoDB's capabilities for sharding and replication to ensure scalability and availability.

Hosted on Amazon EC2 Ubuntu instances, the project incorporates advanced database management techniques such as custom hashing for effective sharding and security measures to protect against unauthorized access. The development of both an admin and a user application allows for complete CRUD functionality as well as user engagement with the data through a web interface.

This project has not only solidified the team’s understanding of database systems and cloud services but has also set a foundation for future exploration in the field of data management.

Future Scope

- Recommendation Model
    - Initially we tried to add a recommendation engine into the model but realized that we lacked the data to build a good one so we decided to delay that feature to another date

- Poster Image
    - We want to add posters to show up for each movies but did not figure out how to store images in a distributed manner so we are excited to look into that in the future
- Bulk Edit
    - While the current system supports bulk insertion and deletion capabilities, a natural progression is to develop bulk editing functionality. This feature would allow admin users to update multiple records simultaneously, which is especially useful for large datasets where batch updates are common.
- Data Overload Control
    - As the database grows, the load time for displaying extensive data in the dashboard increases, leading to a less responsive user experience. To address this issue, the project aims to implement a data overload control mechanism. The envisioned solution involves:
        - Partial Data Loading: Initially loading a subset of data on the dashboard to ensure that the system remains responsive.
        - On-demand Data Loading: Integrating a mechanism to load additional data as needed, possibly through scrolling or pagination controls.