## Frontend Integration:

1. **Frontend Interaction with the API:**

The frontend can interact with the API using **HTTP GET requests** to fetch book recommendations based on the user’s ID. You can send the user’s ID in the request URL.

**Endpoint**: /api/recommendations/{user_id}

Example request:

```
  fetch(`/api/recommendations/${userId}`, {
  method: 'GET',
  headers: {
    'Authorization': `Bearer ${accessToken}`
  }
})
.then(response => response.json())
.then(data => {
  // Handle the recommended books here
  console.log(data);
})
.catch(error => {
  // Handle errors (e.g., no recommendations available)
  console.error('Error fetching recommendations:', error);
});
```

**Error Handling**:
In case of errors (e.g., no recommendations available or invalid user):

The frontend should handle this scenario gracefully, showing an appropriate message to the user, like "No recommendations found" or "Invalid user."

1. **Displaying Recommended Books**:
   The frontend will need to display the recommended books in a user-friendly format. Each book's details can be presented in a list or grid format.
   - **Display Items**:
     - Book Title
     - Author
     - Rating (if available)
     - Year of Publication
     - Image (if available)

## Backend Integration:

1. **Models Used for Generating Recommendations:**
   - **Collaborative Filtering**:
     The backend uses **SVD (Singular Value Decomposition)**, a technique from the surprise library, to perform collaborative filtering. This model predicts

ratings based on historical user-item interactions, providing personalized recommendations.

- **Integration**:
  - The model is trained on the user-book ratings data (ratings_cleaned.csv). The ratings dataset is loaded, pre-processed, and used to train the SVD model.
  - For each user, the backend predicts ratings for books that the user has not rated yet, sorting them by predicted rating and returning the top N recommendations.
- **Content-Based Filtering (optional)**:
  You can further enhance the model with content-based filtering by including book features (e.g., genre, author, etc.) to refine the recommendations. Currently, this API is focused on collaborative filtering.

2. **Configuration Settings**:
   - **Data Files**:
     Ensure the datasets (ratings_cleaned.csv and books_cleaned.csv) are available in the specified file paths.
   - **Dependencies**:
     The server environment should have the following dependencies installed:

     ```
     pip install flask flask_jwt_extended flask_redis flask_limiter surprise pandas
     ```

## API Rate Limiting :

To prevent overloading the server, the API can implement rate limiting. For example, you may want to limit each user to 100 requests per hour.

1. **Rate Limiting Setup**:
   Use the flask_limiter package to set up rate limiting:

   ```
   limiter = Limiter(get_remote_address, app=app)
   @app.route('/api/recommendations/<user_id>', methods=['GET'])
   @limiter.limit("100 per hour")  # Limit to 100 requests per user per hour
   def get_recommendations(user_id):
   ```

**Error Handling for Rate Limiting**:
If the rate limit is exceeded, the user will receive a 429 Too Many Requests response.

## Caching:

Caching recommendations can greatly improve performance by reducing the need to regenerate recommendations for frequently-requested users.

1. **Caching Recommendations**:
   Redis can be used to cache the recommended books for each user. When a user requests recommendations, the system checks if their recommendations are cached. If

cached, it returns the cached result; otherwise, it generates the recommendations and stores them in Redis.

2. **Caching Setup**:
    - Use **FlaskRedis** to interact with the Redis server.
    - Cache the recommendations for 1 hour (3600 seconds):

```
@app.route('/api/recommendations/<user_id>', methods=['GET'])

def get_recommendations(user_id):
    # Check Redis cache first
    cached_recommendations = redis.get(f"recommendations:{user_id}")
    if cached_recommendations:
        return jsonify(json.loads(cached_recommendations))

    # If not cached, generate recommendations
    recommendations = generate_recommendations(user_id)

    # Cache the recommendations for 1 hour
    redis.setex(f"recommendations:{user_id}", 3600, json.dumps(recommendations))

    return jsonify(recommendations)
```

3. **Cache Expiration**:
   The cache expires after 1 hour, ensuring that recommendations are up-to-date but also reducing unnecessary computation.